Delft University of Technology

Lessons learned from developing mbeddr
a case study in language engineering with MPS

Völter, Markus; Kolb, Bernd; Szabó, Tamás; Ratiu, Daniel; van Deursen, Arie

DOI
10.1007/s10270-016-0575-4

Publication date
2019

Document Version
Accepted author manuscript

Published in
Software and Systems Modeling

Citation (APA)
Völter, M., Kolb, B., Szabó, T., Ratiu, D., & van Deursen, A. (2019). Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software and Systems Modeling*, *18*(1), 585-630. https://doi.org/10.1007/s10270-016-0575-4

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Lessons Learned from Developing mbeddr: A Case Study in Language Engineering with MPS

Markus Voelter, Bernd Kolb, Tamás Szabó,
Daniel Ratiu and Arie van Deursen

Report TUD-SERG-2016-025

TUDelft

SERG

TUD-SERG-2016-025

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in *Software & Systems Modeling*, 2017. DOI https://doi.org/10.1007/s10270-016-0575-4

© 2016 Springer. The final publication is available at http://link.springer.com/

Lessons Learned from Developing mbeddr

A Case Study in Language Engineering with MPS

Markus Voelter
independent/itemis, Germany
voelter@acm.org

Bernd Kolb
itemis AG, Germany
kolb@itemis.de

Tamás Szabó
itemis AG, Germany and
Delft University of Technology, The Netherlands
tamas.szabo@itemis.de

Daniel Ratiu
Siemens AG, Germany
daniel.ratiu@siemens.com

Arie van Deursen
Delft University of Technology, The Netherlands
Arie.vanDeursen@tudelft.nl

Abstract

Language workbenches are touted as a promising technology to engineer languages for use in a wide range of domains, from programming to science to business. However, not many real-world case studies exist that evaluate the suitability of language workbench technology for this task. This paper contains such a case study.

In particular, we evaluate the development of mbeddr, a collection of integrated languages and language extensions built with the Jetbrains MPS language workbench. mbeddr consists of 81 languages, with their IDE support, 34 of them C extensions. The mbeddr languages use a wide variety of notations – textual, tabular, symbolic and graphical – and the C extensions are modular; new extensions can be added without changing the existing implementation of C. mbeddr's development has spanned 10 person years so far, and the tool is used in practice and continues to be developed. This makes mbeddr a meaningful case study of non-trivial size and complexity.

The evaluation is centered around five research questions: language modularity, notational freedom and projectional editing, mechanisms for managing complexity, performance and scalability issues and the consequences for the development process.

We draw generally positive conclusions; language engineering with MPS is ready for real-world use. However, we also identify a number of areas for improvement in the state of the art in language engineering in general, and in MPS in particular.

Categories and Subject Descriptors D.3.2 [*Extensible languages*]; D.3.4 [*Code Generation*]; D.2.3 [*Program Editors*]

General Terms Languages, Experimentation

Keywords Language Engineering, Language Extension, Language Workbenches, Domain-Specific Language, Case Study

1. Introduction

The Importance of Languages Languages are the backbone of computer science and software engineering: they are used for programming, configuration, testing, architecture modeling or requirements specification. Languages come with various syntactic styles including textual (most programming languages), tabular (spreadsheets), symbolic (mathematical) and graphical (architecture modeling languages). The productivity of a language is amplified by integrated development environments (IDEs) that are aware of the language structure, syntax and semantics. They support the language user by rendering the notation, providing code completion (or its graphical equivalent, the palette), highlighting errors and debugging the program as it executes. Suitably designed languages are enablers for advanced tooling such as formal verification.

As software is established as the backbone of more and more domains, specialized domain-specific languages (DSLs) are needed to efficiently write this software. Traditional domains for DSLs include language and compiler

implementation [37, 46], embedded software [13, 53, 55, 56, 107] or web applications [94]. More recently, DSLs have been used in domains that are not traditionally associated with formal, executable or analyzable languages. Examples include home automation [43], computational biology [76] and business applications [106] in the insurance industry. Because the users in such domains are not trained as programmers, they have different expectations of how languages and IDE should work; the diversity of linguistic styles and their syntax grows. The economies of DSLs – smaller user groups and more rapid evolution – requires the effort for implementing DSLs to be reduced in comparison to general-purpose programming languages. All of these trends lead to a growing need for addressing language and IDE development more systematically.

Language Engineering Language engineering [102] is the term applied to this field, and provides a holistic view of all aspects of designing, implementing and using languages and IDEs. It encompasses grammars, parsing, type systems, semantics, compilers, refactorings and program analyses and targets programming languages, DSLs, modeling languages and specification languages, covering all the various notational styles mentioned above. Language engineering emphasizes language extension and composition to manage the complexity of ecosystems of languages [31, 97].

Language Workbenches Language workbenches (LWBs) are tools to efficiently support language engineering. While the term was introduced by Martin Fowler in 2004 [32, 34] the field dates back to the 1980s with tools such as the Synthesizer Generator [72] and the Meta Environment [47]. The latter is an editor for languages defined via SDF, a general parsing framework. Rascal [49] and Spoofax [45] provide Eclipse-based IDE support for SDF-based languages and, together with Xtext[1], MetaEdit+ [83] and MPS [4], are contemporary language workbenches.

Jetbrains MPS Jetbrains MPS is particularly interesting because it aims at supporting seamless language composition and mixed notations. It is the target of evaluation in this paper and is described in more detail in Section 2.1.

1.1 Contribution

MPS is a state-of-the-art language workbench that promises to be suitable for developing industrial-grade ecosystems of related and integrated languages, some of them modular extensions of others, mixing textual, graphical, tabular and mathematical notations. In this paper we validate the degree to which this promise holds by critically reviewing the development of mbeddr [103], a set of languages for embedded software engineering built with MPS. Both MPS and mbeddr are open source software, making them especially interesting as a case study.

[1] `http://eclipse.org/Xtext`

Existing publications cover the development of relatively simple DSLs with one or more of the above-mentioned language workbenches (see related work in Section 9), but, to the best of our knowledge, none evaluates a language workbench with a case study at the scale of mbeddr. However, such case studies are critically needed in order to test whether language workbenches are suitable for developing the backbones of the domains mentioned above. The 10 person years of effort invested into development of mbeddr (more details on the effort are discussed in Section 6.2) puts us in a unique position to perform such an evaluation.

1.2 MPS vs. Other Language Workbenches

A case study of this size and scope can only be performed once, based on *one* language workbench. The results are thus specific to this particular language workbench. We chose MPS because it is the only industry-strength language workbench that uses a projectional editor, it is freely available and promises to be ready for real-world use. This makes it an interesting target for evaluation. MPS can justifiably be considered as the most complete language workbench today; evaluating it may lead to increased research in this class of tools. Such research can address the limitations we identify, while at the same time keeping the positive aspects of MPS we point out in this paper.

1.3 Relationship to Earlier Publications

We have published on mbeddr and MPS before. An early paper [96] at MoDELS 2010 was based on a predecessor of mbeddr called the Modular Embedded Language (MEL) and introduced the idea of using language engineering to build embedded software development tools. mbeddr itself was introduced at SPLASH/Wavefront 2012 [101] based on the then-current state of the mbeddr implementation. It contains a thorough and systematic treatment of the challenges in embedded software development and how language engineering can help to solve them. It describes the extensions available in mbeddr in some detail, and discusses how they are implemented. At OOPSLA 2015 we published a case study paper that evaluates the use of mbeddr from the perspective of embedded software development [107].

A paper published in 2013 in the Journal of Automated Software Engineering [103] adopted the perspective of language engineering with MPS, and also evaluates the development of mbeddr to some degree (in Section 5). However, the current paper treats mbeddr much more systematically as a case study: Sections 4, 5 and 6 either do not exist in [103] or are much more detailed in the current paper. We also ask additional research questions in the current paper, and we discuss them in substantially more detail, partly also as a consequence of three more years of mbeddr development (2016 vs. 2013).

We have also published on specific topics of MPS-based language engineering such as notational flexibility [100], language composition [97], usability of projectional edi-

tors [105], language testing [67] and combining language engineering and formal verification [59, 68, 69]. The present papers adds to this by evaluating these aspects collectively, by studying the development of one complex set of languages. A few paragraphs and figures are adopted from these earlier publications.

1.4 Structure

We organize the paper according to the structure for case studies proposed by Runeson et al. [74] and Yin [111]. We begin by outlining the background on MPS, projectional editing and mbeddr in Section 2. A brief tutorial on language development with mbeddr is contained in Section 3. In Section 4 we discuss the setup of the case study by introducing the research questions and the collected data. Section 5 then describes the relevant context of the case study (cf. Dyba et al. [27]), including the team, the development tools and the development process. The implementation of mbeddr in terms of size, effort, development timeline and major structures is discussed in Section 6. The core of the paper is Section 7 where we answer the research questions in detail. The discussion in Section 8 looks at lessons learned from the development of mbeddr that do not fit the research questions and addresses threats to validity. Related work is covered in Section 9 and we conclude the paper with Section 10.

1.5 Audience

This paper is primarily targeted towards language engineering researchers who want to understand the state-of-the-art in language engineering and identify areas for future research. Secondarily, the paper addresses developers of language workbenches who want to understand what contemporary language workbenches, and in particular, MPS, are capable of, and what is still missing. Third, the paper helps practitioners understand the degree to which language engineering (and in particular, MPS) can be used to build large-scale, real-world DSLs.

2. Background

This section provides a brief background on MPS (Section 2.1), projectional editing (2.2) and mbeddr (2.3). More details on language development with MPS (i.e., the subject of evaluation in this paper) and mbeddr (i.e., the case study used for the evaluation) are provided in Section 3.

2.1 Jetbrains MPS

MPS[2] is an open source language workbench developed by Jetbrains since the early 2000s[3]. It has comprehensive support for specifying structure, syntax, type systems, transformations and generators, debuggers and IDE support (see Figure 1). According to the comparison in [32] it is one of the most fully-featured language workbenches. It is also used in practice for developing languages in domains such as computational biology [76], web applications [1], requirements engineering [104], insurance DSLs [106], safety engineering [71] as well as embedded software [103]. MPS has an active user community and continues to be developed by Jetbrains and other contributors. One of MPS' distinguishing features is that it uses a projectional editor; we explain this technology in the next subsection.

2.2 Projectional Editing

Projectional editing is one style of implementing the core of a language workbench. It avoids parsing the concrete syntax of a language to construct the abstract syntax tree[4] (AST); instead, editing gestures *directly* change the AST, and the concrete syntax is rendered ("projected") from the changing AST.[5] This means that, in addition to text, languages can also use non-parsable notations such as mathematical symbols, tables and diagrams [100]. Since projectional editors never encounter grammar ambiguities[6], they support a wide range of language composition [97] techniques, such as those defined in [31].

Projectional editing (sometimes also called structured editing or syntax-directed editing) is not new, and tools such as the Incremental Programming Environment [58], GANDALF [62], and the Synthesizer Generator [72]) were developed in the 1980s. Work on projectional editors continues today; Intentional Programming [17, 23, 77, 78] is its best known incarnation. Other contemporary tools include Más [2], the Whole Platform [6], and MPS, which is the subject of this paper.

Projectional editors have two main advantages, both resulting from the absence of parsing. First, they support notations that cannot easily be parsed, such as tables, diagrams or mathematical formulas—each of which can be mixed among each other and with textual notations [78, 100]. Second, they support various methods of language composition, including modular language extension as well as embedding of unrelated languages into a host language [77, 97].

Traditionally, projectional editors were tedious to use and were hardly adopted in practice. This was mainly because of problems in editor usability and editing efficiency, nicely illustrated by the following quote, taken from a paper [66] that describes the development of a DSL with the Synthesizer Generator mentioned earlier: *"Program editing will be considerably slower than normal keyboard entry, although actual time spent programming non-trivial programs should be reduced due to reduced error rates."* With MPS, in con-

[2] `http://jetbrains.com/mps`

[3] This, as well as other statements about Jetbrains' activities are based on our regular communication with the MPS team at Jetbrains (see Section 5.2) and used with their permission.

[4] Technically, MPS' abstract syntax is a graph because it contains cross-references in addition to the containment structures of a tree. We call it AST nonetheless.

[5] Watch this video `https://www.youtube.com/watch?v=iN2PflvXUqQ` to gain a better understanding of projectional editing.

[6] Essentially, they make the user decide in situations where a parser would encounter an ambiguity.

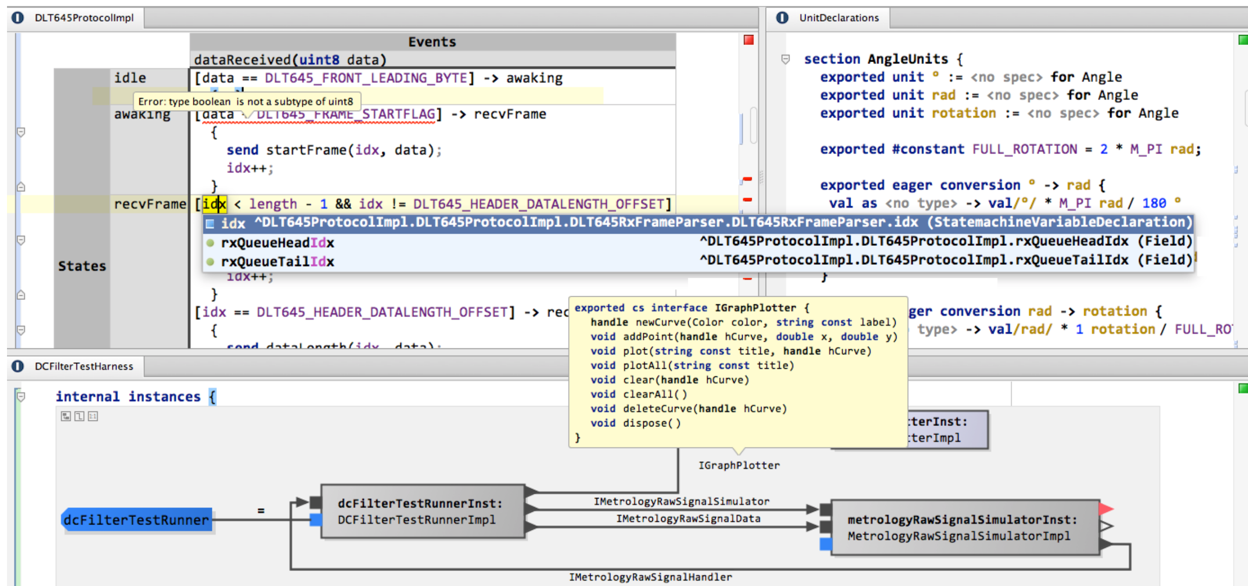

Figure 1. The screenshot shows various parts of the mbeddr languages: a part of a protocol parser state machine (top left), unit declarations (top right) and component wiring for a test case (bottom). It also illustrates how mbeddr provides IDE support for C and its extensions, including syntax highlighting, code completion, error markup, refactorings, quick fixes and tooltips. The screenshot also showcases the support for mixed notations (text, tables, diagrams).

trast, editing textual syntax can be made quite close to "normal text editing". It also supports diff-merge on the level of the projected concrete syntax. The study in [105] shows that users are willing and able to work with the editor after getting used to it. However, even in MPS, the editor is not identical to a text editor; this can be a hurdle for new MPS users who are used to text editors, and thus a barrier to adoption[7]. The trade-offs implied by projectional editing are one aspect of the evaluation in this paper.

2.3 mbeddr

mbeddr applies MPS and projectional editing to embedded software engineering: it provides an extensible version of C plus a set of predefined extensions such as physical units, interfaces and components, state machines and unit testing. Since extensions are embedded in C programs, users can mix higher-level abstractions with low-level C code. Some extensions use tabular, mathematical or graphical notations, usually mixed with text. Figure 1 shows some examples. Developers are not forced to use the extensions; they may use them only when they consider them appropriate. mbeddr also supports product line variability, requirements traces and documentation as well as formal verification [59, 70]. For reasons of space, we do not introduce in detail any of the languages provided by mbeddr; we encourage the interested reader to take a look at [103]. mbeddr is open-source under the Eclipse Public License and is available from `http://mbeddr.com`. Figure 2 shows an overview of mbeddr's ingredients.

mbeddr continues to be actively developed as open source software by a team at itemis in Stuttgart, Germany. In addition, Siemens PLM Software has released the commercial Embedded Software Designer (ESD)[8] that is essentially a set of mbeddr extensions. mbeddr has been (and continues to be) used successfully in a variety of systems with several different users. The case study in [107] reports on an industrial case study on developing the embedded software for a smart meter using mbeddr. It finds that the extensions help significantly with managing the complexity of the developed software. They improve testability mainly by supporting hardware-independent testing, as illustrated by low integration efforts, and do not incur significant overhead regarding memory consumption and performance.

The case study in [107] demonstrates that mbeddr fulfils its purpose from the perspective of an end user, and illustrates that language engineering can lead to useful results which would be otherwise much more expensive to achieve. In this paper we switch the perspective to language engineering, aiming at evaluating the development of mbeddr itself.

2.4 The mbeddr Platform

In addition to mbeddr itself (Figure 2) we also developed a set of MPS utilities called the mbeddr.platform. It consists

[7] We have used the MPS projectional editor also with users from business domains who do not have years of exposure to textual IDEs; for them, a projectional editor is much easier to get used to.

[8] `https://www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/embedded-software-designer.shtml`

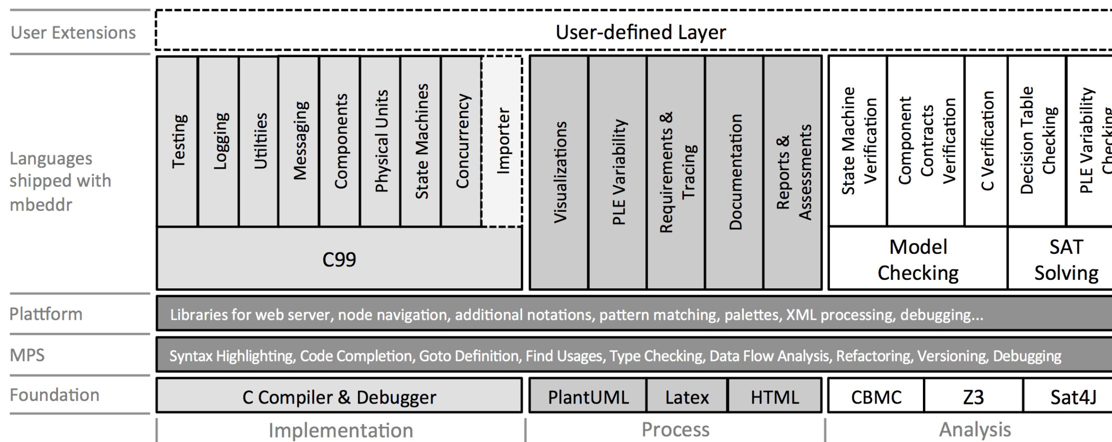

Figure 2. mbeddr addresses software implementation, supports aspects of the development process, and also integrates formal analysis and verification techniques. At the core, mbeddr is a version of C99 that can be extended incrementally. It ships with a wide range of extensions for embedded software development. It relies on MPS as the language workbench, plus various enhancements and extensions collected in the platform. mbeddr relies on exiting (command line) tools as a foundation.

of MPS extensions useful for building languages, ways to customize the MPS application UI (outside the languages and editors themselves) as well as support for mathematical notations, graphical editors or unstructured prose. The platform is used in mbeddr, but also in a variety of other commercial projects developed by the team at itemis. As discussed in Section 6.1, the size and effort spent on the platform is comparable to that of mbeddr; a lot of MPS expertise has accumulated in the platform and is now used to further simplify language and IDE development for language engineers. The platform is open source software and available from `http://mbeddr.com/platform.html`.

