Supporting Diverse Notations in MPS' Projectional Editor

Markus Voelter[1] and Sascha Lisson[2]

[1] independent/itemis, voelter@acm.org
[2] itemis AG, lisson@itemis.de

Abstract. To be able to build effective DSLs, these DSLs must not just use language concepts that are aligned with their respective domain, but also use notations that correspond closely to established domain notations – and those are often not purely textual or graphical. The underlying language workbench must support these notations, and combining different notations in a single editor must be supported as well in order to support the coherent definitions of systems that use several DSLs. In this paper we provide an overview over the notations supported by JetBrains MPS. MPS is a language workbench that uses a projectional editor, which, by its very nature, can deal with many different notational styles, including text, prose, math tables and graphics. The various supported notations are illustrated with examples from real-world systems.

1 Introduction

The GEMOC 2014 workshop description states: *To cope with complexity, modern software-intensive systems are often split in different concerns, which serve diverse stakeholder groups and thus must address a variety of stakeholder concerns. These different concerns are often associated with specialized description languages and technologies, which are based on concern-specific problems and solution concepts.* In particular, these different concerns also require different notations. Ideally, these notations are closely aligned with existing domain-specific notations used by the stakeholders. However, such existing notations are not necessarily just text: they use forms, diagrams, mathematical symbols, a mix of prose and structure or combinations of those. Representing such diverse notations faithfully requires a high degree of flexibility in the kinds of editors that can be built with the language workbench used to create the languages.

Projectional editing (see Section 2) allows creating editors that can use a wide variety of notations, including the ones mentioned above. In particular, it can also mix these notations seamlessly, leading to a more faithful representation of existing domain languages in tools. JetBrains MPS is one of the leading projectional editors, and this paper describes its capabilities in terms of notational flexibility.

Contribution This paper provides an overview over the notational styles currently supported by MPS. For each style we discuss why it is useful, where it is being used as well as some details about how to define the respective editors.

Availability of the Code JetBrains MPS is open source software available from `http://jetbrains.com/mps`. Also, those editor facilities that are separate plugins to MPS are open source software and their repositories are indicated in each case. The examples shown in this paper are mostly based on mbeddr [1] and are open source as well. The screenshots in Figures 7, 9 and 10 are taken from a commercial tool currently being developed by Siemens PLM software; however, the underlying editor facilities are all open source as well.

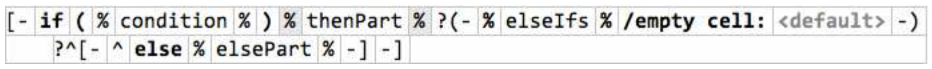
Structure In the next section we provide a brief overview over MPS' projectional editor and show briefly how to implement regular text editors. Section 3 introduces the fundamental notations supported by MPS and Section 4 discusses other useful features of the MPS editor. We conclude the paper with a brief discussion and summary in Section 5.

2 Projectional Editing in MPS

What is Projectional Editing? In parser-based editors users type sequences of characters into a text buffer. The buffer is parsed to check whether the sequence of characters conforms to a grammar. The parser ultimately builds an abstract syntax tree (AST), which contains the relevant structure of the program, but omits syntactic details. Subsequent processing (linking, type checks, transformation) is based on the AST. Modern IDEs (re-)parse the concrete syntax while the user edits the code, continuously maintaining an up-to-date AST in the background that reflects the code in the editor's text buffer. However, even in this case, this AST is created by a parser-driven transformation from the source text.

A projectional editor does not rely on parsers. As a user edits a program, the AST is modified *directly*. Projection rules are used to create a representation of the AST with which the user interacts, and which reflects the resulting changes. No parser-based transformation from concrete to abstract syntax is involved. This approach is well-known from graphical editors: when editing a UML diagram, users do not draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates an instance of `uml.Class` when a user drops a class. A projection engine renders the class as a rectangle. As the user edits the program, program nodes are created as instances of language concepts. Programs are stored using a generic tree persistence format (such as XML).