The case study in this paper refers to mbeddr itself (the C extensions and associated languages for embedded software development), and not the platform. We have made this decision to limit the scope of this paper to a manageable size. We also feel that the actual language engineering case study is mbeddr itself; the utilities we created in the platform along the way are a side-effect of how one manages complexity with MPS. We discuss this aspect in Section 7.3.

2.5 Language Engineering and Formal Verification

As mentioned earlier, we integrated several formal verification tools into mbeddr, exploring the synergies between language engineering and formal verification. In the present paper we discuss these aspects only to a very limited degree to limit the scope of this paper. We refer the reader to [59, 68, 69] for details.

3. Language Development with MPS

This section describes language development with MPS as a foundation for the evaluation of MPS. As a running example, this section illustrates the development of a simple extension of C for concurrent programming. Readers who are familiar with MPS may skip this section. This section is not a full tutorial for which we refer the reader to [102], [15] and [5]. Also, some of the more advanced aspects of MPS language development are introduced later in the context of their respective evaluation.

Example The running example we use in this section shows the implementation of one of the language extensions from mbeddr's concurrency support: the `dequeue` statement. Its purpose it to take an element from a non-blocking queue in a way that is safe (i.e., the queue is locked if necessary) and convenient (i.e., the user does not have to take care of the low level locking details). Here is an example use:

```
shared int64Q data;

cyclic task sumUp {
  dequeue if available from data -> val {
    sum += val;
  }
}
```

We first declare a shared global variable `data` of type `int64Q` (the type `int64Q` is declared elsewhere). Then we implement a task that is expected to be scheduled cyclically (the schedule itself is not shown). `cyclic` tasks are prohibited from blocking and are expected (and possibly monitored) to have an execution time for each activation lower than a particular cycle time. Inside the task we use the `dequeue` statement to take one element from the `data` queue if one is available; that element is available as `val` inside the body of `dequeue`. The body is only executed if an element is available in the queue. This syntax has been chosen to structurally enforce the following characteristics:

- Users cannot forget to lock the queue or test for availability of an element.

- The lock, if acquired, is released immediately after the value has been taken from the queue and placed into `val` (short lock times are essential for good performance of concurrent systems because it reduces contention).
- Users can only access the dequeued element through `val` if one is available; otherwise the whole body is skipped.

These characteristics are enforced by the lower-level code generated from the `dequeue` statement above:

```
uint64 ___val = 0;
boolean ___taken = false;
atomic <data/readWrite> {
  if (!data.isEmpty) {
    ___val = data.take;
    ___taken = true;
  }
}
if (___taken) {
  sum += ___val;
}
```

This code uses other extensions available in mbeddr, such as `atomic` for locking a shared variable or `data.take` for taking an element from a queue. These are then reduced further by downstream generators. Using this kind of multi-step reductions is idiomatic in MPS. Note how the lock is only held while we check the queue for availability and take an element from the queue to achieve short lock-hold times.

The extension as implemented in MPS supports code completion for all syntactic elements, respects scoping rules (for example, when resolving the reference to the queue), restricts the visibility of `val` to the body of `dequeue` and performs type checking (for example, the `val` variable has the same type as the queue elements).

Terminology We introduce some of the most important terminology used in MPS. A *program* denotes code written by a developer. It is represented as the AST and projected in whatever notation is defined for the language in which the program is written. A program consists of a tree of *nodes* (the AST) with resolved cross-references between nodes (so it is effectively a graph). A *root* node is a node that has no parent; it is edited in its own editor tab in the MPS IDE and intuitively corresponds to a file in a classical IDE. The *model* is the granularity of physical storage in MPS. It is an XML file and contains a number of root nodes, each with its own tree/graph beneath it. Models are owned by *modules*, and a *project* is a collection of modules.[9] Modules come in three kinds: *Languages* are modules that contain language implementations. *Solution* are modules that contain end-user programs, as well as support libraries for languages. Thus, language modules are the meta level relative to solution modules. *Devkits* are groups of languages to simplify a solution's import of related groups of languages. Finally, the *BaseLanguage* is MPS' (slightly extended) version of Java. It can be used for Java programming (in solutions) and also plays a role in language implementation. We now discuss some details of the various languages used for language implementation.

Aspects, DSLs and BaseLanguage In MPS, a language definition consists of language aspects such as structure, editor or type system. Each of the aspects is implemented with an aspect-specific DSL. Some of these DSLs are declarative, others are rule-based, and yet others are imperative. However, all of them reuse MPS' BaseLanguage to some extent, typically by embedding BaseLanguage expressions or statements. We discuss each language aspect below, each in a separate subsection. We start with a subsection on language modularity and dependencies in general.

3.1 Created and Used Languages

The concurrency support developed in this running example is intended to be a modular C extension, not requiring changes to the implementation of the core C language. Thus we start out by creating a new language `com.mbeddr.ext.concurrency`. Because we will reuse parts of the definition of C, we make this new language extend `com.mbeddr.core.statements` and `com.mbeddr.core.expressions`; these are part of the modularized implementation of mbeddr C. Creating new languages and defining their dependencies is handled via menu items and property dialogs in MPS; we do not show these in this paper.

The mbeddr concurrency support is intended to provide language extensions for safe and convenient concurrent programming for diverse platforms. This means that the language that defines the extensions should *not* prescribe the way the language is translated to C, because this will be different for the different platforms. So each of the possibly multiple transformations is implemented in its own language.[10] In this example we look at the transformation to an implementation based on Pthreads [61], so the language that contains the transformations is called `com.mbeddr.ext.concurrency.pthreads`. It has a dependency on `com.mbeddr.ext.concurrency` so it can see the concepts for which it provides the translations to C.

3.2 Structure

A language contains a number of language concepts (known as meta class or AST type in other tools). Each of the language aspects mentioned before contributes to each concept's definition. In this sense, a language definition in MPS is a 2-dimensional matrix of concepts and aspects.

The definition of a language concept starts with its structure, because all other aspects refer to the structure of concepts in one form or another. For our example we need two concepts: the `dequeue` statement itself as well as the `val` expression used inside its body.

[9] For Eclipse users: the project corresponds to the Eclipse Workspace, the modules correspond to projects and models correspond to files or packages.

[10] The set of asepcts used to define a language includes transformations. A language that contains *only* transformations for concepts defined in another language is still called a language in MPS

The `val` expression is a keyword expression, i.e., it is an expression with a language-defined structure and syntax. It has no further substructure under it (in terms of the AST). Here is the structure definition:[11]

```
concept ValExpr extends Expression
  alias val
```

MPS uses an object-oriented style subtyping to implement structural compatibility: if the `ValExpr` is to be legal in places where C expects an `Expression`, then `ValExpr` has to extend mbeddr C's `Expression` concept. It is visible here because our new language extends `com.mbeddr.core.expressions`, which contains the `Expression` concept.

The alias is the string a user has to type (or select from the code completion menu) to enter an instance of `ValExpr` when editing a program. It is good practice, though not technically required, to make the alias the same as the leading keyword of the concrete syntax of the concept.

The `DequeueStatement` extends C's `Statement`, and defines an alias `dequeue`:

```
concept DequeueStatement extends Statement
                         implements IAtomic
  alias: dequeue
  children:
    queue : GlobalVarRef  [1]
    body  : StatementList [1]
```

It also defines two children, both reusing existing concepts from mbeddr C, and both using a cardinality of `1` (mandatory single child; `0..1`, `0..n` and `1..n` are also available). The first one represents the queue from which we intend to take an element. Instead of *referencing* the queue directly, the statement *owns* a `GlobalVarRef` as a child; it in turn references the queue (we will discuss the necessary type checks in Section 3.5). The second child is a `StatementList` that contains the code that will be executed inside the `DequeueStatement`. Finally, the `DequeueStatement` also implements the `IAtomic` interface. This is a marker interface that expresses that the `DequeueStatement` acts similarly to an `atomic` statement in that it provides a lock for particular global variables; we explain the details on this below.

3.3 Editors

In MPS, editors play the role of the concrete syntax, or notation: they define how an instance of a concept is visually represented. They also define actions that customize how the user interacts with the instance of the concept when editing a program. We discuss both below.

Notation Each language concept has its own editor.[12] An editor consists of a collection of editor *cells*. There are many different kinds of cells available for use by the language developer: examples include constant cells, cells that contain child nodes, collection cells that act as containers for more cells as well as cells that render arbitrary strings. The editor for the `ValExpr` is the simplest one possible: it contains a single constant cell with the text "val".

```
<default> editor for concept DequeueStatement
  node cell layout:
    [- dequeue from % queue % -> val % body % -]
```

Figure 3. The editor definition for the `DequeueStatement`.

Figure 3 shows the editor definition of the `DequeueStatement`. It contains a horizontal collection (`[- .. -]`), four constant cells (`dequeue`, `from`, `->` and `val`) as well as two child cells[13] for the two child nodes defined in the structure (`%queue%` and `%body%`). When an instance of a `DequeueStatement` is projected in the editor, the two child cells contain the visual representation defined by the editor definitions for the respective children. At the point of embedding, the language designer does not have to care about the concrete syntax of the embedded child; they only declare the syntactic nesting using the syntax shown above.

Actions In addition to the definition of the visual representation of concepts in the editor, the editor aspect[14] also defines how users interact with the notation. Examples of such interactions include:

- Deletion: what happens when the user presses `Backspace` on a given cell.
- Side transformations: how can tree structures be entered linearly (entering `2+3` by typing `2`, then `+` and then `3`).
- Substitutions: allow a local variable declaration (such as `int32 x;`) to be created by entering the type `int32`.

No such editor behavior definitions are necessary for the `DequeueStatement` or the `ValExpr`; the defaults provided by MPS suffice. However, to illustrate the mechanism, let us imagine that the `DequeueStatement` can be configured to block; the syntax would be

```
dequeue blocking if available ...
```

To enable the user to "just type" the `blocking` keyword on the right side of the `dequeue` keyword, the language developer has to define a right transformation on the `TakeStatement` keyword. The following code is a slightly simplified version of this transformation:

```
right transformed node: TakeStatement {
  matching text: "blocking"
  transform: (model, sourceNode, pattern)->node<> {
```

[11] As long as MPS uses textual notations for language definition, we show the example code as text; when non-textual notations are used, such as in editor definitions, we use screenshots.

[12] A concept can have several editors; they can be switched for each program.

[13] For historic reasons, MPS uses %child% to enclose child cells in editors.

[14] Strictly speaking, some of this behavior is defined in the Actions aspect. However, since the Editor and Actions aspects will be merged in the upcoming version of MPS, we already treat them as one in this tutorial.

```
    sourceNode.blocking = true;
    sourceNode;
  }
}
```

Many such actions, as well as others, are necessary to build usable editors for realistic languages. It is a significant effort to implement those completely and consistently, as we discuss in the Editor Usability paragraph of Section 7.2. Based on this experience we have automated the generation of such actions from semantically more meaningful editor cells based on an MPS extension called Grammar Cells [108].

3.4 Constraints

As a first approximation, the validity of a program is determined by the structure: only nodes of a compatible concept (in terms of subtyping through the `extends` relationship between concepts) can be instantiated in any given program location. However, validity is further determined by typing rules (see next subsection) and constraints. A constraint is a Boolean expression that determines whether a structurally compatible concept can actually be instantiated in a given location, thereby further restricting the tree structure beyond pure structural compatibility.

Tree Constraints The `ValExpr` extends `Expression`, so, structurally, it can be used wherever an `Expression` is expected. However, from a semantic perspective, it is only valid inside the body of a `dequeue` statement. To enforce this, we define a `can be child` constraint for the `ValExpr`:

```
can be child
  (childConcept, node, parentNode)->boolean {
    parentNode.ancestor<DequeueStatement> != null;
  }
```

Note that constraints *prevent* the user from entering invalid code[15]. This means that they are executed *before* the node, in this case the `ValExpr`, has been created. This is why the constraint is expressed in terms of the `parentNode`: we check if the ancestors (the sequence of parent nodes, i.e., the containment hierarchy) contains an instance of `DequeueStatement`. Only if we find one are we allowed to instantiate the `ValExpr`.

Scopes Constraints are also used to express visibility rules, known as *scopes* in MPS. A scope is a constraint that determines which target nodes are visible to a *reference* (as opposed to the containment constraints discussed before). These targets are then made available to the user through the code completion menu, so they can be selected or just typed in (scopes also lead to error markers in the case where a (now) invalid reference was entered when the scope was (erroneously) still more permissive). Our language extension does not contain any references, but we can look at the `GlobalVarRef` used to refer to the queue. It has a reference member called `var`:

[15] Remember that a projectional editor only lets the user enter programs that are structurally valid.

```
concept GlobalVarRef extends Expression
  references:
    var : GlobalVarDecl[1]
```

This reference can refer to *any* global variable declaration; it is the responsibility of the scope to determine the set of valid targets. Since the reference is typed to be a `GlobalVarDecl`, the scope implementation must return a sequence of global variable declaration nodes (an `nsequence<GlobalVarDecl>`). The implementation of the scope shown below starts from the enclosing node, navigates up the tree to find the `Module` in which the current node resides, gets its contents and filters for `GlobalVarDecl`s. Since it is structurally ensured that all C code is written in `Module`s, we can assume that one of the ancestors is actually a `Module`.

```
link var scope:
  (enclosingNode, pos) -> nsequence<GlobalVarDecl> {
    enclosingNode.ancestor<Module>.contents.
         ofConcept<GlobalVarDecl>;
  }
```

Note that this implementation is slightly simplified compared to mbeddr's actual implementation of this scope, because mbeddr uses a set of library functions to take import relationships between modules into account.

3.5 Type System

The type system aspect encodes the static semantics of a language; it ensures the consistency of the types in the program and also checks other arbitrary correctness rules beyond structure.[16] We start with the latter, because it is different in an interesting way from the constraints just discussed.

Checking Rules As discussed above, constraints use Boolean expressions to determine whether a node is valid in a given program location. If it is not valid, they *prevent the user from entering that node*. The type system's checking rules similarly use Boolean expressions to determine validity. However, they are evaluated *after the node has already been entered*. Instead of preventing invalid use, they flag invalid use with a red squiggly line and an error message after the fact. The following code shows a checking rule for the `queue` global variable reference of the `DequeueStatement`:

```
checking rule for DequeueStatement {
  ensure node.queue.var.type.isInstanceOf(QueueType)
    else error "global variable not a queue"
         on dqs.queue;
  ensure node.queue.var.@shared != null
    else error "queue must be shared"
         on dqs.queue;
}
```

The first of the two `ensure` statements verifies that the type of the variable referenced by the `queue` child of the current

[16] In traditional, parser-based language definitions, the type system is typically considered to also include the name-based resolution of references. However, in MPS, references are not encoded by name resolution rules, but by actual references to the unique ID of the target node that are established upon *entering* the reference (by code completion or plain typing).

node (the `DequeueStatement`) is a `QueueType`. If not, we report an error message on the global variable reference. The second `ensure` checks that the global variable referred to by `queue` has the `shared` annotation, since only shared global variables can be accessed safely in a concurrent context. Annotations are discussed in the Annotations paragraph in Section 7.1; for now you can consider them to be a optional flag on the variable. In addition to `ensure` statements, checking rules can also use regular `if` statements to check more complex conditions, and report errors using the `error` statement.

Typing Rules Our example also makes use of an actual typing rule: the `val` expression inside the body of the `DequeueStatement` must have the same type as the elements of the queue from which we take the element. MPS' type system relies on typing equations: for every concept, the developer specifies one or more typing equations. MPS then instantiates all type equations for all program nodes and uses a solver [8, 12] to infer types and detect typing errors. Below is the code that types the `val` expression:

```
typing rule typeof_QueueValExpr for ValExpr {
  node<DequeueStatement> dqs =
          node.ancestor<DequeueStatement>;
  node<QueueType> qt = dqs.queue.type : QueueType;
  typeof(node) :==: typeof(qt.queue.elementType);
}
```

We first find the `DequeueStatement` inside whose body the current `ValExpr` resides (the tree constraint shown above enforces that `val` can only be used under a `dequeue` statement). We then get the type of the `queue` global variable reference. We ensured earlier that it is a `QueueType` so we can safely downcast using the colon operator. We then declare a typing equation that expresses that the type of the current node (the `ValExpr`) must be identical to the element type of the queue. Note that the `:==:` operator is not an assignment, but it establishes type equivalence. If one of the two `typeof` expressions evaluates to an unbound type variable (based on all the other typing equations for the given program), the operator propagates the type to the unbound type variable, thus implementing type inference. If both `typeof` expressions return actual types, then the operator acts as a constraint: if the two types are not the same, an error is reported. There are additional typing operators beyond `:==:`. For example, `:<=:` takes subtypes into account.

The `QueueType`, though not shown, is just another concept defined in the `com.mbeddr.ext.concurrency` language. It has a child `elementType` that contains the type of the queue elements. It's syntax is `queue<ElementType>`. The `int64Q` type used in the initial example for the `dequeue` statement is `typedef`'d to be a `queue<int64>`.

3.6 Behavior

The behavior aspect allows the definition of methods on concepts. These act similar to Java methods and can be invoked from all other aspects (often called from constraints, the type system and generators). Methods are polymorphic, and they can also be declared on interfaces. For example, the `IAtomic` interface (implemented by the `DequeueStatement`) defines two abstract behavior methods:

```
concept behavior IAtomic {
  public abstract boolean getsReadLockFor(
          node<GlobalVarRef> r);
  public abstract boolean getsWriteLockFor(
          node<GlobalVarRef> r);
}
```

The purpose of this interface is to express that a statement acts similarly to an `atomic` statement in that it provides read or write locks for global variables. The `DequeueStatement` implements this interface and realizes the `atomic`-semantics by translating to code that uses an `atomic` statement. Since `DequeueStatement` is not abstract, it has to implement these two methods; we show one of them:

```
public boolean getsReadLockFor(node<GlobalVarRef> r) {
  return r.var == this.queue.var;
}
```

This one returns true ("dequeue statement provides a read lock for") if the global variable referenced by `r` is the same variable referenced by the global variable reference owned by the `DequeueStatement`.

The method is then invoked polymorphically from a checking rule. The rule reports an error if a global variable is accessed from outside an `IAtomic` context, and if that atomic context does not provide a lock. The somewhat simplified implementation is show here:

```
checking rule check_Lock for GlobalVarRef {
  node<IAtomic> a = node.ancestor<IAtomic>;
  if (a == null || !a.providesReadLockFor(node))
    error "global variables must be locked" -> gvr;
}
```

3.7 Generators

The name generator is a little bit misleading: in fact, generators are tree transformations that map a source AST to an output AST. Generators consist of various different kinds of transformations rules, which in turn make use of templates, i.e., fragments of target language code that determine how the source AST is transformed to the target AST.

Templates The most important kind of transformation rule is the reduction rule. Reduction rules replace a program node that is an instance of a particular concept with another node, typically an instance of another concept. The `DequeueStatement` is translated with the reduction rule shown in Figure 4. We will now discuss this rule in detail.