The projectional approach can be generalized to work with any notation, including textual. A code-completion menu lets users create instances based on a text string entered in the editor called the *alias*. The aliases allowed in any given location depend on the language definition. Importantly, *every next text string is recognized as it is entered*, so there is never any parsing of a sequence of text strings. In contrast to parser-based editors, where disambiguation is performed by the parser after a (potentially) complete program has been entered, in projectional editors disambiguation is performed by the user as he selects a concept from the code-completion menu. Once a node is created, it is *never* ambiguous, *irrespective of its syntax*: every node points to its defining concept. Every program node has a unique ID, and references between program elements are represented as references to the ID. These references are established during



```
[- if ( % condition % ) % thenPart % ?(- % elseIfs % /empty cell: <default> -)
    ?^[- ^ else % elsePart % -] -]
```

Fig. 1. Editor definition for the `IfStatement` (details in the running text).

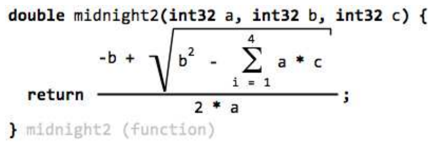
program editing by directly selecting reference targets from the code-completion menu; the references are persistent. This is in contrast to parser-based editors, where a reference is expressed as a string in the source text, and a separate name resolution phase resolves the target AST element after the text has been parsed.

Projectional editing has two advantages. First, it supports flexible composition of languages because the ambiguities associated with parsers cannot happen in projectional editors. We do not discuss this aspect in this paper and refer the reader to [2]. The other advantage of projectional editors is that, since no parsing is used, the program notation does not have to be parseable and a wide range of notations can be used. This paper focusses on this aspect. Traditionally, projectional editors have also had disadvantages relative to editor usability and infrastructure integration; those are discussed in [3].

Defining a Simple Editor In order for the reader to better understand the explanations in Sections 3 and 4, this section briefly introduces the MPS structure and editor definitions. MPS' meta model is similar to EMF Ecore [4]. Language concepts (aka meta classes) declare children (single or lists), references and primitive properties. Concepts can extend other concepts or implement concept interfaces; subtype polymorphism is supported. Programs are represented as instances of concepts, called nodes. Each concept also defines one or more editors. These are the projection rules that determine the notation of instance nodes in the program. Editor definitions consist of cells arranged in various layouts. A cell can be seen as an atomic element of an editor definition. As an example, let us consider the `if` statement in C. Its structure is defined as follows:

```
concept IfStatement extends Statement
  alias: if
  children:
    condition: Expression[1]         elsePart:  StatementList[0..1]
    thenPart:  StatementList[1]      elseIfs:   ElseIfPart[0..n]
```

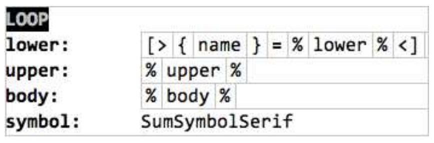
Fig. 1 shows the editor definition for the `IfStatement` concept. At the top level, it consits of a collection cell `[- .. -]` which aligns a sequence of additional cells in some particular way – a linear sequence in this case. The sequence starts with the constant (keyword) `if` and a pair or parentheses. Between those, the editor projects the `condition` expression; the `%` sign is used to refer to children of the current concept. The `thenPart` follows, and since it is a `StatementList`, it comes with its own curly braces. The `(- ... -)` collection captures the list of `else if` parts, if any. The `ElseIfPart` comes with its own editor which is embedded here. Finally, there is an optional set of cells (represented by the `?` and a condition expression that is not shown) that contains the `else` keyword as well as the `elsePart` child. A flag (not shown) determines that the `else` part is shown on a new line, leading to the expected representation of `if` statements.



double midnight2(int32 a, int32 b, int32 c) {

return $\frac{-b + \sqrt{b^2 - \sum_{i = 1}^{4} a * c}}{2 * a}$;

} midnight2 (function)



LOOP	
lower:	[> { name } = % lower % <]
upper:	% upper %
body:	% body %
symbol:	SumSymbolSerif

Fig. 2. Mathematical symbols used in C expressions embedded into C functions.

Fig. 3. The definition of the sum symbol editor using the `LOOP` primitive.