At the top level, a reduction rule consists of the source and the target, separated by the `-->` arrow. The source specifies the to-be-reduced concept, a flag whether subconcepts should be transformed as well and a Boolean expression that further constrains applicability.[17] The target side is what replaces each match of the left side. Notice how the target side

[17] It is also possible to use a pattern that is matched against the tree, roughly similar to tree pattern matching in Scala [63] or Stratego [95].

```
concept     DequeueStatement
inheritors false
condition  <always>

--> content node:
    @shared
    intqueue q;

    void dummy() {
      <TF {
           $COPY_SRC$ [int8] ___val = 0;
           boolean ___taken = false;
           atomic <->$[q]/readWrite> {
             if (!$COPY_SRC$ [q].isEmpty) {
               ___val = $COPY_SRC$ [q].take;
               ___taken = true;
             }
           }
           if (___taken) {
             $COPY_SRCL$ [int8 code; ]
           }
         } TF>
    }
```

Figure 4. The generator templates used for translating the `DequeueStatement`. The `dummy` function acts as scaffolding for the template fragment; the details are explained in the text.

syntactically resembles the generated code (cf. the example at the beginning of Section 3). The reason for this resemblance is that MPS uses the concrete syntax of the target node in the templates. In contrast to text-generation tools, MPS provides full IDE support for the to-be-generated target code (an example of MPS' language composition abilities).

The target code also contains generator markup, in particular the template fragment (`<TF .. TF>`) and macros (`$..$[]`). Macros are attached to nodes and represent instructions to the generator engine itself. Macros are an example of annotations, and hence can be attached to any program node, without the node's concept being aware of this; they are orthogonal to the definition of the language to whose instance nodes they are attached.

Transformations are executed in two steps. The first step copies the target code verbatim into the to-be-transformed tree (including the macros), replacing the source node. In step two, the transformation engine executes the macros, which typically change the copied tree. We discuss this in more detail below.

The template fragment separates the code that should be used to replace the source node from the scaffolding, the code outside the fragment. The scaffolding is code that is required to be able to write the target code fragment in the first place (the `dummy` function in Figure 4 is an example). For example, if the to-be-generated code contains a reference to a global variable (such as the `q` in the `atomic` statement), then there must be a global variable in the generator template; otherwise the transformation developer could not enter a global variable reference (remember that the template code must be a valid tree expressed in the target language). During transformation, only the code inside the template fragment is copied into the source tree, and references to scaffolding code have to be changed to point to a node in the source tree by a reference macro.

Let us look at some of the macros. The simplest one is the `$IF$` macro (not used in Figure 4). It contains a Boolean expression and if it evaluates to false, the node to which the macro is attached is not copied into the source tree (strictly speaking, it is copied and then later deleted again during step two, macro evaluation). The `$COPY_SRC$` replaces the node it is attached to with a node returned from an expression inside the macro. For example, the `$COPY_SRC$` around the `int8` type in the first line of the template fragment contains the following expression[18], which returns the type of the queue used in the `DequeueStatement`:

```
node.queue.var.type : QueueType.queue.elementType;
```

The effect is that the `int8` dummy node is replaced with the type used by the queue from the input node. Similarly, the `$COPY_SRCL$` (notice the `L`) copies a *list* of nodes. We use it to copy all the body statements into the generated code. The `->$` is called a reference macro. It is used to change the target of a reference. The contained expression has to return the new target (either by unique name to be resolved by MPS, or by returning an actual reference to the target previously stored in a map). In Figure 4 we use a reference macro to make the reference to `q` point to the actual global variable referenced in the source tree. The expression inside that reference macro returns the variable referenced by the `dequeue` statement's `queue` child: `node.queue.var`.

Reduction rules are executed recursively, until no more rules apply. For example, when copying the statements in the body of the `DequeueStatement`, we have to take care of the `val` expression: it is not valid in regular C code, outside the `dequeue` statement. Figure 5 shows the right side of the reduction rule for the `val` expression. It reduces the `val` expression to a reference to a local variable `___val`. This works because the reduction rule for the `DequeueStatement` creates a local variable of this name (see the first line of the template fragment in Figure 4).

Scripts MPS also supports generation scripts. These are essentially BaseLanguage programs that procedurally create the output AST. Scripts are typically used for transformations that are algorithmically complex (for example, flattening of hierarchical structures) or for generic transformations (such as removing commented code).

Priorities The rules inside a particular language's generator are automatically scheduled in a meaningful way. For

[18] The expression is not visible in Figure 4 because it is shown in the macro's inspector, an additional IDE window that shows details about the node currently selected in the main editor.

```
void dummy() {
  int8 ___val = 0;
  <TF [ ___val ] TF>;
}
```

Figure 5. The reduction rule for the `val` expression used inside a `dequeue` statement's body.

example, the rule that reduces an `atomic` statement to valid C code is executed after the rule that creates an `atomic` statement from a `DequeueStatement`. However, the generators of different languages have to be scheduled manually, if they have a dependency. To this end, MPS supports generator priorities. A priority is defined between any two generators and expresses whether a generator should be run before or after the other one. For example, a priority schedules the concurrency generator before the one that reduces mbeddr's C extensions (because they are used in some locations in the concurrency generator):

```
com.mbeddr.ext.concurrency.pthreads: main
                         << com.mbeddr.core.utils: main
```

Cyclic dependencies are not allowed and will lead to a generator error.

Text Generators Finally, MPS also supports to-text generators to generate text files as part of the transformation chain (for subsequent compilation with an existing C compiler, for example). MPS provides a DSL that essentially serializes text into a buffer, with support for indentation and dealing with list separators. Since this is not very important for the remainder of the paper we do not discuss this any further.

3.8 Intentions

Intentions are program transformations that change the program in the editor (as opposed to generators which transform programs during MPS' make process). Intentions are invoked by pressing `Alt-Enter` on a program node and then selecting a particular intention from the menu that pops up. MPS has a DSL for specifying such intentions, but the actual transformation is typically implemented procedurally using BaseLanguage (similar to generator scripts). Since intentions are not very important for the remainder of the paper we do not discuss them any further.

3.9 Refactorings

Refactorings are available from the context menu. They are also implemented procedurally. While they are important for the user, they are not particularly important for the rest of this paper, so we provide no more details.

3.10 IDE Customization

MPS supports the customization of various aspects of the user interface of the tool itself, including buttons, menu items, customized project views as well as additional windows. These customizations are crucially important for building end user friendly products based on MPS. However, they are not relevant to language engineering per se, which is why we do not discuss them in this paper.

4. Case Study Setup

In this section we describe the setup of the case study. In particular, we introduce the research questions (Section 4.1), and the data we have collected to answer these questions (Section 4.2).

4.1 Research Questions

The research questions aim at critically evaluating the degree to which it is feasible to build non-trivial ecosystems of languages with MPS. The questions will be evaluated based on the development of mbeddr; we describe the structure of the implementation of mbeddr as well as some details about the development process in the next two sections.

RQ1: Is it practically feasible to define a modular set of languages of the size of mbeddr? A cornerstone of mbeddr is the modularity of the languages; mbeddr consists of 81 different languages, 34 of them extensions to C. Our first research question evaluates the feasibility of defining such a large set of modular languages. Among other things, we investigate the modularity and extensibility of the structure, editor, type system and generator language aspects.

RQ2: What is the contribution of projectional editing to the success of mbeddr? Projectional editing is the distinguishing characteristic of MPS compared to most other industrial-strength language workbenches [32], and as far as we know, mbeddr is the largest set of languages built with a projectional editor. This question evaluates whether the tradeoffs inherent in projectional editing work in practice: its contribution to language modularity, the flexibility of using and mixing diverse notational styles, the usability of the editor and the effort of developing them.

RQ3: How effective are MPS' mechanisms for managing the complexity inherent in language development? As illustrated in the tutorial in Section 3, MPS is essentially a set of DSLs for language definition, each language describing a particular aspect of the language (such as syntax, the type system or transformations). It is also bootstrapped, i.e., implemented with itself. With this research question we elaborate on whether this approach is adequate for managing the complexity of implementing sets of integrated languages.

RQ4: What are the performance and scalability implications? mbeddr is one of the largest language engineering projects built in MPS (and maybe, generally). This raises the question of whether we encountered problems of performance and scalability as a consequence of the size of the system.

RQ5: What are the interactions with the development process? This question evaluates whether and how an es-

tablished development process (requirements, test, implementation, build, packaging) has to be adapted in the face of language development with MPS.

Another natural research question relates to the usability and productivity of using the projectional editor, because, even if RQ2 finds advantages, these are irrelevant if nobody is willing to use the editor. We have investigated this question in detail in [105]. More generally, the present paper presumes that it is actually worthwhile to build large, integrated sets of languages. Empirical support for this assumption is provided in our previous paper assessing the usefulness of mbeddr itself for the purpose of embedded software development [107].

4.2 Data Collected

Quantitative Data For the numbers regarding the size of mbeddr in Section 6.1 we have counted concepts and lines of code in the mbeddr sources. The efforts reported in Section 6.1 are careful estimates based on developer involvement, when they joined the team, and how much work they performed on mbeddr vs. on other projects. For the development progress discussed in Section 6.3 we have analyzed the history of our two source code repositories. For the performance numbers cited in Section 7.4 we have made measurements in the way outlined in that section.

Qualitative Data Most of the evaluation in this paper is qualitative in nature, reflecting on the good and the bad in the development of mbeddr. It is based on recollections of the team members, notes taken during the project, and findings discussed in [99] and [103].

5. Case Study Context

5.1 Development Team

39 developers have contributed to mbeddr over the 4.5 years of its existence so far. During the first two years, where mbeddr was developed as part of the government-funded LWES research project[19], 4 developers did essentially all of the work. Since mid 2013, the core team has steadily grown to 8 developers. The remaining 31 have contributed small amounts of code, in total 10% of commits.

At the beginning of the project, two of the four developers (the first and second author of this paper) had significant experience in language engineering with Xtext, the Intentional Domain Workbench and other language engineering tools. One of them had initial experience with MPS from building an early prototype implementation of C with MPS [96]; Team member three had no experience with DSLs, but had built (graphical) modeling tools before. He also had experience with formal verification techniques. Developer four had no previous experience with either of these techniques. The first three had some experience with embedded software and C, but only number three could be considered reasonably proficient.

Of the four team members that have been added to the core team since the research project ended, two had experience with EMF-based (meta-)modeling [79], one was proficient with Xtext, and two had previous experience with MPS (one superficial, the other one deep). All developers have learned the details of MPS-based language engineering during the project – which partially explains why some of the languages were reimplemented later, after the team had gained more experience with MPS.

[19] The first two years of mbeddr development happened as part of the LWES research project, a KMU Innovativ project funded by BMBF under FKZ 01/S11014. The participating companies were itemis, fortiss, SICK, Lear and BMW Car IT.

5.2 Collaboration with Jetbrains

The mbeddr development team collaborates closely with Jetbrains' MPS development team: they report bugs, test milestone releases and contribute extensions to MPS. In return, the MPS team has supported the mbeddr team by explaining MPS details, implementing required features or prioritizing bugs. Both parties see this as a fruitful collaboration.

5.3 Tools

The development mainly relied on MPS itself for all aspects of language development and testing. In this section we introduce the tools we used beyond MPS; we put them into the context of the development process in the next subsection.

git We used git for version control. MPS is fundamentally file-based, so any file-based version control system can be used. MPS provides GUI integration with a wide range of version control systems, git included, so users can pull, commit or merge from within MPS.

Assembla and github For storing the centralized repositories we used a private Assembla repository for the duration of the research project. A private repository was important because it also contains project-proprietary reports. After the research project ended, the code was moved to the public github repository at `github.com/mbeddr/mbeddr.core/`.

Bugtracker We used the bugtracker associated with the Assembla and github repository for reporting and discussing bugs and features.

Wiki Both Assembla and github come with simple Wikis, which we used to document various aspects of the mbeddr design and implementation.

Teamcity We used Jetbrains' Teamcity[20] continuous integration (CI) server for building the languages, packaging the plugins, and running the tests. The build itself relies on ant scripts, so it can be used with any other CI server as well.

Since mbeddr is a version of C, we also used the gcc tool suite for compiling and debugging C code. In addition, we used various verification tools. However, these tools did not

[20] `https://www.jetbrains.com/teamcity/`

play a role in the *development* of the mbeddr languages, so we will not discuss them any further.

5.4 Development Process

Stakeholders and Requirements The primary stakeholders for mbeddr were the members of the development team: several team members had enough experience with embedded software engineering to be able to identify useful abstractions that would be worth adding as language extensions. In addition, the team collaborated with embedded software developers from various associated companies, among them the partners of the LWES research project: SICK, Lear and BMW Car-IT. After one year of development, the Smart Meter project started [107], and the developers of the Smart Meter became additional stakeholders. They helped guide the evolution of the mbeddr C language and its extensions. They also suggested a few additional general-purpose extensions. The mbeddr team also helped develop a few extensions specifically for the Smart Meter project (see [107]). Finally, several requirements for mbeddr came out of the Siemens ESD project which itemis developed with Siemens from 2013.

Development Process The development process was iterative and incremental, with the team learning about the target domain and MPS-based language engineering along the way. We were able to quickly build a first version of a language, try it out, learn from the experience, and then iterate to build better versions. We used an issue tracker, but no formalized requirements engineering process. In the last two years of the project, as external users started using mbeddr and the number of stakeholders grew, the pace of change slowed down and we started collecting requirements more systematically through the issue tracker. As we illustrate in Section 6.1, we wrote a significant number of tests for the languages we developed. We used a CI server for continuous build and testing.

6. The mbeddr Implementation

In this section we describe the implementation of mbeddr in MPS. The goal of this section is mainly to provide an overview over the size (Section 6.1), effort (Section 6.2), timeline (Section 6.3) and structure (Section 6.4).

6.1 mbeddr Quantified

Top-level structure As shown in Figure 2, mbeddr consists of five major parts. *core* contains C itself plus the support for build, unit tests and a few smaller utility extensions such as queues and stacks. *ext* covers the major extensions, including physical units, components and state machines. *cc* is the support for cross-cutting concerns such as requirements, requirements tracing and product line variability. *doc* is a Latex-like language for creating documents that are tightly integrated with mbeddr code. And *analyses* contains support for formal verification of programs written with several of the mbeddr DSLs.

Part	Language	#L	#LC	#S	#C	#LOC
core	base	1	-	1	98	6,163
	C	8	-	7	354	20,114
	unittest	1	1	1	26	1,014
	utils	3	3	1	116	6,306
	build	2	-	1	48	2,080
ext	components	9	9	1	160	11,173
	state machines	1	1	1	48	3,194
	concurrency	3	3	0	65	3,078
	math	1	1	0	11	446
	messaging	2	2	0	60	2,151
	units	1	1	0	30	1,884
cc	variability	7	3	1	87	3,638
	reqmts/tracing	9	1	2	171	5,563
doc	doc	10	-	1	153	6,355
analyses	analysis	18	10	18	170	15,235
Total		81	34	38	1,597	88,394

Table 1. This table lists the top-level parts of mbeddr, the languages/extensions contained in them plus the number of actual MPS modules (separate languages (`#L`), C extensions (`#LC`) and solutions (`#S`)) that makes up each language. We also list the number of concepts (`#C`) and the lines of code used in the implementation (`#LOC`). Of the 81 language modules in total, 34 are actual C extensions (the others are unrelated to C, for example, to express requirements or product-line variability).

Size of the Implementation Table 1 breaks down these parts further. Each part consists of a number of languages (if they are independent of C) or language extensions (if they are C extensions). For various technical reasons, each language may be implemented through a number of MPS language and solution modules. Table 1 also lists the number of language concepts for each of the languages as well as the approximate lines of code (LOC) used in the implementation.

We refer to the *approximate* LOC because, as a consequence of MPS' projectional editor, counting the number of lines is not straightforward: the same code may be projected on one or several lines and some parts of the code may be projected, but do not have to be entered manually (for example, in Figure 3, only the cells have to be typed; the first two lines are mostly rendered automatically by projection rules). Counting the instances involved in the various language aspects is also not a good idea, because each is quite different in size, and so things are not comparable. Instead we use the approach introduced in [101]: we have associated a LOC count with each language concept.[21] For example, a statement counts as 1 line; in a type system rule we count all the

[21] This is not the number of lines *generated* from the DSL code, but an equivalent, linearized, textual representation of the program nodes in the MPS AST.

lines where the user has to enter something (but not those that are purely projected) plus the number of statements in the body of the rule; for editor definitions, we count each cell as 0.5 lines; on average, we estimate 4 editor cells per line.

In total, mbeddr consists of 1.597 language concepts. Many of them do not directly relate to C extensions; for example, 324 are in the documentation and requirements languages. 919 concepts are related to the C extensions. In other words, the extensions are roughly 2.5 times the size of C itself.

The total lines of code for the whole mbeddr implementation (excluding utilities and MPS extensions in the mbeddr.platform, as well as the test code) is 88,394. The highest number is C itself (20,114), the components extension (11,173) and the analyses (15,235). The reasons for the high line count for C and the components is that the languages are large and complex. For analyses, a lot of code has been written to run the analyses in an external verification tool and interpret the results.

It is hard to put these numbers into perspective. One attempt is to compare the implementation of C itself to the implementation of other C IDEs. One open source option is the Eclipse CDT, which, however, also supports C++. We have counted the lines of Java implementation code in the `core` directory, removing all files that contain `test` and `cpp` in their qualified package name, which should roughly represent the non-test code for C only. The resulting number is 562,000 lines.[22] This makes it roughly a factor of 25 bigger than mbeddr C, illustrating MPS' expressive power in terms of language implementation. Another comparison would be to an implementation of C using another language workbench such as Xtext or Spoofax. However, since no reasonably complete implementation of C exists in any other language workbench to our knowledge, we cannot make this comparison. For a general comparison of language workbenches, including the LOC to implement benchmark languages, see [32].

Size of the test code Table 2 provides an overview over the test code for mbeddr, counting test cases, assertions as well as lines of code. As we will discuss in Section 7.5, mbeddr relies primarily on type system tests (verifying static semantics, i.e., whether red squigglies show up where they should) as well as executable tests (verifying execution semantics by running generated C code). We also have a number of regular JUnit tests that verify the behavior of non-language definition code, such as UI extensions or analyses. All in all, mbeddr has 45,695 lines of test code, with is roughly 50% on top of the implementation code (88,394 LOC).

The test code is not evenly distributed over the languages. For example, as a consequence of the complexity of these languages, `ext.components` alone has 35% of all executable tests, and `core` has 57% of all type system tests. `ext.units` has almost no executable tests, because this C extension affects primarily the type system. `doc` has very few of either kind, because it has few type checks, and we generate static Latex code that cannot be executed for testing the semantics. Of the 22,413 lines of regular test code, 10,704 relate to the analyses and 11,749 test the debugger. We also have around 50 tests that verify especially tricky editor behavior.

	#Test Cases	#Assertions	#LOC
Type System Tests	324	1,381	8,243
Executable Tests	310	990	14,999
Other Tests	784	1,275	22,453
Total	1,418	3,646	45,695

Table 2. Breakdown of the test code for mbeddr.