3 Notations

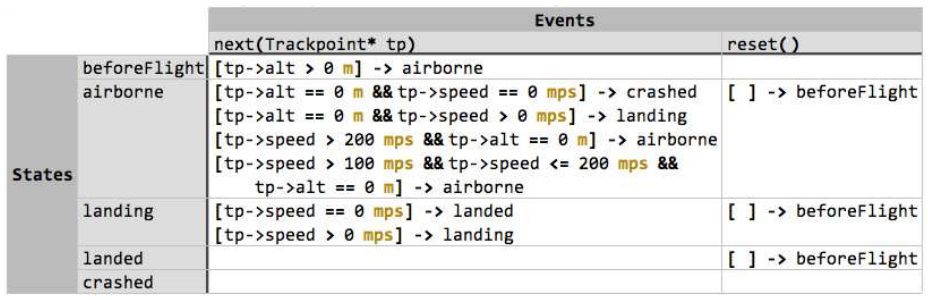
This section discusses the notations supported by MPS. For each we provide a rationale, an example and a hint on how to build editors that use the style.

Textual Notations The first notation supported by MPS has been textual notations. Notations used by programming languages such as Java, C or HTML can be represented easily. The example in the previous section shows how to create editors for textual notations. The backbone is the `indent layout` collection cell which can deal flexibly with sequences of nodes, newlines and indentation.

Mathematical Symbols A plugin [5] supports mathematical notations. The plugin comes with a set of new layout primitives (cell types) that enable typical mathematical notations such as fraction bars, big symbols (sum or product), roots and all kinds of side decorations (as used in `abs` or `floor`). The plugin contributes only the editor cells so they can be integrated into arbitrary languages. So far they have been integrated into C (Fig. 2) and an insurance DSL.

Fig. 3 shows the definition of the sum editor. It uses the new primitive `LOOP` which can be used for everything that has a big symbol as well as things above, below and right of the symbol. The particular symbol is defined separately and referenced from the editor definition. The `LOOP` cell has three slots (`lower`, `upper` and `body`) into which child nodes can be added. The `Sum` expression defines children `upper`, `body` and `lower`, which are mapped to these slots. These slots can contain arbitrary editor cells, not just child collections: the `lower` slot contains a collection that projects the `name` property, an equals sign and the `lower` child.

Tables Tables can be used to represent collections of structured data or to represent two-dimensional concerns. For example, Fig. 4 shows a state machine



		Events	
		next(Trackpoint* tp)	reset()
States	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 [tp->speed > 100 mps && tp->speed <= 200 mps && tp->alt == 0 m] -> airborne	[ ] -> beforeFlight
	landing	[tp->speed == 0 mps] -> landed [tp->speed > 0 mps] -> landing	[ ] -> beforeFlight
	landed		[ ] -> beforeFlight
	crashed		

Fig. 4. A state machine represented as a table; also shows nested headers.

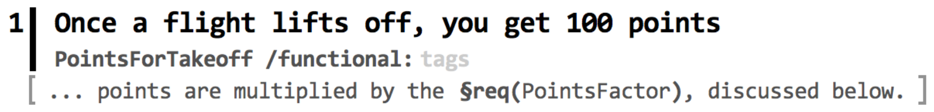
rendered as a table. Another example used in mbeddr [1] is decision tables (essentially two nested `if` statements).

Tables come in several flavors. For example, a row-oriented table has a fixed set of columns and a variable list of rows. Users can add rows, but the columns are prescribed by the language definition. In contrast, the state machine shown in Fig. 4 is a cell-oriented table: users can add new columns (events), new rows (states) and new entries in the content cells (transitions). The language for defining tabular editors [6] takes these different categories into account. For example, the definition for the state machine uses queries over the state machine model to determine the set of columns and rows. The contents for the transition cells are also established via queries: each transition is a child of its source state and references the triggering event. Since both columns and rows can be added (or deleted) by the user, callbacks for adding and deleting both are implemented. The code below shows part of the table implementation for state machines.

```
table
  column headers:
    group "Events" {
      query {
        getHeaders (node)->join(string | EditorCell | node<> | Iterable) {
          node.inEvents(); }
        insert new header (node, index)->void { // callback for inserting }
        on delete: (node, index)->void { // callback for deleting }
    } }
  row headers: // similar
  cells:
    column count: node.inEvents().size;
    row count: node.states().size;
    cell: (node, columnIndex, rowIndex)->join(node<> | string | EditorCell | Iterable) {
      node<InEvent> evt = node.inEvents().toList.get(columnIndex);
      node<AbstractState> state = node.states().toList.get(rowIndex);
      node.descendants<Transition>.
        where({~it => it.parent==state && it.trigger.event==evt; });  } as vertical list
```

Prose with Embedded Code One characteristic of projectional editors is that the language structure strictly determines the structure of the code that can be written in the editor. While this is useful for code, it does not work for prose. Hence, an MPS plugin [7] supports "free text editing" in MPS: all the usual selection and editing actions known from text editors are supported. The resulting text is stored as a sequence of `IWord` nodes. By creating new concepts that implement the `IWord` concept interface, other specific nodes can be inserted into the sequence of words. Said differently, arbitrary structured program nodes can be embedded into the (otherwise unstructured) prose. The user can press `Ctrl-Space` anywhere in the prose block and insert instances of those concepts that are valid at this location. Fig. 5 shows an example of a requirement with a prose block that embeds a reference to another requirement. Other examples include references to arguments in function comments or embedded formulas.

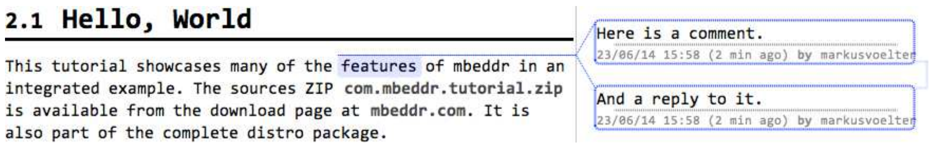
Margin Cells Margin cells are rendered beyond the right editor margin; each margin cell is associated with an anchor cell inside the editor, and the margin cell is rendered at the y-position of that anchor cell. Fig. 6 shows an example. To use margin cells in the editor of some concept, the editor for that concept embeds a `margincell` cell which points to the collection that contains the margin



```
1 Once a flight lifts off, you get 100 points
  PointsForTakeoff /functional: tags
[ ... points are multiplied by the §req(PointsFactor), discussed below. ]
```

Fig. 5. The prose block includes a sequence of ”normal” words plus a reference to another requirement (`§req(..)`). The reference is a real, navigable and refactoring-safe pointer, not just nice syntax.

contents (the comments in Fig. 6). The contents specified for the margin cell must implement the `IMarginCellContent` interface which contributes the facilities that connect the margin cell content to the anchor cell. Margin cells are available in the mbeddr.platform at `http://mbeddr.com`.



```
2.1 Hello, World

This tutorial showcases many of the features of mbeddr in an
integrated example. The sources ZIP com.mbeddr.tutorial.zip
is available from the download page at mbeddr.com. It is
also part of the complete distro package.
```

```
Here is a comment.
23/06/14 15:58 (2 min ago) by markusvoelter

And a reply to it.
23/06/14 15:58 (2 min ago) by markusvoelter
```

Fig. 6. Margin cells used to support Word-like comments in MPS; other contents can be projected into the margin as well.

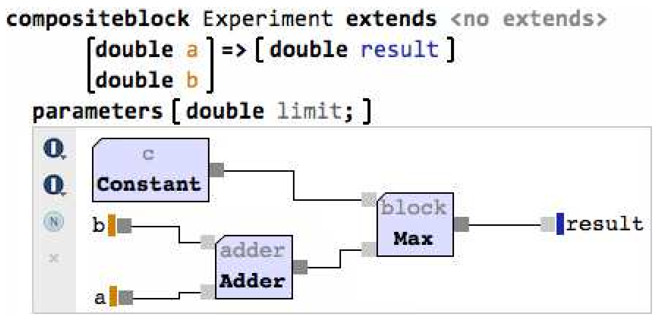
Graphics MPS supports editable graphical notations as shown in Fig. 7. They can be embedded into any other editor. MPS' support for graphical notations is new (available since MPS 3.1, June 2014) and not yet as mature as the rest of MPS, and the API for defining the editor is not yet as convenient as it should be. The code below shows part of the definition of the editor for Fig. 7: the contents of a block plus its ports are mapped as the contents of the diagram canvas.

```
diagram {
  content: this.contents,
           this.ancestor<Block>.allInPorts().toList,
           this.ancestor<Block>.allOutPorts().toList
  palette: custom AccentPaletteActionGroup
}
```

Custom Cells MPS supports embedding custom cells. This means that the user can plug in their own subclass of `CellProvider` and implement specific `layout` and `paint` methods. This way, any notation can be drawn in a low-level way. The cell provider can be parameterized, and ultimately, it can become a new, reusable primitive. Fig. 8 shows an example of a language that reports progress with work packages. It uses three custom cells: horizontal lines (parameterizable with thickness and color), check boxes (that are associated with boolean properties of the underlying language concept) and progress bars (whose percentage and color can be customized, typically by querying other parts of the model).