Size of the MPS extensions As we will see in Section 7.3, one of the major ways of managing complexity in language development with MPS is to develop MPS extensions (which are also languages) that simplify language development. We have used this approach extensively, and the resulting mbeddr.platform contains utilities ranging from a debugging framework to new notational primitives to a language for pattern matching over ASTs.[23] In total, our utilities and extensions contain 1,088 language concepts and 75,097 lines of code, making it comparable in size to mbeddr itself. Of these, 43,025 lines are in solutions (i.e., not in language definitions) illustrating the utility nature of the platform.

Distribution among Language Aspects Figure 6 illustrates the distribution of the implementation code among the language aspects for the various languages in mbeddr. The distribution is quite different for the different languages, illustrating their different natures. We discuss some examples. `core.base` has only a little structure but a lot of behavior. This is because it contains a relatively small number of abstract base concepts with lots of utility methods. For the same reason, the type system is small. `core.c` and `ext.components` are relatively similar in their distribution and are typical examples of "real" programming languages: they have relatively equal ratios of structure, editor, type systems and generators. `ext.concurrency` has a comparatively large generator. This is because the language structure of the extension is different from the structure of the generated C code, so the generator is sophisticated. `analyses` has a lot of code that implements IDE support, because it has a number of additional views to show the analysis results. `doc` has a high ratio of structure and editor, but hardly any type system because, as a Latex-like language for writing documents, little type checking is required. Finally, `units` has almost no generator and an overwhelming ratio of type sys-

[22] Note that debugging and build support are outside `core`; counting all of CDT's non-test `*.java` files results in roughly 1.4 million lines.

[23] Despite the fact that it is called mbeddr.platform, its contents have been developed during several MPS-based projects, not just mbeddr.

tem code. This is because the units are a type system extension, and the computation with units in the type system is non-trivial. The generators simply delete everything relating to units during the transformation to plain C.

6.2 Development Effort

Based on worksheets reported by the core developers, the total effort of developing mbeddr and the platform was roughly 16 person years. Splitting this total amount based on the distribution between mbeddr (88,000 LOC) and the platform (75,000 LOC), 8.8 person years must be allocated to mbeddr itself. Considering that we redid some parts of mbeddr because of confusion about what was needed, we estimate that a total of 10 person years was spent on mbeddr itself (i.e., excluding the platform).

Adding up mbeddr's implementation and test code (88,000 + 45,000) and dividing it by the number of developer years (10), each developer, on average, wrote 13,300 lines of code per year. To put this into context, a large-scale study by Jones and Bonsignour [44] has found that, independent of process and language, developers on average write between 3,900 and 9,000 (surviving) lines of code per year.

6.3 Development Timeline

Figure 7 shows the development timeline of the mbeddr project. In this section we discuss some interesting aspects.

Projects mbeddr development started in 07/2011, after a few preliminary prototypes with MPS by some of the team members. Its development continues as we write this paper. mbeddr was initially developed during the government-funded LWES research project. It ran for two years, from 07/11 through 06/13. Roughly halfway through the project, the development of the Smart Meter system [107] project started. In May 2015, shortly before the end of LWES, we were approached by Siemens PLM Software (then: LMS International) about developing the previously mentioned Embedded Software Designer (ESD) on top of the open source mbeddr stack. Development started in July 2013 and continues today.

Resources and Efforts On average, 3.5 full-time equivalent (FTE) developers worked on mbeddr. During the first 18 months, 4 developers worked as 3.2 FTEs. Because of customer interest and the resulting business potential for mbeddr, we hired a fifth developer who started working full-time on mbeddr, resulting in 4.2 FTEs. Shortly before the end of the research project, the Siemens ESD project started. Over the following year, until 04/2014, it increased in volume, reducing the FTEs working on mbeddr to 2.6, with a correspondingly slower pace in commits. This is also a consequence of two of the developers involvement in management of the team as well as sales and marketing activities for mbeddr. We hired two more developers in 04/2014 and another one in July. This brought up the workforce to 4.0 FTEs; their remaining work time went into ESD. From 03/2015 onwards, the FTEs working on mbeddr again started to decline because of number of additional projects (mbeddr-based, as well as in other domains) started ramping up. Note that the overall effort reported in this picture is higher than the 10 person years total effort reported earlier; this is because the efforts for developing the mbeddr.platform could not be factored out completely (see Section 2.4).

Language Development Start Dates We started the mbeddr implementation with `core.c`, because this is the basis for everything else. We also implemented support for unit tests and build very early on so we could write tests for the languages we developed. `core.base` was only started 5 months in when we recognized the need for a C-independent foundation layer. Components and state machines were also started early, as a means to validate that two of the critical extensions can actually be built the way we envisioned them. There is a small peak in `cc.variability`, a very early prototype to verify that our approach to managing product line variability would work.

Analyses were also started in month 5, an indication that the synergy of language extension and formal analyses plays an important role in mbeddr. `doc` was started in early 2013 when we realized the need for efficiently documenting code written in mbeddr. Finally, messaging and concurrency were started only in 2015, when we got a few additional days of research funding from another research project.

Development Effort over time The set of line diagrams in the lower half of Figure 7 illustrates the development effort over time for each language. Specifically, the diagrams show the number of changed (added + deleted) LOC, scaled relative to the size of the language. We also annotate migration events, because these typically lead to a lot of changes in all languages, roughly at the same time (for example, a change in MPS' persistence format touches almost every LOC in all model files and the migration to github meant re-adding everything). Note that the migration to MPS 3.3 was different. There is not a huge peak, because, for the first time, we migrated various parts over time, based on subsequent milestone builds of MPS.

We now investigate the development effort in more detail, pointing out some interesting aspects of the line charts at the bottom. `core.c` had modifications all through the project. This was mainly because we had to fix type system problems over and over again (as we better understood the intricacies of the C type system), and because we had to undo some "simplifications" we added to C initially that turned out to be naive. `core.build`, instead, was hardly ever touched after it had been created initially. Similarly, `core.unittest` was also stable over time. `core.utils` had small scale changes throughout the project as we saw the need for additional utilities. The same occurred with `core.base`, into which we factored reusable abstractions as the potential for reuse became evident.

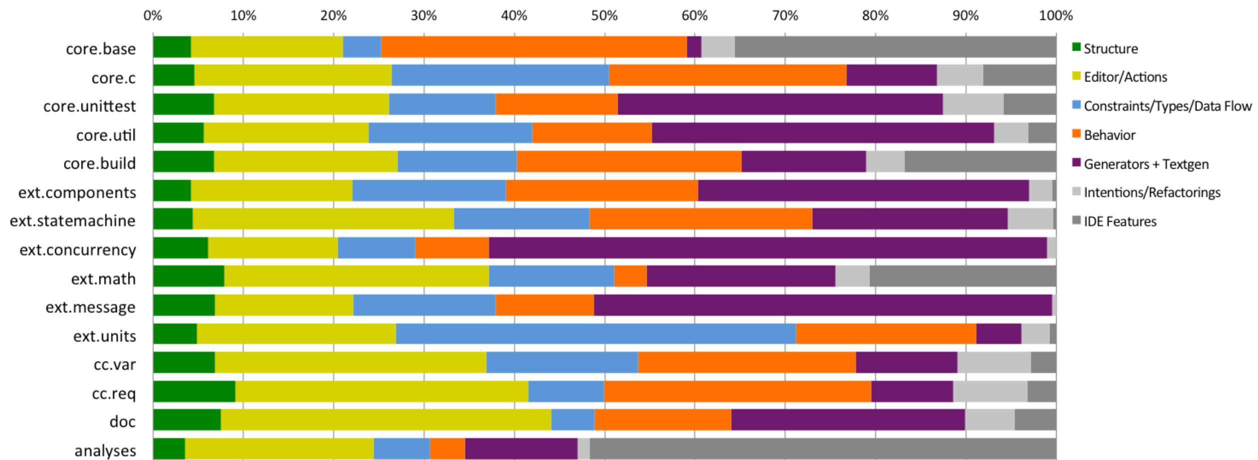

Figure 6. The distribution of the implementation code over the various language aspects. Different languages exhibit different distributions, the reasons for this are discussed in the text.

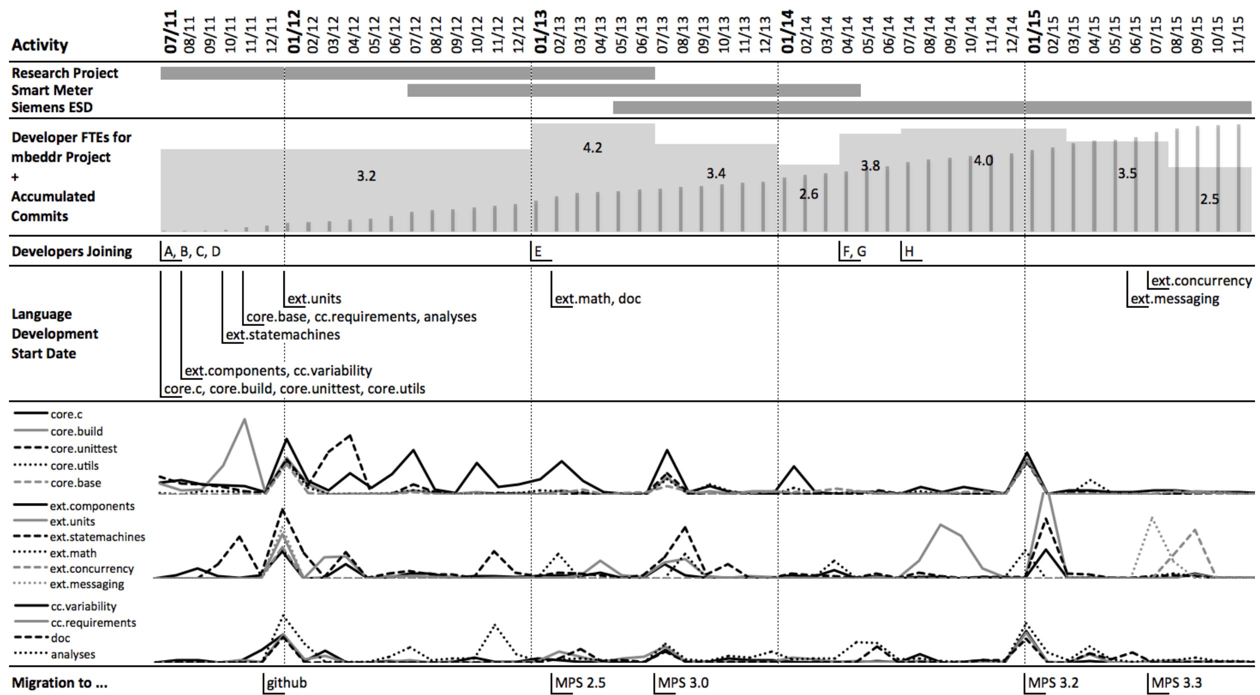

Figure 7. mbeddr development timeline. From top to bottom it shows when important projects (LWES, Smart Meter, ESD) happened, FTEs working on mbeddr and the accumulation of commits in the repository, when the various developers joined, when the development on the various languages was started, plus the time distribution of work on the various languages, scaled relative to the size of the particular language. The bottom row illustrates migrations, which typically lead to a lot of batch code changes.

`ext.components` was modified during the smart meter project, where components were used extensively – a bump is visible in the line in early 2013. `ext.units` has a major peak in the summer of 2014, which is where the language was completely reimplemented: at this point we had a much better understanding of how the MPS type system works, and we were able to provide a more scalable and more flexible implementation of units (for example, we introduced generic units so that functions could work with arbitrary units but still describe how the units of the return type are computed from the units of the input arguments). State machines had a lot of changes in late 2012 when we added hierarchical states and in the summer of 2014 when we added epsilon transitions (implicitly triggered transitions) and junctions (states that only have epsilon transitions). `ext.math` has a bump in the spring of 2014 when we completely reimplemented the notational primitives. `cc.variability` also had changes throughout. For example, we have added compositional variability in 2014 and integrated a variability-aware type system in 2015 when MPS added the necessary new concepts in the type system language.

Implementing the analyses was an ongoing effort that can be split into four phases, reflecting the learning process that we went through. We started by integrating Yices [26] and NuSMV [18] to analyze subsets of C on the model level (by transforming C into the respective tool input languages). Phase two was the integration of CBMC [20] in order to analyze all of C on the level of generated C code. Phase three was the maturation of the CBMC-based analyses and the resulting abandonment of NuSMV and Yices. Finally, the ongoing fourth phase integrates new tools beside CBMC such as Sat4J [54] (for checking the consistency of feature models and presence conditions) and Z3 [24] (we are experimenting with Z3 to verify the consistency of simple decision tables), as well as Spin [42] (with which we experimented for model checking).

6.4 Overall Structure

MPS uses inheritance in the style of object-oriented programming as the means of language extension. Inheritance can occur between language concepts as well as between languages themselves. Only if a language `A` extends `B` can concepts from `A` extend concepts from `B`.

Figure 8 shows a part of the inheritance structures of the `core` languages and of `ext.statemachines`. The concepts that are extended most are printed in bold font. They are `Expression` (extended 270 times), `Statement` (115), `Type` (74) and `IModuleContent` (70). The importance of the first three should be obvious in C; the last one is an interface used for all top-level language concepts such as `structs`, functions, `typedefs` or component declarations.

In terms of language dependencies, `core.expressions` is (almost, see below) at the root of the hierarchy. `core.statements` depends on `core.expressions`, and `core.modules` in turn depends on `core.statements`. Extensions like `ext.statemachines` typically extend all of the three aforementioned core languages because most extensions contribute new top level contents (e.g., the `Statemachine` itself), expressions (e.g., `EventRef`), types (`StatemachineType`) and statements (e.g., `TriggerStatement`).

We mentioned before that `core.expressions` is *almost* the root of the dependency chain. There are two additional levels, both part of the mbeddr.platform (see Section 2.4). In particular, `core.base` contains a number of utilities useful for essentially all mbeddr languages (and others beyond mbeddr). Most importantly, it contains about 40 trait-style interfaces that can be implemented by otherwise unrelated language concepts to benefit from predefined behavior or UI features. Some examples are shown below; the number in parenthesis represents the number of implementors:

- `IIdentifierNamedConcept` (250) used by concepts that require a valid identifier as a name; contributes a `name` property, a constraint check plus some IDE behavior.
- `IReference` (65) implemented by all references; contributes an abstract method that returns the target of the reference as well as a type system rule that propagates that target's type to the reference.
- `ITyped` (53) implemented by every concept that has a `type:Type` child; provides the necessary typing rules.
- `IGenericDotTarget` (74) represents everything on the right of a dot expression (as in `struct.member`).
- `IConfigurationItem` (32) implemented by all concepts that control the transformation process (similar to compiler switches).

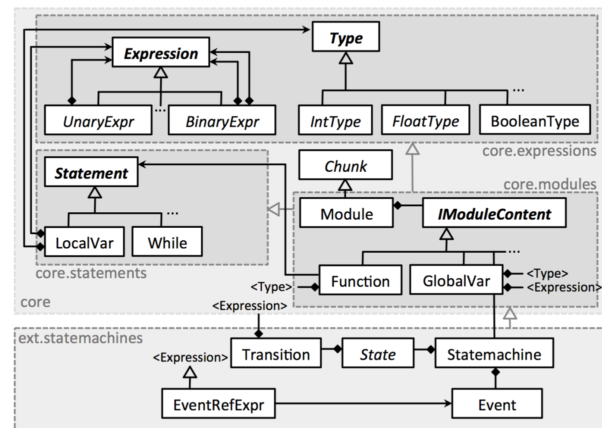

Figure 8. Class diagram that illustrates some of the core structures and dependencies in mbeddr. Italics text denotes abstract concepts, bold text represents important extension points, names in angle brackets act as proxies for the named concept to avoid even more lines. Details about the design are explained in the running text.

- `ITreeViewable` (39) implemented by all concepts whose instances should be viewable in the tree view tool; this tool relies on the interface to define behavior methods that render the tree.

`core.base` contains utilities that reside on the same meta level as mbeddr itself (i.e., mbeddr concepts *extend* them). The second level of additional dependencies are the languages in the `mpsutil` package. These are extensions of MPS languages; so the mbeddr language definitions *instantiate* those. Examples include the tabular, mathematical and diagram notations (instantiated in editors) or the pattern matching language (instantiated in generators). `mpsutil` also contains utilities that are not directly related to languages, such as an embedded HTTP server to control MPS from external applications.

7. Evaluation

In this section we evaluate the research questions introduced in Section 4.1, each in its own subsection. While we rely mostly on the MPS features introduced so far, we introduce and explain some new features as we evaluate them.

7.1 RQ1: Language Modularity

An important feature of MPS, and a cornerstone of modern language engineering in general, is language modularity and composition. The term refers to using a set of independently developed languages together, in one file or model. It refers not just to syntax, but also to the type system, the semantics and the IDE support. Different approaches exist, and different classifications schemes have been proposed [31, 97]. The following approaches are most widely used (cf. Figure 9):

- *Embedding* refers to syntactically embedding an independent language into a host language. Importantly, the embedded language must have been developed without knowledge about, and dependencies on, the host language. This means that it can potentially be embedded into several different host languages. Typically, an adapter language (cf. the Adapter pattern [36]) is required for each embedded/host language combination to fit them together. mbeddr does not use embedding.
- *Extension* refers to adding new language constructs to a base language. In contrast to embedding, the extending language is developed specifically for a particular base language; it depends on this base language and can only be used with this base language (but does not modify that base language invasively). mbeddr's state machines are an example; Figure 8 shows how this extension works: the `StateMachine` concept extends the `IModuleContent` interface defined in mbeddr C.
- Once several extensions have been developed for a particular base language, each independent from all others, *extension composition* refers to the ability to use several of these independent extensions in a single program. For example, state machines can be used together with physical units in the same program Sometimes, when the semantics of extensions need coordination, adapter languages are required for specific compositions of extensions. We have one such mbeddr language in mbeddr: embedding state machines into components.

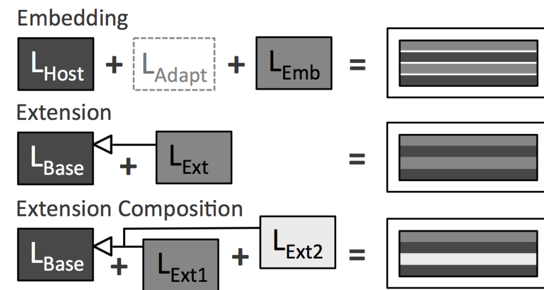

Figure 9. Three important kinds of language composition: embedding, extension and extension composition

mbeddr relies almost exclusively on extension and extension composition: mbeddr consists of 34 extensions to C (see Table 1), all of which can be used together in a single program. The requirements and doc languages act as base languages as well, each having their own set of modular extensions.

Language modularization is essential for managing complexity in language development, just as it is essential for software engineering in general [80]: it helps break down a large language into several smaller ones that are easier to understand and maintain, it makes building and testing each language module faster, and it allows new languages (or language extensions) to be developed without changes to the other modular languages, and in particular, the base language (in case of extension).