4 Other Features of the MPS Editor

As a language workbench, MPS supports the features known from traditional IDEs for custom languages. These include code completion, quick fixes, syntax



```
compositeblock Experiment extends <no extends>
      [double a ] => [ double result ]
      [double b ]
  parameters [ double limit; ]
```

Fig. 7. A graphical editor embedded in a regular text editor.

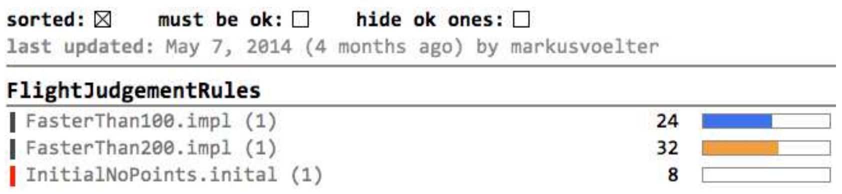
coloring, code folding, goto definition, find references and refactorings. In this section we describe editor features that are specific to MPS' projectional editor.

Mixed Notations The various notations discussed in the previous section can all be mixed arbitrarily (with the aforementioned exception of embedding things *into* graphical editors). Since all editors use the same projectional architecture this works seamlessly. In particular, non-textual notations can be used inside textual notations. Examples include:

– mathematical symbols embedded in textual programs
– tables that contain text or math symbols
– tables embedded in textual programs
– mathematical symbols embedded in prose
– lines, progress bar other other shapes embedded arbitrarily

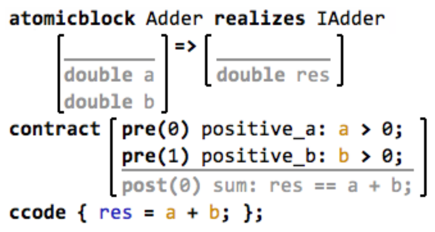
Multiple Editors A single concept can define several editors, and a given program can be edited using any of them. Each of the multiple editors has a tag, and by setting tags in an editor window (either by the user or programmatically), the editors corresponding to these tags can be selected. For example, state machines can be edited in a textual version (roughly similar to Fowler's state machine DSL [8]) or in the tabular notation shown in Fig. 4.

Partial Syntax Editors can also be partial in the sense that they do not project all contents stored in the AST. Of course the non-projected aspects of the program cannot be edited with this particular editor. But the contents remain stored in the AST (and are cut, copied, pasted or moved) and can be edited later. Using this facility, programs can be edited in ways specific to the current process



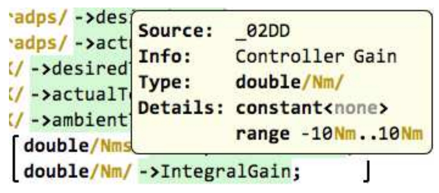
```
sorted: ☒    must be ok: ☐    hide ok ones: ☐
last updated: May 7, 2014 (4 months ago) by markusvoelter

FlightJudgementRules
| FasterThan100.impl (1)                         24
| FasterThan200.impl (1)                         32
| InitialNoPoints.inital (1)                      8
```

Fig. 8. Custom widgets (checkbox, line, progress bar) used in an MPS editor.



```
atomicblock Adder realizes IAdder
    [ ________ ] => [ ___________ ]
    [ double a ]    [ double res  ]
    [ double b ]
contract [ pre(0) positive_a: a > 0; ]
         [ pre(1) positive_b: b > 0; ]
         [ ___________________________ ]
         [ post(0) sum: res == a + b;  ]
ccode { res = a + b; };
```

Fig. 9. A querylist is used to project the ports and contracts inherited from the interface realized by this block (in grey). New nodes ports or contracts can be entered above the grey lines.



```
'adps/ ->des
'adps/ ->act
(/ ->desired
(/ ->actualT
(/ ->ambient
[double/Nms
[double/Nm/ ->IntegralGain;      ]

Source:   _02DD
Info:     Controller Gain
Type:     double/Nm/
Details:  constant<none>
          range -10Nm..10Nm
```

Fig. 10. This tooltip shows the definition of the quantity referenced via the -> notation: it shows its type and various additional details. The tooltip uses the querylist to project derived nodes.

step or stakeholder. An example are the requirements traces shown in Fig. 14; programs can be shown with or without them.