Modularization is also important for language users, because they can use only those features (extensions) relevant to their particular task or skill level. This has turned out to be useful in mbeddr [107], since it allows mbeddr applications to be developed bottom-up: first implement C, and then, incrementally, add domain-specific abstractions as modular language extensions. Users can skip down to the next lower abstraction level if an abstraction is not suitable or if it introduced too much performance overhead (a very important concern in embedded software). Based on a case study in enterprise systems, [57] suggests that "the use of hand-written code [..] was instrumental in implementing the fine details of some functional requirements. We propose that designers of model driven development environments introduce such a feature [..]" mbeddr supports this not by generating skeletons that users can then fill in with "manually written code" in an external IDE. Instead, users can write lower-level code *directly in mbeddr*, avoiding integration issues with the external IDE. [57] continues to conclude

with "[..] and at the same time try to prevent [the user of manually written code] by (incrementally) perfecting their modeling languages." This is exactly what has been done in mbeddr by successively adding language extensions.

MPS supports different composition techniques for the various language aspects. For the structure, composition works roughly similar to object-oriented programming, in that a language concept can extend a base concept (with which it is then polymorphically replaceable) and implement several interfaces. The syntax, behavior, actions, intentions and refactorings can be understood as methods (defined with their own particular DSL) on these OO-style structures.[24] In the type system, language concepts contribute typing equations which are then solved by a solver. This approach is declarative, so a language can contribute additional typing rules for existing and new language concepts; in the rare cases where conflicts arise, a rule for a subconcept can specify that it overrides the inference rule for a superconcept (we have two such cases in mbeddr, both related to pointers). For generators, composition is based on priorities that determine the execution order of transformations. Composition is specified by declaring pair-wise priorities of a particular generator relative to other generators. Almost all of our C extensions only specify priorities relative to C itself, and not to other extensions (which would compromise the independent composability of extension composition). We discuss our experience with each of these composition mechanisms below. Details on language composition with MPS in general can be found in [97].

Fundamentally it works and scales Fundamentally, MPS' support for language modularization and composition scales well in terms of managing the complexities of language composition, as exemplified by mbeddr's 34 extensions of C. Importantly, the extensibility does not just apply to coarse-grained syntactic blocks (as in Converge [85], for example). In contrast, modularity works down to the expression level, which is particularly important, for example, in the extension that supplies mathematical symbols to C [100], which supports new expressions (such as abs $|x|$), but also the sum symbol which contains other, regular C expression in its body, as exemplified by $\sum(x+i)$.

Annotations Annotations are another mechanism available in MPS for implementing language extensions. Specifically, an extension adds arbitrary child nodes (the *annotation*) to an existing node (the *annotated node*), without changing the definition of the annotated node. This supports adding arbitrary additional data into an AST. In terms of concrete syntax, the annotation can contribute concrete syntax to the annotated node (above, below, left or right of the annotated node), or even replace it completely. There is no limit to the syntactic structure of the annotation, because, as a consequence of projectional editing, no interference can happen with the syntax of the annotated node.

Annotations are frequently designed to be independent of the annotated language, which makes them a form of embedding instead of extension (where the extension depends on the base language). For example, a comment or an architectural layer constraint may be attached to any program node. Since the language that defines the original node cannot know about the annotation, the type system and the transformations will not take into account the annotation. This means that the semantics of the annotation either must not change the semantics of the annotated node, or the semantics are implemented by a preprocessor that is built specifically for the annotation. We describe a couple of example annotations in the remainder of this paragraph, grouped by how the semantics are realized.

The simplest case for annotations are those where the semantics are completely generic and independent of the semantics of the annotated node. For example, an annotation that stores the code review state (`new, ready for review, reviewed`) of a piece of code only stores the name of the reviewer, the date of the review, as well as a hash of the reviewed code to detect changes. Checking rules specific to the annotation verify the validity of a code review, but there is not semantic relationship to the annotated node.

The second case are annotations where the semantics affect the annotated code, but in a generic way. For example, mbeddr's presence conditions [22] (Boolean expressions that determine whether a node is part of a given product line variant; think `#ifdef`) are processed by removing the annotated node from the tree if the the condition evaluates to false. Similarly, comments are simply pushed down the transformation chain, so that they eventually end up in the generated code. These behaviors can be implemented independent of the specific semantics of the annotated nodes. Note that another case of annotations with generic semantics are the transformations macros (such as `COPY_SRC`, `IF` or `->`) used in the generator (and explained in Section 3.7).

The third category of annotations are those that *are* developed specifically for a given language. While the dependency is similar to language extension (because the language that defines the annotation has a dependency on the language that defines the annotated concept), there is an important difference: a concept can be annotated without changing the identity of the node. In this case, the semantics of the annotation relates to the semantics of the annotated node. An example of this type of annotation is the `checked` annotation for state machines. If attached, the generator for the state machine generates additional code (i.e., has different semantics) that is used to verify properties using the CBMC [20] model checker. The annotation can be attached to existing state machines; if an extension had been used, the existing instance of `StateMachine` would have had to be replaced with a (dif-

[24] There are restrictions. For example, in an editor one cannot call *super* to delegate to the editor of the base concept.

ferent) instance of `CheckedStateMachine`, breaking references to that existing state machine.

Generally, annotations are an essential ingredient to language modularity, and as the examples above illustrate, mbeddr makes use of them extensively. Nonetheless we have encountered a few limitations. For example, the annotations are not automatically attached to nodes created during a transformation. This means, for example, that the above-mentioned comments have to be pushed down explicitly in every transformation. This is tedious and error prone. The second limitation is that annotations cannot affect the text generators. We encountered this problem when trying to generically generate `#ifdef`s for mbeddr's presence conditions (instead of generically deleting all nodes where the presence conditions evaluated to false). The third limitation was that an annotation cannot affect the type of the annotated node. This feature is necessary to implement typing rules where the type of a node can vary based on the product variant. All of these limitations have been fixed in MPS in the meantime, based on the experience from mbeddr.

Limits to Non-Invasiveness In terms of structure and concrete syntax, MPS' language modularity and composability is analogous to object oriented programming in Java. For example, a concept can extend another concept and add children, override the editor or provide new behavior methods. However, as we know from Java, this approach also has certain limitations. For example, sometimes Java requires refactoring a class hierarchy to introduce abstract classes or interfaces that serve as extension points. This is also true for language implementations in MPS. We sometimes had to invasively modify the C base language to introduce abstract concepts or concept interfaces. An example is `IAssignmentLike`, which is implemented by all concepts that implement an assignment (such as `+=, *= or |=`) so we can generically check whether the expression on the left is an lvalue. Strictly speaking, this breaks the modular extensibility of languages. In practice, however, we did not perceive this as a major problem, since the number of such changes got lower over time as the language structure matured (again, similar to the process of designing object-oriented programs). We are confident that not many such invasive modifications will be required in the future.

The invasive modification did not have an impact on end users. Just introducing abstract base concepts does not impact existing models; the refactorings are internal to the language definition. Those changes that have consequences for existing models were handled with MPS' support for migrating models. For details on model migration see the Evolution of Models paragraph in Section 7.5.

Tree Constraints In object-oriented programming, and also in MPS language design, a concept `A` that extends concept `B` implies that an instance of a `B` can be used wherever an instance of a superconcept `A` is expected. However, this can be a significant limitation. Consider test cases: inside of a test case, a range of `assert` statements should be available. To make them available in a test case body (a `StatementList`), they have to extend `Statement`. As a consequence of polymorphism, this design means that `AssertStatements` can now be used anywhere a `Statement` is expected (such as in regular functions or components runnables, which also use `StatementLists` as their body). However, this is semantically wrong. The alternative of making a body of a test case use something other than `Statements` is not a good alternative, because all of C's remaining statements must be legal in tests as well. To solve this apparent contradiction, MPS supports tree constraints. A tree constraint decides whether a particular concept can be a child of some other concept. For reasons of dependency management, such a constraint can be written from the perspective of a parent ("can I have this concept as a child") or from the perspective of the child ("can I be used under this parent"). As we have seen with the `val` expression in the tutorial in Section 3, these constraints actually prevent nodes from being instantiated, so the user cannot enter the respective node. This effectively removes the respective concepts from the language *in a given context*. Returning to the `assert` statements, this means that they can use a tree constraint to limit their use to inside ("under") a test case.

The approach works well, and we have used it in many locations; mbeddr has 254 tree constraints. The problem is that there is a tendency to forget to write the constraints, and the transformations may break because of the incompatible semantics; for example, in one case, operators specific to the analyses have shown up in regular C, because they extended `Expression` without suitable constraints that limited their use to analysis specifications. We used extensive testing and automatic test case generation to address this issue in mbeddr, as will be discussed in Section 7.5.

Overmodularization When designing the structure of mbeddr C, we decided to modularize C itself. We created separate MPS languages for primitive types and basic expressions (`core.expressions`), for statements (`core.statements`), pointer types and their operators (`core.pointers`), user-defined types such as structs, unions and enums (`core.udt`) as well as modules and top level concepts such as global variables or functions (`core.modules`). The reason for this modularization was the expectation that we would reuse some of these languages independent of others (e.g., use a safer subset of C without pointers, or using the expressions in an external state machine DSL).

However, it has turned out over time that there are dependencies in both directions and the modularity cannot really be sensibly maintained. In order to retain an adequate layering of these languages and avoid cyclic dependencies, we had to introduce a number of interfaces in some of the languages (we estimate one third of the 380 interfaces are playing this role in mbeddr and all of its extensions). In retrospect we should have modularized the C core much less.

Probably only `core.expressions` makes sense as a separate language: we have reused the C expression in many other DSLs that generate code down to C.

Language Refactoring Moving concepts between languages is not a trivial matter. A concept is implemented via several aspects, each residing is its own model inside a language. These models have dependencies expressed as imports. Also, models that use a concept in a program have a `use` relationship to the language that defines the concept. When a concept is moved from one language to another (as happens when changing the modularization structure of languages), all these dependencies have to be updated. If the models that use the language are not in the current project (they may reside on customer machines), migration scripts (see Section 7.5) have to be created. More generally, since the modularization structure is hard to get right a priori, strong refactoring operations are needed in language workbenches that support language modularization and composition. While MPS provides some of those, this is an interesting area for further research.

As a practical matter, at the time when mbeddr was built, some of MPS' refactoring operations have not always been reliable and thus lead to invalid, unnecessary or circular model imports. This means that changing the modularization structure of languages is more complicated than it needs to be – and consequently is often not done. Some of our languages, in particular, `core.base`, thus have become a huge collection of unrelated features that require separation into separate languages, and we do not dare to remodularize it. This is also the reason why the aforementioned overmodularization of the C implementation in the mbeddr core has not been cleaned up yet.

Extension Composition As mentioned above, extension composition refers to using several independently developed extensions of a common base language in the same program [31]. We distinguish two cases, which we discuss below.

In the first case, the extensions have neither structural/syntactic nor semantic interactions. For example, a state machine can be used in the same program with an interface or component: both of them are top-level module contents (illustrated in Figure 10 A). Another example is the use of units inside a state machine transition guard. Even though the two are syntactically nested, they have no interactions because the units attach to types and literals from C, which are transitively used by state machine guards (Figure 10 B).

As a consequence of MPS' OO-style model for language modularization and composition, and because of how projectional editing works, there are *never* any syntactic interactions when independently developed language extensions are used in the same program – this is why units inside state machines are not a syntactic problem. However, this is not true for the execution semantics: it is possible that the two generators for independently developed languages produce code that does not behave as expected in terms of structure

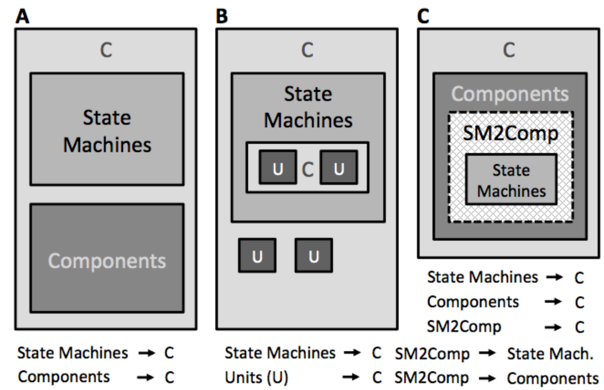

Figure 10. Different scenarios for extension composition. `A:` two independent extensions used in the same C program. `B:` nesting independent extensions within each other. `C:` making independent extensions collaborate by using an adapter language.

(resulting C code may not compile), functionality (the program behaves erroneously) or non-functional properties (the program becomes unexpectedly slow). In particular, it is not possible to detect such semantic interactions statically by analyzing the generators. This is because transformation code contains significant amounts of procedural BaseLanguage code that is practically impossible to analyze (a limitation of MPS we revisit in the Declarativeness and Analyses paragraph in Section 7.3). While, in our experience, at least structural and functional problems occur only rarely (we cannot remember such a case in mbeddr), it is still a significant limitation that this cannot be statically guaranteed. In particular, this will become a problem once mbeddr should be qualified [21, 41] for use with safety-critical systems, where some level of guarantees must be made regarding the correctness of the generated C code.

We have addressed this problem by writing test cases (programs that use more than one extension) for those compositions of extensions that were found most often in application projects. Some of these tests have been generated [67]. Note that, with 34 C extensions, it is infeasible to test all combinations. If an interaction were detected, it could be resolved by actively preventing the use of the two extensions in the same program (which of course puts some kind of dependency between the two languages, compromising independence) or by changing one or both generators in a way that avoids the interaction. The few examples of unintended interactions we have found in mbeddr were related to name collisions in generated variables because the MPS transformation framework is not hygienic [33].

The second case refers to the situation where two extensions *should* interact in a controlled way. In mbeddr, we have two such cases: it is possible to embed a state machine into a component (Figure 10 C), and it is possible to use a component runnable as a task in terms of the concurrency language.

In this case, one could create a dependency between the two languages. For example, the state machine language could have a dependency on the components language and contain language concepts that make the state machine fit into a component. However, this would compromise the required independence of the two extensions. A better solution is to factor the adapter code into a separate language (`SM2Comp` in Figure 10). This way, the two original extensions remain independent, and the adapter language deals with adapting syntax and semantics. The language that moderates the interactions between components and state machines is called `ext.components.statemachine`; among other concepts, it contains a concept `StatemachineComponentAdapter` that adapts `StateMachines` to `Components` and `RequiredOperationBinding` which allows component `Runnables` to be used as a state machine's out event bindings.

Generator Exchange and Orchestration Idiomatically, MPS uses transformations from AST to AST as the means to define the semantics for languages (see Section 3.7). In the context of extension, the transformation maps a concept from the extension language to an "implementation" based on concepts in the base language. Typically, there are several such mappings involved to map a high-level C extension to executable C code, so generators are stacked. In a test model that uses all of mbeddr's languages, the generator scheduler computes 40 phases (AST to AST transformations) from the dependencies expressed between the languages used in the model.

As an example of step-wise transformations consider the mocking [82] extension for components. It transforms to the components language, which transforms to C + utilities, which in turn transforms to plain C. This stacking leads to the challenge of defining the order of the generators, because a generator for a language `D` that maps `D` concepts to concepts defined in some language `I` has to run before the transformation that further transforms the `I` concepts. Expressing this ordering constraint inevitably leads to a dependency between `D` and `I`. If `D` is an extension of `I`, then there is already a dependency between the languages, and the transformation dependency does not make the situation any worse (see Figure 11 A-C).

If two extensions `D1` and `D2` of `I` interact, then an ordering constraint between `D1` and `D2` may be required: this compromises the independence of `D1` and `D2` and is generally not desired (see Figure 11 D). While we did not encounter such a case in mbeddr, a hypothetical example could be where `D2` is the components language which transforms `Runnables` into `Functions` and `D1` is a tracing language that adds a trace point to every `Function`. It is important that the runnables are transformed into functions *before* the trace points are added to all functions (to make sure the functions generated from runnables also get trace points), even though the two extensions have no other relationship and hence are independent[25]. We have no such case in mbeddr, although the trace point example is a realistic one.

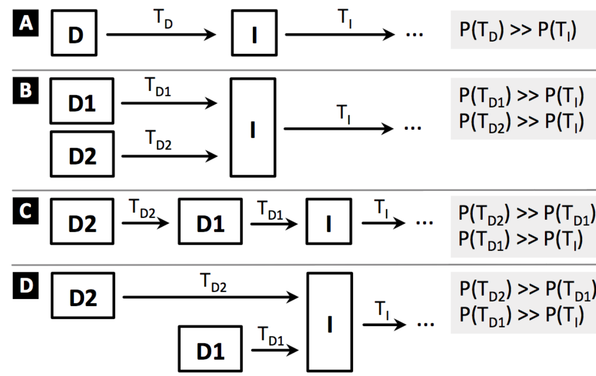

Figure 11. Ordering constraints for generators. **A:** simple ordering, where an extension generator runs before the generator of the base language. **B:** Two independent extensions both expressing an ordering constraint relative to the common base language. **C:** Generator stacking, where an extension `D2` of `D1` must be transformed before `D1`. **D:** Two independent extensions of `I` who nonetheless have an ordering constraint between each other.

An important form of modularity is the ability to define several alternative generators for the same language concept(s). Typically, their semantics will be similar in terms of functionality, but differ in terms of non-functional properties. We have used this approach for components: they can be translated with function pointers (supporting polymorphism, but somewhat slower and requiring more memory) or via direct function calls (no polymorphism, but faster and using less memory). While the ability to exchange generators adds flexibility, it also makes testing more cumbersome: it must be assured that the functional semantics are identical for all generators, typically by running the same tests for all generators.

Another issue is the case where a particular instance of a concept must be mapped to an instance of several target concepts at the same time (so this is different from the alternative generators discussed in the previous paragraph). This does not directly fit with the MPS approach of reducing nodes into other nodes, because a reduction *consumes* the node and replaces it with the transformation result – so the source node is gone after the first transformation. We have encountered this problem in the context of early analyses where, in addition to generating the executable code, we also generated the input to the analysis tool.

[25] Another way of solving this issue is to make functions and runnables implement a common interface, and then make the tracing language insert trace points to all nodes that implement that interface; however, this would lead to the need to (retroactively) introduce this new interface, requiring invasive changes to the components language. This is also not desirable.

We use an idiomatic approach to solve this issue. We clone the original node first (typically with a script involving `node.copy`), and mark the clone by reflectively storing a flag in the node's user objects[26]. We then define several reduction rules, the first one reducing the original node, and the second one reducing its clone with the flag set to true. While this works, first class support for "branching" the generation chain would be very useful. Jetbrains is planning to add support for this in MPS 3.4.

Summary for RQ1, Language Modularity:

mbeddr's 34 extensions to C are a clear indication that MPS' language modularity works. Modularity is useful for language understanding, testing and reuse.

In rare cases, modularity is compromised by necessary changes to the base language and unwanted dependencies between independent extensions.

Currently there is no way to detect (unwanted) semantic interactions between independent language extensions through analysis of their transformations.

7.2 RQ2: Contribution of Projectional Editing

We feel that the support of projectional editing for language modularity and mixing notations has been an enabler for managing the complexity involved in building mbeddr (and for making it useful to end users, as discussed in [107]). At the same time, projectional editing also has disadvantages in terms of usability and infrastructure integration. In this subsection we discuss the trade-off.

Language Modularity As mentioned earlier, projectional editors never run into ambiguities when composing extensions. This was a significant contribution to mbeddr, because languages extensions could be designed without considering the syntax of other extensions, and/or without writing disambiguation code for every possible combination of extensions.

We now investigate the claim that projectional editors never run into ambiguities when composing extensions in more detail. If two extensions define the same syntax for different concepts, this may be confusing for the user, but the underlying data structure is still unambiguous once it has been entered. Similarly, if two extensions define the same alias (i.e., the same way of *entering* the construct), then the user, when typing the alias, has to use the code completion menu to make a decision. In other words, disambiguation is delegated to the user. While this can be seen as a drawback, it is still better than failing to compose the extensions completely. The problem of overlapping aliases only affects the user if both extensions constructs are valid *in the same location*. Constraints, as discussed before, limit such situations. In mbeddr this happened only very rarely. One example is the `{` alias, which is used for entering statement lists `{...}` as well as closure literals `{ => ... }`. Both are allowed in statement context, hence the overlap. Users have to manually disambiguate. `{` is also used for array init expressions (as in `int8[3] = {1, 2, 3};`). However, they are only allowed in variable initializations (where the other two are not allowed), so there is no conflict in this case.

If two language extensions that have a syntactical similarity or alias overlap are used together regularly, then it is always possible to define a third language that (non-invasively) changes the syntax of one of the two or uses actions to effectively change the alias of one of them. We have not seen the need for such a language so far.

Flexible Notations The ability of projectional editors to support a wide range of notations [100] in the same editor was very useful; Figure 12 shows a few examples. Notational flexibility is not just a gimmick, it was actually useful in practice, as illustrated in the smart meter case study [107]. For example, connecting component instances is naturally done with a graphical notation (Figure 1), state machines rendered as tables (Figure 1) or diagrams help understand their structure, and mathematical symbols improve readability for complex mathematical expressions.

What is particularly important is that all the different notations are based on the same editor architecture: this means that the different notations integrate seamlessly from a user's perspective (e.g., in terms of interactions). It also means that, from the perspective of the language developer, the editors for the different notations are defined in the same way (instantiating and parameterizing editor cells, see Section 3.3). For example, to embed text in a graphical shape, one just has to use MPS editor cells for textual notations as the `editor` part of the editor for the graphical node (see Figure 13). The effort for building such mixed-notation editors is not higher than for building single-notation editors, once the respective primitive editor cells for text and diagrams have been defined (the mbeddr.platform contains such primitive cells for math, tables and diagrams). This is in sharp contrast to mixing notations with classical editor technologies (for example, integrating text and diagrams in Eclipse [92]).

Decoupled Primitives MPS editor definitions rely on the instantiation and parametrization of editor cells. Once new editor cells are defined, they can be used in multiple languages. This way, the notations themselves can be reused, independent of the language (which defines structure and semantics). For example, mbeddr uses the same math notation cells as an insurance system (which is not built on top of mbeddr C); two different languages (with different semantics) rely on the same cells for the definition of notations. Similarly, the languages for connecting component instances and for graphical state machines both rely on the diagram notation, and the table notation is used for both decision tables and state machines.

[26] A map associated with each node.

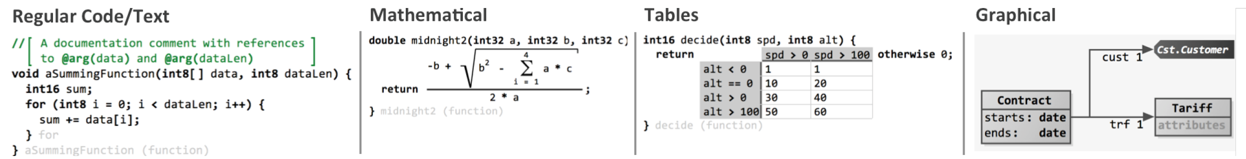

Figure 12. Illustration of the syntactic flexibility afforded by projectional editing. The comment in the code example is unstructured text with embedded real references (to arguments in this case). The tables and diagrams can be embedded in text, and text can be embedded in diagrams, each time with full IDE support.

Developing new Notations Since MPS is bootstrapped, all the languages used in MPS for language definition can be extended in the same way as any other language built with MPS. This is also true for editor cells, the language for defining the notation of language constructs. The editor cells for rich text, tables, math and diagrams have been built as modular extensions to MPS' editor definition language. The diversity of these notations illustrates the flexibility of MPS' underlying architecture. The effort for implementing such languages depends on the kind of notation. MPS was originally developed with textual notations in mind, so the more the new notation is different from text, the more effort it is to develop. For example the mathematical cells were developed in 8 person days. The table notation was developed in 2 months, and the diagrams was roughly 8 months of effort. The latter also required a few patches to MPS' editing framework which made assumptions about the notations being fundamentally line-oriented. In particular, the layout algorithms had to be adapted; in the meantime we have implemented a new top-down layout that replaces MPS' default bottom-up layout. Note, however, that this effort had to be spent only once; as discussed above, these notations are now available for use with arbitrary languages.

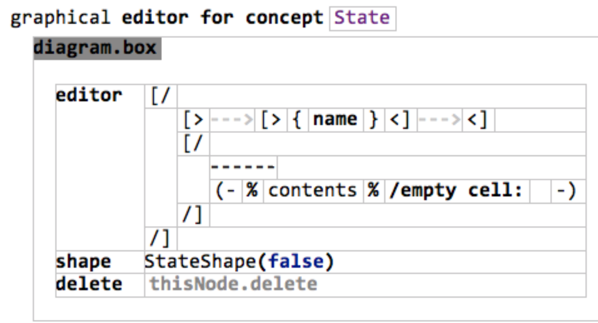

Figure 13. The definition of the graphical editor for a `State` (as part of a state machine diagram). It uses the `diagram.box` primitive, delegates to the `StateShape` shape to actually render the rounded-edge rectangle, and them embeds regular MPS editor cells as the content editor. Here, the embedded editor contains the name of the state (`{name}`), a horizontal compartment (`--`) as well as the list of `contents` of the state (transitions, entry actions, exit actions). These use their own textual editor for representation.

Editor Usability Projectional editing, while having a lot of advantages, also has drawbacks. The two primary ones are that the editor feels similar, but not identical, to a text editor when editing textual notations and that programs cannot be edited, or diff/merged outside the tool. However, diff/merge in MPS works well, and the editor usability for textual notations has improved over time. As our survey [105] indicates, these issues are less and less of a problem in practice.

We encountered two issues regarding editor usability that are fundamental and worth discussing here. The first one relates to notations whose on-screen representation is different from the way they are entered. Traditionally, for example in an `if` statement, the construct can be entered by typing what is seen on the screen ("if" in this example). However, compare this to the $\sum \square$ or $\sqrt{\square}$ symbols used as part of the math notation as shown in Figure 12. There, the user has to type the alias `sum` and `sqrt` to enter the symbols. This is hard to find out and we have added a palette (Figure 14), similar to diagram editors or the formula editor in MS Word, to help users enter these constructs. The palette always shows constructs dependent on the current editor position and the languages included in the program. We have generalized the palette to be able to show arbitrary actions (not just help with node creation), depending on the current context.

The second problem is that, while it is possible to define languages with a textual notation that are good enough to be usable [105], the effort for building them, and for building them consistently within one and between several languages, is still too high. The reason is that the editor behavior (i.e., how the editor reacts to a user's editing gestures) must be implemented manually, using actions. It is easy to forget an action (for example, for deleting elements) or to implement different behaviors for the same editing gesture in different languages, confusing users. The solution to this problem is to put more behavioral semantics into the editor cells used

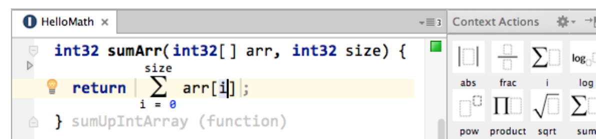

Figure 14. The context actions palette that helps users enter language constructs whose syntax is dissimilar from the way they are typed.

for defining the editors, and then automatically generate the actions from the semantically enriched editor cells. We have since developed such cells and used them in several language projects [108]; the experience with these cells suggests that they are a significant step forward regarding the definition of usable projectional editors.

Integration with Version Control As mentioned above, version control integration of a projectional editor requires the use of the editor (MPS in our case) for diff and merge operations. MPS supports this for arbitrary notations, and we have used a git-based workflow for several years now. For the graphical notations we encountered a problem: we did not want the merge algorithm to report conflicts just because the position and size of diagram nodes was changed by both users. To solve this issue, Jetbrains has added a feature to MPS where the language designer can mark properties with `merge-ours` or `merge-theirs` annotations to force the respective merge strategy and never report a conflict.

Another conceptual problem relates to merging of language definitions and instance models at the same time. MPS requires a working (and compiled) language definition to render programs expressed in this language. If, in a single update from the version control system, both the language and its instance models change, then the editor for the instances cannot be rendered in the diff view because the language has not yet been updated and compiled. To remedy this issue, one should first update/merge the language definitions, build the language, and then update/merge the the instance models. Note that this problem only occurs during language design and implementation, and not during normal language usage.

Multiple Notations MPS supports the definition of several editors (with different notational styles) for the same language concept. Each editor is associated with an editor hint (essentially a tag). By "pushing" editor hints on a given program, the program's notation can be switched. Different hints can be pushed on different parts of a program, so several instances of the same concept can be shown with different notations in one editing session. We have used this feature extensively. For example, a state machine can be edited as text, table or diagram. Similarly component instances can be connected graphically or textually. The approach works well in general, but there are two problems.

The first problem relates to the fact that (some) editor behavior has to be implemented explicitly with actions. The behavior of the editor may depend on the projection. Consider infix (`a + b`) versus prefix (`+(a b)`) notations for expressions. In the former case, an action is needed to handle a user's pressing + on the right side of variable references (`a` or `b`) in order to construct the infix binary expression that represents +. In the case of the prefix notation, this action should not be available because it is not applicable. The problem is that currently, such transformations and other editor behavior actions are not specific to the various notations assignable to the editors – they can only be defined once for every language. This is currently being fixed by the Jetbrains team.

The second problem is that sometimes notation influences structure. For example, the textual notation for state machines intuitively leads to a structure where the state machine owns a number of states which in turn own entry actions, exit actions and transitions; each transition points to the triggering event as well as the target state; Figure 15 shows this structure. A textual syntax for state machines can directly resemble this structure. However, a tabular notation for state machines that renders states as column headers, events as row headers and the transitions in the content cells suggests a structure where the state machine owns the transitions; in addition, the representation is not a tree, but more like a matrix where a transition has to be located in the table by its source state and its triggering event (row and column, respectively). This is especially critical since, by default, MPS can show a node only in the location that corresponds to its position in the tree (the parent editor cell of a node N must be the cell rendered for N's AST parent node). To address this apparent contradiction, the newly developed editor cells for tables and diagrams rely on expressions to allocate content, and not just references to links in the structure of language constructs. In other words, an arbitrary expression can be used to "collect" one or more cells from anywhere in the tree to show in a given location. In the example of the tabular notation for state machines, each (non-header) cell contains an expression that collects all those transitions whose originating state corresponds to the current row header, and whose triggering event corresponds to the event located in the current cell's column header. We have generalized this ability to collect to-be-projected nodes from arbitrary locations in the tree to so-called `querylists`, i.e., lists whose contents can be dynamically assembled through a query expression. We use this construct to render nodes in locations other than their location in the AST. We use querylists in many places. The most prominent one is tooltips for references, where the tooltip contains a rendering of the reference target node.

Partial Projections A special case of the ability of having multiple projections for the same language concept is partial projections. This is useful to hide information that is not relevant to a particular stakeholder or for a specific step

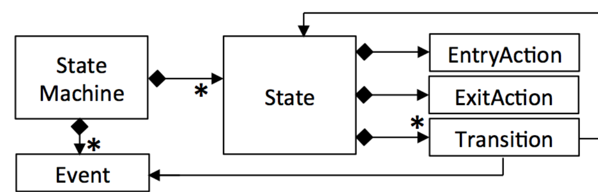

Figure 15. An AST structure for a state machine language where the transitions are owned by their source state. It directly fits a corrsponding textual notation, but does not fit with a tabular notation for state machines.

in the development process. To avoid confusion, this should only be used for program elements whose absence does not change the semantics of the core program (for example those that are added in via annotations, see paragraph Annotations in Section 7). For example, we use this feature to optionally hide requirements traces [104][27] (programmers may not want to see them while optimizing an algorithm) or review state (the coloring of the code can be distracting). Hiding parts of programs can be achieved either by defining separate editors using the mechanism explained in the previous paragraph or by using `IF` cells in the editor definition. There is one problem, though: type system rules and other checks are evaluated on the AST, not on what is shown in the editor. So if a hidden program element has an error, this error is still shown in the gutter of the IDE, while the node that causes the error is not projected in the editor; obviously, this can be confusing to users and should hence be avoided by the language designer. In mbeddr we have this problem in the state machines: the tabular notation does not show entry and exit actions.

Enhanced Projections We developed a new editor cell called a conditional editor. It can be used to intercept the rendering of existing concepts and show additional information above, below, left or right of the original editor. An arbitrary condition can be used to determine which concepts should be intercepted. For example, the following condition can be used to identify program nodes (`n`) that are variables and that have vector type, useful for rendering an arrow above them (as in $\vec{x}$); this example is taken from mbeddr's math extension:

```
n.isInstanceOf(VarRef) && n.type.isSubtypeOf(VecType)
```

In contrast to annotations, which can also be represented above, below, left or right of the annotated element, conditional editors only change the editor and do not insert a new node into the tree. Data needed by the conditional editor can be retrieved from places other than the AST. In mbeddr, we use the approach to change the background color of program parts according to whether they have been covered by a test suite, or to annotate test cases and `assert` statements according to whether they have failed or succeeded. In both cases we use external files (written by the test execution framework) to determine how to color the nodes.

Projectional Editing and Transformations Projectional editing supports language composition. Thus it is possible to compose the transformation language with the language of the to-be-generated program (C in the transformations of mbeddr's extensions). Consequently, MPS supports using the concrete syntax of the target language in the transformation templates (see Figure 4). This is useful because, compared to text-based template engines where the to be generated code is "just text", MPS can provide full IDE support for the target language while writing transformation templates. However, this comes with two drawbacks.

First, it means that when generating a reference, the *template code* has to contain the reference target, even if the target itself should not be generated. This is, because in valid target language programs, a reference can only be written if a target for the reference is available in the code as well. This leads to the use of scaffolding (as shown in Figure 5), which is hard to learn and sometimes leads to repetition and bloat in templates. Second, the composition mechanism between the transformation language and the target language is embedding, so no dependencies between the two are allowed. Consequently, the target language definition cannot allow for "meta escapes", the expressions that navigate, and retrieve nodes from, the input model. The way this is solved in MPS is that annotations are used to attach generator macros to target program nodes. This, in turn, leads to two consequences. The first one is that the meta escapes are not shown in the template code itself, but in the inspector. Some language developers find this confusing. The other problem is that the template code contains nodes whose sole purpose is to act as anchors for macros; they will be replaced during generation.

Some new MPS users have reported that this was hard to learn initially, but became clearer with more experience. Language developers who are not comfortable with the approach can avoid template-based transformations altogether and rely on abstract syntax-based transformations in transformation scripts. These are essentially procedural Base Language code with embedded tree literals or builders (expressions that use nested node constructors). Figure 16 shows an example. mbeddr uses generation scripts mostly for algorithmically complex or generic transformations, and not to avoid the problem described above. Examples include removal of units before code generation, flattening of hierarchical components or state machines, and the removal of nodes from the AST whose presence conditions evaluate to false.

Handling Comments There are two kinds of comments in program code: documentation comments that contain prose (perhaps with references to program elements) and program code that has been commented out. In parser-based editors the two kinds are technically similar – both are just sequences of characters, preceded or surrounded by whatever token represents comments (`//` or `/*` and `*/` for C). However, in a projectional editor, the two kinds must be distinguished, because commented code must retain its tree structure so that it can be uncommented later.

MPS handles documentation comments with the multiline text widget (discussed in detail in [98]). It supports editing unstructured text, but can also seamlessly embed program nodes such as references to other nodes (such as variables or arguments). Documentation comments are inserted into the program tree primarily via annotations. Because this means that they are attached as a child to a program element,

[27] Traces are pointers from program nodes to requirements to help with quality assurance [109]

```
(genContext, model, operationContext)->void {
  nlist<ForStatement> allFors = model.roots(ForStatement);
  foreach f in allFors {
    node<StatementList> w = <StatementList(statements: [
        # f.iterator,
        WhileStatement(
          body: StatementList(statements: [
              # f.body.statements,
              ExpressionStatement(expr: # f.incr)]),
          condition: # f.condition)])>;
    f.replace with(w);
  }
}
```

Figure 16. A generation script that collects all `for` statements and replaces them with an equivalent `while`. At the core of the loop is an AST builder that creates a new `StatementList` that adds the iterator (e.g., `int i = 0;`) of the `for` as the first statement, and the `while` as the second one. The body of the `while` is comprised of the original body of the `for`, plus the `for`'s incrementor (e.g., `i++`). The condition of the `while` is the condition of the `for` (e.g., `i < arrLength;`.

they are included in cut, copy and paste operations of the documented node. This approach works well, and, in our experience, is better than comments in textual editors, where the relationship between the comment and the commented code is expressed only by conventions (location proximity).

Commenting out code requires storing the commented nodes in the program tree, retaining their structure, surrounding them with comment tokens, painting them grey, ignoring errors inside them, removing potential reference targets from scopes, and ignoring them during generation.

In earlier versions of MPS, addressing these requirements required special support for each particular commentable node. This was a lot of effort and not really practicable, which is why commenting out code was supported only in a very limited way. In MPS 3.3, based on our feedback, Jetbrains added a generic commenting facility. Commented AST subtrees are stored in a special child list[28] of the parent, but rendered (in grey) *in the original location*. This requires special support in the editor framework, which is warranted because it works generically for all languages and all forms of comments. Since this feature has been added, commenting out nodes in MPS has worked well, and generically for any language, without any mbeddr-specific extensions.

[28] The mechanism is the same as for annotations. They are also stored in a special child collection, and the editor is aware of this collection when projecting the tree.

Summary for RQ2, Projectional Editing:

The two main benefits of projectional editing – language modularity and a range of combinable notations – have been used extensively in mbeddr. The anticipated benefits have been observed.

The editor can be flexibly extended with new notational styles with acceptable effort, as exemplified by the support for math, tables and diagrams.

The ability to use multiple and partial projections must be further improved by integrating with other language aspects, in particular, editor actions and type checks.

7.3 RQ3: Mechanisms for Managing Complexity

MPS uses a set of DSLs for implementing language aspects. More generally, the fact that MPS is bootstrapped suggests that developing additional meta-languages is a natural way to manage complexity. In this section we discuss the extent to which this approach worked for mbeddr.

Projectional Editing We consider projectional editing itself an important mechanism for managing complexity. While it introduces some *additional* complexity (in particular, writing editor behavior actions, as discussed in Section 3.3 and [108]), the fact that no disambiguation code [7, 28, 39] between language extensions is required to support extension composition represents a huge reduction in accidental complexity. We would not have been able to develop such a rich and composable ecosystem of C extensions using parser-based language engineering technology; we were unable to find a comparable system (for C or any other language) built using parsers.

Aspect DSLs As explained in Section 3, MPS uses a separate language for specifying each language aspect (structure, notation, type system, transformation, etc.). While these languages all rely on BaseLanguage (Java) expressions and statements as their common core, they are all tailored specifically to their task: the structure language is essentially a declarative meta modeling language, editor definition relies on instantiating, composing and parameterizing editor cells, the type system uses typing equations solved by a solver and the transformations use various kinds of templates to map a source AST to a target AST.