Query List An MPS editor normally displays nodes at their phyiscal location. For example, the child `condition` of the `IfStatement` shown in Fig. 1 is projected as part of its parent editor. Sometimes, however, it is useful to render nodes in other places. An example is shown in Fig. 9: the grey parts are defined by the interface realized by the block, but they are still projected for the block itself.

To project nodes in locations where they are not defined, a `querylist` editor cell is used (available as part of the mbeddr.platform at `http://mbeddr.com`). Like other MPS collection cells it projects a list of nodes, but this list is assembled via an arbitrary model query. The result can be projected read-only (as in Fig. 9) or fully editable. The querylist also supports callback functions for adding new nodes (because it is not automtically clear where they would have to be inserted physically) or for deleting existing ones. This way, querylists support *views*.

Tooltips MPS can use the projectional editing facilities in tooltips (available in the mbeddr.platform). To define a tooltip, a special cell is inserted into the editor of the cell that should display the tooltip. Since the purpose of a tooltip often is to project information gathered from other parts of the model, tooltip editors often use querylists (Fig. 10).

Conditional Editors Conditional editors essentially support aspect orientation for editor cells. A conditional editor defines a decoration for existing editors as well as a pointcut that determines to which existing editor cells the decoration is applied. Figures 12 and 13 show examples. Importantly, these conditional editors can be defined *after the fact*, and potentially in a different language module. This way, arbitrary decorations can be overlaid over exiting syntax. The example in Fig. 12 renders an arrow above all references that have pointer type. Another example could be to change the background color of some nodes based on external data such as profiling times. By including a tooltip in the definition of the decoration, users can get more detailed information by hovering over the decorated part of the program. Another use case for conditional editors is the expression debugger shown in Fig. 11.

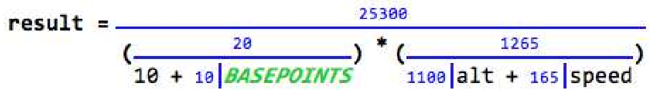


Fig. 11. This expression debugger renders the values of all subexpression over or to the left of the expression itself. The original expression (without the debug info) is `(10 + BASEPOINTS) * (alt + speed)`.

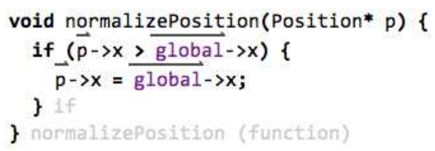
Annotations Annotations are similar to conditional editors in that they can render additional syntax around (or next to) existing syntax without the original syntax definition being aware of this. However, in contrast to conditional editors, annotations are additional nodes (i.e., they are additional data in the program) and not just a property of the projection. The additional nodes are stored as children of the annotated node. Fig. 14 shows an example in which requirements traces are added to C code (details are discussed in [9]).

Read-Only Contents Especially in DSLs for non-programmers it is often useful to be able to project rigid, predefined, non-deletable skeletons of the to-be-written program in order to guide the user. For example, in Fig. 9, the keywords `atomicblock`, `realizes`, `contract` and `ccode`, as well as the brackets and lines, are automatically projected as soon as a user instantiates an atomic block. Similarly in Fig. 8, the grey line starting with "last updated" is automatically projected and consists of computed data. In MPS, parts of the syntax of a program can be marked as `readonly`, meaning that they cannot be deleted or changed. This does not just work for constants (keywords), but for arbitraty content (such as the inherited ports of blocks shown in Fig. 9).

5 Discussion and Summary

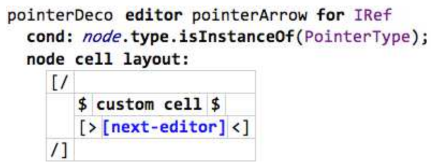
In this paper we have discussed the syntactic flexibility supported by MPS' projectional editor. We have described the various supported notational styles and emphasized that they can be combined flexibly. However, it is not enough to just compose different *notations* – other aspects of languages must be composed as well. Language composition with MPS is discussed in [2].