The approach of providing specialized DSLs for each aspect is not unique to MPS (Spoofax [45], for example, uses this approach as well). The alternative (used for example by Rascal [49] and Xtext [11]) is to have a single, powerful language (usually with aspect-specific libraries).[29]

[29] Typically, the language syntax is specified using a grammar, which is an aspect-specific DSL. In addition, some tools provide DSLs for code generation or model transformation. But all other aspects are implemented using the same language.

During the development of mbeddr, we found that using aspect-specific DSLs is an appropriate way to address the complexities in language development. We discuss a few examples below. For textual notations, MPS' editor language approximates the to-be-rendered notation in the editor definition itself, making it easier to understand the editor definition. In addition, as we have discussed, the abstraction of cells can handle textual, tabular, mathematical and diagrammatic notations; we have used all of those in mbeddr. The use of a declarative, rule-based language for expressing type system means that the type system is easily extensible – developers do not have to care about the execution order of typing rules of composed languages. For example, the units extension, which performs computations of physical units in the type system, has been added as a modular set of typing equations requiring no changes to the type system implementation of C and the other C extensions. In contrast, since the order of stacked transformations is semantically relevant, the priority-based scheduling of transformations seems to be the right approach for this particular aspect. So is the use of concrete-syntax templates in code generation rules, because it makes it very easy to understand the structure of the generated code and allows IDE support for the target language.

However, the approach also has drawbacks. It takes time for new developers to learn the various languages (a characteristic of DSLs that is typically defended by stating that a DSL is built for the expert in the domain, not the novice). In addition, every language requires its own debugger. MPS does not always do a good job with debuggers; we elaborate on this issue in the next item.

Debugging Aspect DSLs A drawback of using specific DSLs for each language aspect is that tool support must be built specifically for each of these DSLs. While supplying code completion, error checking and syntax highlighting is essentially free in MPS and is supported for all aspect DSLs, supporting a meaningful debugger is not. This is particularly true if the DSLs stray away from fundamentally imperative execution paradigms. For example, MPS' type system language is based on typing equations and a solver. A debugger for this paradigm must essentially illustrate the solver state, and do this in a way that is useful to the type system developer. MPS has a way to visualize the solver state, but we, as a team, have not figured out how to use it to effectively debug problems in the type system. The debugging experience for type systems is not satisfactory and we have reverted to inspecting types in programs and using console-debugging in the typing rules.

The MPS transformation engine works by applying various kinds of rules to models, stacking transformations on top of each other, resulting in the potentially long transformation chains (up to 40 phases in mbeddr) mentioned before). To debug the transformation of a particular model, MPS provides two mechanisms. First, MPS can show the list of generation phases (also known as the generation plan) as calculated from the pair-wise priorities specified for each involved transformation. This helps to understand which transformations run, and in which order. This is important, because inadvertently running transformations in the wrong order as languages are composed in new combinations is a major source of errors when writing the generators for composable extensions (see also the paragraph on Extension Composition in Section 7.1).

The second mechanism is the ability to (temporarily) store all intermediate models and inspect them. In addition, users can select a node in any of the intermediate models and navigate forward and backward between intermediate models and see which transformation rule created a specific node.

Overall, debugging template-based transformations with the tools provided by MPS works reasonably well. The vast majority of transformations in mbeddr are implemented this way. Transformations that are implemented as a imperative scripts (we have 70 transformation scripts in mbeddr, most of them short and simple) can only be debugged with the limited facilities provided for BaseLanguage discussed below. The same is true for the BaseLanguage code embedded in generator macros.

For the editor, MPS supports outlining cell boundaries with lines and showing the cell structure (as opposed to the AST) of a given editor as a tree. This allows the language developer, to some extent, to track down problems in editor definitions. We have used this facility only during the development of new notations.

Those parts of MPS that are based on Java/BaseLanguage expressions or involve procedural code (behaviors, refactorings or expressions and statement lists in the DSLs discussed above) can be debugged by running MPS inside MPS, and then stepping through the code with MPS' Java debugger. However, this is slow and unwieldy, and we hardly use this approach. Consequently, we mostly used console-debugging using `System.out.println` statements or slightly more fancy alternatives based on custom Java extensions. A final fallback for debugging relies on running MPS from the sources inside IntelliJ, and using IntelliJ's Java debugger to debug MPS. The problem with this approach is that debugging happens on the level of the Java code generated from the aspect DSLs, which can be quite voluminous and initially hard to understand. One of our developers reports: "I prefer the IntelliJ solution. Interestingly, this approach results in a change of the coding style on the DSL level, because I have learned roughly what kind of code is generated and I try to write the aspect code in a way that results in better debuggable code". Clearly, this is not a satisfying solution. The debugging of MPS language definitions must be improved. More generally, debugging DSLs, whether they are used for language definition or not, is an area in language engineering that requires much more research beyond what has already been done [16, 64, 110].

Missing Aspects While MPS comes with a significant number of predefined language aspects and associated DSLs, some aspects are missing. These include language documentation (similar to JavaDoc, but for languages defined in MPS), various mapping specifications (for example, to tree views or graphical visualizations), debugger specification, an interpreter for the language or importers/parsers for legacy code. As a workaround, we have put many of these things into behavior methods, sometimes using custom Java extensions inside these methods to simplify the implementation (for example, of the debugger; see below). However, this is unsatisfactory, because it clutters the behavior with methods for different language aspects. It also requires invasive changes to the concept in order to implement a new aspect-specific interface and the associated behaviors. Finally, it makes it hard to use abstractions that cannot be sensibly expressed as more or less imperative code.

The solution to this problem would be to make the set of language aspects supported by MPS extensible. While language documentation and debugger specification can be argued to be central to language definition and should thus be supported by MPS out of the box, some of the other aspects, such as the interpreter or mappings to various graphical visualizations, are more mbeddr-specific. Based on the experience with mbeddr, Jetbrains has added custom language aspects in MPS 3.3. This way, users can define their own language aspects, together with custom specification languages, and integrate them seamlessly into the MPS language definition infrastructure. Based on this new feature, we are currently in the process of developing a language documentation aspect. We will also move the interpreter definitions (which are already expressed with a DSL) into an aspect.

Declarativeness and Analyses Some of MPS' aspect DSLs aim to be declarative. Examples include the editor definition (instantiate, compose and configure editor cells), the type system (type equations that are solved) and transformations (templates with node replacements). However, the declarativeness is compromised by the ability to inject unrestricted Java expressions and statements into the programs. From the pragmatic perspective of making something work, this is useful. However, from the perspective of analysing the programs written with these DSLs, it is problematic. Some of the potentially useful analyses include: analysing editor definitions to automatically derive grammars/parsers for legacy code import, impact analysis on the typing rules to understand when to re-evaluate which rule, as well as analyzing the transformations to automatically derive a debugger and to verify user-specified properties on the transformations. The debugger issue is especially problematic; we discuss this in the next paragraph.

Analyzing language definitions is also important when certifying mbeddr for use in safety-critical environments [21, 41]. One ingredient to such a certification is a correctness proof of the transformations. In principle, such proofs are feasible, and we are cooperating with a group at McGill university and Fortiss GmbH to prove transformations relevant to mbeddr. However, this relies on reimplementing the transformations with a more restricted, and hence analyzable language [9, 75]. Proving properties on MPS' transformation specifications that mix declarative parts with imperative Java code is not feasible. We are also considering using scope graphs [60] to express scoping rules; as a consequence of their declarative nature, we hope to be able to automatically derive wizards to create reference targets. Finally, we are considering replacing the use of procedural Java code with a more limited, functional expression language that can be analyzed more easily.

Debugger According to the language workbenches survey [32], the development of debuggers for the languages developed with a workbench is not a widespread feature: only 4 of the 10 evaluated workbenches support it, and to various degrees. MPS is one of them. Out of the box, it supports debugging of Java (in its BaseLanguage incarnation), and also allows language developers to reuse the UI infrastructure for DSL-specific debuggers (setting breakpoints, inspecting variables, step in/over/out buttons) for the developed languages.

However, MPS does not specifically support the development of debuggers for languages that support modular language extensions. To address this shortcoming, we have developed our own debugger framework for extensible languages as part of mbeddr [65]. It works by running a debugger (typically gdb in case of mbeddr C) on the generated code. However, the user sees and interacts with the program's MPS representation, including domain-specific extensions. The current statement, breakpoints, the watched variables, and the step in/out/over are shown relative to the program in MPS, not the executing C code. Two ingredients are necessary to make this work: first, it must be possible to trace back from the low-level C code to the original source nodes in the extended MPS program. MPS writes traces during transformation, and the traces can be evaluated backwards. Second, the behavior of the debugger (where to step into, which variables to show and hide in the Watch window) must be specified by the language developer. Using this approach, we have built a debugger that works for the existing mbeddr language extensions, and extension developers can define debugger integration for their own extensions. By using the Eclipse CDT debug bridge [3], the approach even supports on-device debugging (debugging of programs running on an embedded device, connected, for example, through a serial port).

However, there is a problem. The specification of the debugger behavior involves conceptually reversing the transformation that maps the extensions to C, a problem known as origin tracking [91]. Unfortunately, due to the non-declarativeness of the implementation of the transformations discussed in the previous paragraph, this reverse mapping

cannot be obtained by analyzing the transformations[30]. Instead, it is implemented in behavior methods using a custom, embedded DSL. While the DSL significantly reduces the implementation effort, the specification is still fundamentally a duplication of the transformations. Another problem is that a language can have several generators, mapping an extension to different low-level C code structures (for example, to realize different non-functional requirements). In these cases, the debugger specification must also be changed. However, since it is specified in the behavior and not as part of the (exchangeable) transformations, this is not easily possible.

Overall, the situation with building debuggers in MPS (and more generally, in language workbenches) is not yet satisfactory. Further research is required. Our team is looking into debugging mbeddr code at different intermediate transformation steps [64]; we are investigating the development of a custom language aspect for debugger specification or inlining the debugger definition into the transformation rules. More generally, it may be possible to derive transformations and debuggers from a single integrated semantics specification, expressed in a language like DynSem [93] or the K framework [73]. The developers of the K framework have already defined a formal semantics for C [29, 38] and we are currently investigating if and how this could be exploited for mbeddr. There are other attempts at formalizing C, such as the work by Krebbers [52] that might be useful for mbeddr. However, we have not investigated this in any detail.

Self-Extension The development of MPS extensions is outside the scope of the case study itself; however, in this paragraph we briefly discuss such extensions as a means of building mbeddr. They are an important approach for managing the complexity involved in language definition. As mentioned several times before, the languages used by MPS for defining languages are MPS languages themselves. This means that they can be extended with the same mechanisms that are used to extend mbeddr C. We have used this capability extensively and consider it a major ingredient of dealing with complexity in MPS. Here are a few examples:

- Notations: We have already discussed earlier that we have added math, tables and diagrams as new notations by implementing additional editor cells.
- Debugger: In the previous paragraph we have discussed a DSL for defining the behavior of debuggers for C extensions. The DSL is essentially a set of special-purpose statements and expressions used in behavior methods.
- IncQuery: We have integrated the IncQuery graph pattern language [86] as a means for expressing incremental computations for program analyses. This is a new language that is used as part of the behavior models.
- Patterns: This language is primarily used in transformations where it can be used to match a transformation rule against program trees. The language supports nested patterns, where the pattern components can also be referenced from the consequence part of a transformation.
- Importer Test: This language is used to specify test cases for the legacy C code importer. It uses the Pattern language to verify the structure of code imported from textual C files (includi preprocessor directives).
- Counter Example Testing: This DSL is used to assert that the verifier reports the correct counter examples in tests of the mbeddr C model checking infrastructure [59].
- Interpreter: This language is used to quickly and efficiently implement AST interpreters. It also supports interpreter composition, to ensure that interpreters can be extended in the same way as the languages themselves.
- IDE Customization: Several DSLs are used to configure the set of actions shown in the IDE (menus, buttons), for customizing the contents of the palette, as well as for defining new project tree views. All of these languages are used to customize MPS for different user groups.

In addition to these major extensions, we have also developed a number of smaller utilities, including tooltips in the editor, the conditional editor and querylists mentioned earlier, plus a more sophisticated version of `System.out.println` for console debugging.

However, there are also some limitations. The most prominent example is the inability to extend the generator language with additional kinds of transformation rules or generator macros: we would like to introduce new macros that support inserting subtrees matched by the pattern language or capture some data necessary for debugging. These limitations are not conceptual, but rather a consequence of specific low-level implementation decisions in the generator language.

Summary for RQ3, Managing Complexity:

The approach of using a DSL for each language aspect works well based on our experience, even though some aspects are missing and some are not declarative enough to support meaningful analyses.

The support for debugging is spotty: it works well for transformations, but debugging generator macros, behaviors and type system rules is very tedious.

The ability to extend MPS' language definition DSLs with MPS itself is a powerful approach for managing complexity, and we have used it extensively, even though it has some limitations.

[30] The same problem occurs in the context of verification: lifting the low-level counterexamples generated by a verification tool to the level of the DSLs in MPS also requires this reverse mapping.

7.4 RQ4: Performance and Scalability

Performance and scalability is crucial for any tool if it is to be used for real-world applications. The Smart Meter system that was built with mbeddr [107] is non-trivial in size: it has roughly 30,000 lines of code, spread over 428 chunks (which are roughly equivalent to files). As described in Section 6.1, the mbeddr implementation is also significant in size, probably one of the largest systems developed using a language workbench. So, generally, performance and scalability are sufficient for building significantly sized systems. However, there are a number of issues to consider during the design and implementation of a language.

Editor responsiveness For a user's perception of a language workbench's performance, the responsiveness of the editor is of utmost importance: editing operations must work without any noticeable delay. In MPS, editor performance is governed by three ingredients: rendering, code completion as well as type checks and other analyses. We discuss them in turn.

Rendering of the editor involves re-layouting and redrawing the concrete syntax of the program after it has changed as a consequence of an editing gesture. MPS tracks the changes to the AST and then computes the parts of the editor that must be redrawn. Depending on the notations used, and the layout algorithms they employ, the fraction of the editor that has to be redrawn varies. As a consequence, this limits the amount of code that can be edited in one editor window without slowing down the editor too much. The MPS aspect DSLs tend to use a relatively large number of relatively small roots, and we have not run into any performance problems while editing MPS language definition code.

The situation for mbeddr C is different: mbeddr's equivalents of C files can become large, especially if legacy code is imported (it is not untypical to find C files with thousands of lines of code). A recent test showed that up to 4,000 lines of code in one editor/file can be edited with good editor responsiveness.

Code completion is extremely critical to editor performance in MPS. In contrast to a parser-based editor, a projectional editor only lets users type things that are listed in the code completion menu. This means that for every "token" a user types, the code completion menu must be calculated in real time, *even if the user does not explicitly invoke the code completion menu and just enters the code by typing*. For example, if a user wants to instantiate a class using Java's `new` keyword, the list of all visible classes must be put into the code completion menu so that the user can type the name of the to-be-instantiated class. Similarly, when calling a function in mbeddr C, all functions defined in all (potentially transitively) imported modules must be collected before the user can type anything. This illustrates why code completion performance is crucial.

The code completion menu is populated in the following way: MPS checks the used languages for subconcepts of the concept under the cursor position (for example, `Expression`). It then evaluates any existing tree structure constraints to filter the list of applicable concepts (for example, the `EventArgRef` expression may only be used in state transitions that are triggered by an event). If the concepts are smart references, i.e., their only purpose is to reference another node, then the scoping rules are evaluated, and the valid targets are put into the code completion menu (for function calls all visible functions must be collected). While finding the subconcepts of a given concept is framework functionality and aided by caches, the tree constraints and scopes are code written by the language developer. Local constraints (such as resolving the event arguments owned by the event triggering the transition under which we are editing) can be handled efficiently (even many of them), but global lookups (for example, for all functions in all imported modules) can become a performance bottleneck for large systems. While in our practical work with MPS, mbeddr and Smart Meter we did not run into serious problems, we are experimenting with speeding up global lookups by using IncQuery [86], an incremental query engine. It essentially keeps caches of all visible elements, so lookup is cheap. However, it leads to additional memory use because of the necessary caching. The caches are incrementally updated as the model changes.

Type checking in MPS includes two tasks. The first one involves calculating derived types (for example, calculating the type of `+` in `2 + 3.33`, an `int8` and a `float`) and checking type compatibility (for example, verifying that the init expression is compatible with the declared type, as in `int32 x = 2 + 3.33;`). As discussed earlier, this task relies on the type equations and the solver. The second task evaluates checking rules. These are essentially procedurally written code that uses `if` statements to check arbitrary Boolean conditions across the model to detect errors (for example, violations of name uniqueness rules). In addition, checking rules may also invoke other, more advanced analyses such as SMT solvers.

MPS evaluates type system rules in realtime, as the user edits the program. MPS does this incrementally: it tracks which nodes are changed in the editor (using a fine-grained notification mechanism) and reevaluates only the type system rules affected by the edits. For the first task, the actual type calculation and checking, this works reasonably well. But for the second task, where the user potentially writes checking code that traverses large parts of the model, performance can become an issue. The evaluation of these rules is not on the critical path: they do not have to be evaluated before the user can type (in contrast to the constraints used in code completion). In fact, MPS evaluates type system rules in the background.

However, to access the model, the type checker must obtain a read lock on the model repository. While this happens, the editor itself cannot get a write lock, preventing the user

from editing the code. Consequently, the type system unfortunately interferes with the user's editing activities. While MPS abandons long running checking rules when the user continues typing, the type checking required for large programs can become a problem. The performance is also impacted by the number of actual errors that must be annotated to the code. As an extreme case, we once had an imported header file with thousands of errors. The mbeddr C editor was unresponsive for several seconds while MPS added the error markers to the code. If type checking becomes the bottleneck in an editor (as opposed to the rendering process), MPS allows disabling realtime type checking as a last resort.

To address these issues, we use several strategies. First, we are experimenting with integrating IncQuery to incrementalize global checks, as discussed above. Second, we have developed assessments, which are persisted queries over the model with persistent result sets. They are executed only upon user request. We use them instead of checking rules for long-running, global queries, for example to detect code smells [35]. We also allow users to selectively disable checks and analyses they are not interested in at a given time. Finally, the Jetbrains team is currently prototyping a new type system engine that hopefully scales better. They are also looking into avoiding the need for read locks in the type system, so that users can continue typing during the execution of checking rules.

Generation MPS stores programs in models: each model is essentially an XML file that contains program elements expressed in any MPS language. When code generation is initiated, each model is processed separately. The MPS build engine makes sure that, when `make`-ing a project, it only regenerates those models that have changed since the last make (a full rebuild is of course also supported). Consequently, generation is modular/incremental for each model.

Figure 17 shows generation time (vertical axis) over the size of the model (horizontal axis) and the number of languages used (the three lines). We can make the following observations. As the size of the synthesized model increases, the generation time increases roughly linearly: doubling the number of nodes or roots leads to doubling the generation time. If more languages are used, the generation time takes significantly longer. However, the difference between using one extension (state machines) and several (state machines + units + components) is very small. This means that the number of different transformations (and the resulting number of generation phases) is not what slows down the transformations. Instead, it is the growth of the size of the model as the transformations execute (because the lower the abstraction level, the more code is typically needed to express the same behavior): in both cases, the LoC generated is around 200,000 for the largest models; the last intermediate models have roughly 400,000 nodes that must be processed. In the smart meter case study, the generation times for each model were kept under 10 seconds by modularizing the system into an appropriate number of models. While we would appreciate a faster generator, the performance characteristics discussed so far are acceptable and scale to reasonable program sizes.