Over the last three years a team at itemis has been developing mbeddr [1], using MPS in a non-trivial development project. Many of the notations discussed



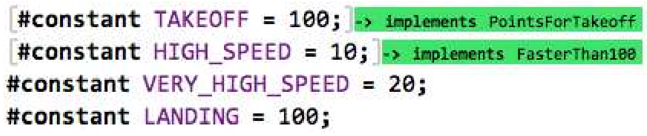
```
void normalizePosition(Position* p) {
  if (p->x > global->x) {
    p->x = global->x;
  } if
} normalizePosition (function)
```

Fig. 12. C references whose type is a pointer are annotated with the arrow on top. This works for all kinds of references, including to arguments, local variables and global variables.



```
pointerDeco editor pointerArrow for IRef
  cond: node.type.isInstanceOf(PointerType);
  node cell layout:
    [/
       $ custom cell $
       [> [next-editor] <]
    /]
```

Fig. 13. The editor applies to concepts that implemenet `IRef` and whose type is pointer. The editor renders the arrow (manually drawn in the `custom cell`) on top of the existing editor.



```
#constant TAKEOFF = 100; -> implements PointsForTakeoff
#constant HIGH_SPEED = 10; -> implements FasterThan100
#constant VERY_HIGH_SPEED = 20;
#constant LANDING = 100;
```

Fig. 14. The first two constants have traces attached. These are pointers to requirements shown in the code. The original definition of C is not aware of these annotations.

in this paper are used in mbeddr and its commercial addons. Several of the extensions have also been developed in the context of mbeddr. In addition, we are now also developing business applications (in the insurance and financial domains) with MPS. There, non-textual notations (and in particular, math and tables) are essential to be able to allow non-programmers to directly contribute to the programming effort. User feedback is very positive: they said that the abililty to have such notations is a signigicant advance over existing or alternative tools and approaches.

Based on this experience we conclude that the notations supported by MPS are reasonably complete relative to the notational styles encountered in practice. Classical textual notations are found in programming languages and DSLs; graphical notations are used by many modeling tools; mathematics are widespread in scientific or financial domains; tables are ubiquitous, as the the popularity of Excel demonstrates. And prose (with interspersed program elements) is an important ingredient to almost all these domains (for documentation, requirements or comments).

Acknowledgements Thanks to Bernd Kolb and Niko Stotz for feedback on this paper, and to Niko Stotz for building the margin cell notation.

References

1. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: instantiating a language workbench in the embedded software domain. ASE Journal **20**(3) (2013) 1–52
2. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: GTTSE 2011. LNCS. Springer (2011)
3. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Proceedings of SLE'14. (2014) 20
4. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
5. Lisson, S.: MPS Math Plugin. `https://github.com/slisson/mps-math/`
6. Lisson, S.: MPS Tables Plugin. `https://github.com/slisson/mps-tables/`
7. Lisson, S.: MPS Richtext Plugin. `https://github.com/slisson/mps-richtext/`
8. Fowler, M.: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010)
9. Voelter, M., Ratiu, D., Tomassetti, F.: Requirements as first-class citizens. In: Proceedings of ACES-MB Workshop. (2013)

Putting the Pieces Together – Technical, Organisational and Social Aspects of Language Integration for Complex Systems

Håkan Burden

Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
`burden@cse.gu.se`

Abstract. Dealing with heterogenuous systems is often described as a technical challenge in scientific publications. We analysed data from 25 interviews from a study of Model-Driven Engineering at three companies and found that while the technical aspects are important, they do not encompass the full challenge – organizational and social factors also play an important role in managing heterogenuous systems. This is true not only for the development phase but also for enabling early validation of interdependent systems, where processes and attitudes have an impact on the outcome of the integration.

Keywords: Empirical and Exploratory Case Study, Model-Driven Engineering

1 Introduction

Complex systems, consisting of numerous and interdependent subsystems [15], require a plethora of languages for efficient implementation [7]. From the aspect of Model-Driven Engineering (MDE), the challenges are often described in technical terms [4] since heterogenuous languages imply different abstraction levels, representations and aspects of software [8], but also since the languages have their own domain-specific and platform-dependent constraints [11]. The one-sided focus on technical aspects is surprising since Kent already in 2002 pointed out that if MDE is to be successful it needs to encompass also the organisational and social aspects of software engineering [10], a claim that has since been reiterated [1, 9].

To explore to what extent language integration for comlex systems is a challenge in terms of technical, organisational and social aspects we analysed data collected at three different companies, looking for evidence regarding the motivations and challenges of heterogenuous development of embedded systems. Among the findings are that engineers tend to favour integration at the concrete code level instead of at the more abstract model level, that management needs to fit the right team with the right task and that which language you use identifies you as a software engineer.