For large systems, however, models have dependencies: a reference in model `M` may target a node in model `B`. `B` can still be generated modularly. However, when generating `M`, then the intermediate models created from B must be available, otherwise the references in intermediate models break. Currently, MPS does not support this alignment of intermediate models. To work around this problem, our first generation step, when generating some model `M`, is to physically copy in all the elements from all models referenced by `M`, and then generate the compound model. This has several problematic implications, for example, regarding product line engineering and generation of C code with `#ifdefs`. More importantly, however, it means that generation will take significantly longer for all models that have outgoing model dependencies, because generating such a model *always implicitly generates the transitive closure of all dependencies*, voiding the benefits of the modularization into separate models. While we have managed to deal with this problem in Smart Meter by carefully designing the dependencies, it is an impediment to building larger systems. Based on this experience, Jetbrains is currently working on a way to support cross-model generation, where intermediate models are (temporarily) persisted so that models with outgoing references can rely on these persisted models to resolve their dependencies when they are generated.

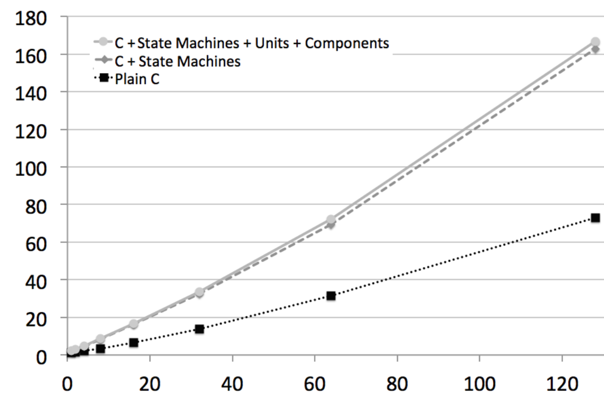

Figure 17. Generation time (in seconds) for a single, synthesized model depending on the number of roots with 1,000 nodes each. The model with 128 roots corresponded to 68,400 LoC of generated C code (for Plain C), 202,000 LoC (C + State Machines) and 198,000 LoC (C + State Machines + Units + Components). The data was measured on a Macbook Pro, 2.7 GHz Core i7, 8GB RAM.

Summary for RQ4, Performance and Scalability:

If attention is paid to the size of roots and the distribution of code over multiple models, then systems of significant size can be built with MPS.

The performance of the type system (as it is evaluated in realtime in the editor) and support for cross-model generation are the two most critical ways of improving MPS performance.

During the *development* of languages we have not run into any problems regarding performance or scalability (of editor, type system or generator definitions).

7.5 RQ5: Interactions with the Development Process?

Automated testing, continuous integration and version control are crucial for any project. In this section we evaluate how those work for MPS language development.

Testing Works The following aspects of a language implementation can be tested: structure and syntax, type checks, scopes and constraints, IDE support, plus the semantics. We discuss all of them in turn.

Testing the *structure and syntax* is not necessary in MPS. In a projectional editor, entering code that is structurally invalid at a given location is impossible in the sense that the editor cannot bind the text and construct the AST; the code is rendered with a red background as shown in Figure 18. No AST is constructed. MPS does not support any way to test this: since the code cannot be entered, one cannot even write a test for this case.

Another aspect of testing structure and syntax is whether users can express all the programs that are required to cover a given domain. Strictly speaking, this is not really testing ("does the program do things right") but validation ("does the program do the right thing"). This is mostly a manual process: a tester tries to write programs that are meaningful in the domain, and if she cannot express a particular program, the language has to be changed. We have done this implicitly when writing the test cases discussed below, and based on feedback from users.

```
for ( int8 i = 0; i < 10

int8 add(int8 x, int8 y) {
  int8[3] a = {1, 2, 3};
  int8[3] b;
  struct
  return x + y;
} add (function)
```

Figure 18. Trying to enter code in a context where it is not allowed leads to unbound (red) code. This example shows the attempt of entering a `for` statement outside a function and trying to define a `struct` inside a function.

```
Test case testSideTransformations
nodes
  ( [ AnImplementationModule   constraints          ] )
                               imports      nothing

    void f() {
      <expr  4 + 3 * 2>;
    } f (function)

test methods
  test testPrecedence {
    assert expr.isInstanceOf(PlusExpression);
    assert expr.left.isInstanceOf(NumberLiteral);
    assert expr.right.isInstanceOf(MultiExpression);
  }
```

Figure 19. This test case asserts that the structure of the expression, after being entered linearly using editor actions, respects the precedence specified for the + and `*` operators.

A final aspect of structure testing involves testing the construction of the AST based on what the user types. For example, if the user enters an expression such as `4 + 3 * 2`, precedence rules must be respected and the resulting tree must correspond to `4 + (3 * 2)` and not `(4 + 3) * 2`. This can be tested by writing a node test case: the tester writes the program and then adds a test method that procedurally inspects the tree and asserts over the structure. Figure 19 shows an example.

Note how the test case in Figure 19 is also a nice example of language composition: the node under test is entered between the large square brackets using the syntax and IDE support of the target language (C in this case). It is marked up using annotations (the `expr` label attached to the expression we are interested in). Below we then write BaseLanguage code to inspect the structure of the C program.

A similar approach based on example code and annotated markup is used for testing type system rules and checking rules. An example is shown in Figure 20. When such a test case is executed in MPS, nodes that have an error message attached (i.e., are marked with a red squiggly line) without having the green `has error` annotation will result in a test failure. Conversely, nodes that have a `has error` annotation but do not actually have an error annotated will also count as a failure. This approach is very convenient to test type systems, checking rules and constraints/scopes (even though in the latter case one has to first disable the constraint to be able to enter the code that violates the constraint).

As Table 2 shows, mbeddr has 324 such tests, with 1,391 assertions (mostly `has error` annotations). It would have been impossible to implement the C type system without this capability. We have regularly used a test-first approach here, where we first wrote the code with the expected error markup, and then implemented or fixed the type system to behave in the expected way.[31]

[31] Note that, because one cannot write code before the language structure and syntax are implemented, MPS does not support test-*first* development

```
Test case testingTypeChecksForLocalVariables
nodes
  ( [ AnImplementationModule    constraints        ] )
                                imports     nothing
    void f() {
      int8 anInt = 42;
      int8 aFloat = <check 3.3 has error>;
      int8 aString = <check "hello" has error>;
    } f (function)
```

Figure 20. Testing type system rules is done by writing code that provokes the error, and then asserting that the error actually occurs.

MPS also supports writing *editor tests* in order to test whether editor actions (side transformations or deletions; see the Actions paragraph in Section 3.3) work correctly. They rely on entering an initial program, a script to describe a user's changes to the initial program using interactions with the IDE (`press Up arrow`, `type "int8"`, `press Ctrl-Space`), and an expected resulting program. When running such tests, MPS starts with the initial structure, executes the scripted user actions on that initial structure, and then validates whether the resulting tree is structurally identical to the resulting node structure specified in the test case. All IDE interactions can be tested this way. In mbeddr, we have decided to use editor test cases only in a few circumstances, where editor behavior is especially complex (for example, parenthesis editing). Using it throughout mbeddr would not have been cost effective in our assessment. Instead, we relied on user feedback when writing the other kinds of test cases to detect problems with IDE behavior. This resulted in a few regressions from time to time, but not enough of them to justify the effort of writing more editor test cases.

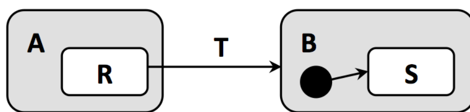

Figure 21. Example state machine with transitions used in the text to explain semantics testing.

Whenever possible, we tested the *semantics* of a program as opposed to its structure. Since mbeddr extensions define their semantics by transformation to C, semantics testing means that we generated the mbeddr program to C and then executed it. As an example, consider the verification of the order of entry, exit and transition actions for hierarchical state machines, as shown in Figure 21. If a transition `T` goes from a state `R` nested in state `A` to a state `S` nested in state `B`, then the order of actions must be as follows: `R.exit, A.exit, T.transition, B.entry, S.entry`. Instead of testing the (quite complex) structure of the generated code, we wrote a test that allocated a buffer of 5 integers all initialized to zero, and then each action added a specific number to that buffer. We then asserted over the contents of this buffer: if the actions were executed in the wrong order, the order of the action-specific numbers in the buffer would be wrong, and the test would have failed.

for language syntax. However, it supports test-*driven* development, where, for every piece of language syntax one defines, one can immediately write a test case. For other kinds of tests (type checks, semantics), a test-first approach is feasible and has been used occasionally by the mbeddr team.

```
statemachine SM initial = S1 {        exported testcase testState {
  in event e()                          SM sm;
  in event f()                          assert(0) sm.isInState(S1);
  state S1 { on e [] -> S2 }            test statemachine sm {
  state S2 { on e [] -> S3 }              e → S2
  state S3 { on e [] -> S1 }              f → S2
}                                         e → S3
                                          e → S1
                                        }
                                      }
```

Figure 22. An example state machine plus a test cases that uses a state machine-specific C extension to express the test for the state machine.

Testing semantics this way requires that it is possible to express expected behavior in the DSL programs. As a consequence, one of the first C extensions we have developed (even before C itself was completely implemented) was the support for unit tests (cf. Figure 7). As the right half of Figure 22 shows, test cases are almost like `void` functions, but support `assert` statements. The number of failed assertions is counted, and the test case as a whole fails if that count is non-zero. Test cases can be executed from the console (where the test count is typically returned as the return value of `main`), or they can be run from within the MPS IDE. We have built an MPS UI extension that enables users to click on an assertion failure message in the MPS console window to navigate to the failed assertion in the code.

For many of the C extensions, we have developed corresponding C extensions to test the original extension; this allows writing tests at the same level of abstraction as the extension itself, thereby improving efficiency of testing. The example in Figure 22 shows the `test statemachine` statement that can be used to concisely test the transition behavior of state machines: each line is an assertion that checks that, after receiving the event specified on the left side of the arrow, the state machine transitions to the state given on the right. Such extensions are not just useful for the language developers to test the transformations; they are also useful for the end user to write tests for a particular state machine. In the former case, one expects the state machine and the test case to both be correct, testing whether the transformations work correctly. In the second case, one expects that the transformations are correct, testing whether the state machine corresponds to the assertions expressed in the test case.

A concern for tests cases where both the test subject and the test case are expressed with mbeddr language extensions is that the generators for both may be compatibly wrong, resulting in erroneously successful tests. However, for any

non-trivial coverage, this is exceedingly unlikely to happen. We have never experienced this problem during the development of mbeddr. We are currently investigating this issue more systematically in the context of using MPS and mbeddr in safety-critical systems.

Relative to semantics testing we have only experienced one significant problem. MPS relies on multi-step transformations. For example, the `test statemachine` statement shown in Figure 22 is translated to a sequence of `trigger/assert` statement pairs, which are then reduced to C code using their own generators. Currently, MPS does not support executing single transformation steps at a time, asserting the structure of the intermediate results. This has been submitted as a feature request for MPS.

A conceptual problem relates to mbeddr's extensibility. Users can compose languages arbitrarily, but, as discussed earlier, we cannot prove that such combination will not lead to unintended semantic interactions (and hence, program failure). If we find out that a particular combination does lead to faulty behavior, we report an error if such an invalid program is written. More generally, testing for the absence of required error messages is hard: in order to write a test for a missing error message, the tester has to think about a particular use of the language. If he does think of it, then he could write the error check; if he forgot to write the checking rule, then it is likely he will not think about writing a test either. To address this issue, we have implemented a test case generator that uses the following algorithm to automatically explore the set of possible programs:

- Select a set of language extensions
- Randomly generate structurally sound programs (structurally unsound programs cannot be written in MPS)
- If the type system finds errors in the program, discard the program (because errors are correctly reported)
- If the program has no errors, try to generate and compile it. If either generation or compilation fails, flag this particular generated program as a failed test.

While successful generation and compilation is not conclusive evidence of the absence of undesired semantic interactions, it is a meaningful proxy. We have found several tens of missing checks using this approach. The details are discussed in [67].

Version Control MPS stores its models in XML files, which cannot be used for diff/merge directly. However, as we have already discussed in Section 7.2, MPS provides a diff/merge facility in the tool that uses the projectional editor to render the code in its natural notation. So, as long as users use MPS for resolving conflicts, the established workflows based on git or other version control systems can be continued unchanged. To reduce the number of merge conflicts, MPS require a dedicated merge driver. These exist for git and SVN, but according to Jetbrains, drivers for version control systems can be built easily.

CI Server Integration Automatically building the languages and test cases, executing type system and semantics tests, and packaging the plugins and the mbeddr IDE can be automated using a continuous integration server. In mbeddr we use Jetbrains Teamcity, mainly because it scales well as a consequence of its support for distribution of build agents over several machines.

MPS supports a headless mode to run tests or generate models. To initiate this, MPS provides ant[32] tasks. These tasks can be integrated with essentially all existing CI servers.

However, it is painful to manually write the necessary ant scripts: assembling the necessary `path` and `classpath` variables, plus establishing references to all necessary language definition artifacts is tedious and error prone. So is packaging the built languages as deployable plugins for the IDEA platform that forms the foundation of MPS. To remedy this problem, MPS ships with a DSL for specifying builds and packaging languages and solutions into plugins. While such a language is generally a good idea, it is not as mature as it should be. For example, the DSL does not report all relevant errors, which means that even though a build script has no error shown by MPS, generation can fail with a low-level error that is hard to track down. Second, these scripts contain substantial duplication relative to project, solution and model properties specified in the IDE. For example, dependencies (on other languages or solutions, as well as on libraries) must be maintained in both places. While the data in the build scripts can be updated based on the data in the properties with one click, this still has to be initiated manually. Forgetting this is a frequent source of build failures. Third, as a consequence of its implementation, the build language is not as extensible as it should be. We have tried to add several extensions that would make an idiomatic mbeddr build easier to express, but found it hard or impossible to write these extensions. Generally, specifying and maintaining the build scripts and CI server infrastructure is still too much work and requires very specific experience. Only two of our team members know in detail how to do it.

Another problem with the ant-based command line build is that it takes very long: during a typical, modularized mbeddr build, several dozen build tasks are started, each of them launching a new (headless) MPS instance. In all, a full rebuild of mbeddr takes around 25 minutes on a modern root server or a developer's notebook. While this is not a problem for nightly builds on the server (we also compensate with more build agents), it is a problem for the command-line rebuilds that are typically run by each developer locally before they check in or after they switch a branch.

Evolution of Models Iterative incremental development implies frequent changes to the languages. This then may

[32] `http://ant.apache.org/`

require changes to existing models.[33] One proven way of avoiding the need for this kind of model migration is to not make incompatible changes; however, this severely limits the flexibility in terms of how languages are allowed to change over time and is generally not desirable.

During the early phases of mbeddr development, the development team had access to all instance models because no external parties used mbeddr; all instance models were in the mbeddr repository. We created a project that contains everything, and developers who made incompatible changes often directly migrated everything, typically through a (structural) global search and replace. Over time, as we got external users and the instance models were not all in our repository anymore, we had to use the following more principled approach when changing a concept `C` in an incompatible way.

1. Implement a new concept `C’`
2. Mark the existing concept `C` as deprecated. The deprecation is achieved through an annotation that stores the timestamp of the deprecation, plus a message that points the user to the new, replacing construct `C’`. The message is reported as a warning in the IDE.
3. If possible, prevent the old construct `C` from being entered, for example, by removing transformations or marking it as `IDontAllowEntering`.
4. If an algorithmic migration is semantically possible, write the necessary transformation that transforms instances of `C` to instances of `C’`. Otherwise, we expect the user to migrate manually, based on the instructions in the `deprecated` annotation.
5. Finally, after a suitable period of time (and possibly after checking existing models for occurrences of `C`), we remove `C` from the language, making sure that no “sediment of old stuff” accretes in the languages.

Writing the migration transformation is typically not a big challenge: the developer who changes the language by definition understands the original structure `C` and the new structure `C’`. The migration script is simply a model transformation written in BaseLanguage, similar to the code that implements an intention, a refactoring or a non-template transformation. What is more challenging is to decide when to execute the migration script.

One approach is to execute the script manually, triggered by the user. While this works for the language developers if they have access to the instance model, it is not realistic to expect end users to do this.

Another approach relies on attaching the migration script to the deprecation warning, letting the user run the transformation by executing the quick fix via `Ctrl-1`. On the plus side, this makes the user aware of the change, and lets the user decide whether to run the scripted migration or manually change the code, maybe in a different way. The problem is that only models that anyone ever opens in an editor are migrated. MPS can also run quick fixes for warnings and errors automatically: as soon as the model is opened in the editor the quick fix is executed, without user interaction. The good thing is that migrations happen automatically. The problem is that the code “changes under the user’s fingertips”, and also that it only happens in the editor, so the model must be opened by the user in the first place.

Since MPS 3.3, based on feedback from the mbeddr team, MPS provides a more robust approach. As soon as a developer creates a migration script in MPS, the language that contains the script (presumably the language that is in the process of changing incompatibly) gets its version number incremented. This version number is stored in all models that use the language. This way, MPS can detect which version of a language has been used to edit a model. When MPS opens a project, it checks this version number for all models: if a version number `n` is found in a model when the version of the deployed language has advanced to `m`, then all migration scripts from `n` to `m` are executed automatically. This way, models are always kept up to date, and they are migrated if and when the user updates his language definitions. The user does not have to open the model in the editor, loading the project is enough to trigger the migration.

The migrations, if implemented wrongly, have the potential to damage models. This is all the more critical if they are run automatically on a client’s computer. To avoid this, the migrations must be tested. Jetbrains is working on a new kind of test case that supports testing migrations. For now, we implement the migration behavior in a regular class and call it both from the migration script and from an intention. This way, a migration developer can try out the migration code on a number of example models before deploying it for automatic execution with customers.

All in all, the need for systematic handling of migrations slows down our development process, because a change to a language must be done more consciously. However, this is not a tool problem; it is a fundamental consequence of mbeddr being used with real end users, and the models being out of reach for the mbeddr team.

[33] In a projectional editor like MPS, only changes to the language *structure* may lead to the need to migrate models; a change in concrete can simply be achieved by changing the projection rules; the new notation is rendered as soon as users open the program with the updated language.

Summary for RQ5, Development Process:

Except for the missing test support for model migrations and single-step transformations, language testing works well, and we have achieved good coverage as demonstrated by a stable code base.

We have successfully integrated mbeddr's build, test and packaging with the Teamcity CI server, but the effort to get there was significant, partially as a consequence of the inadequacy of MPS' build language.

Migrating instance models as the underlying languages change incompatibly is feasible with manually scripted migrations and their automatic execution based on implicitly-maintained language version numbers.

8. Discussion

8.1 Frictions in mbeddr's development

Unsustainable C Improvements We made a few changes to C when implementing it in MPS. For example, users are not required to manually maintain .c and .h files; they just attach an `exported` flag to a struct, function or typedef if they want it to be declared in the header file (and hence potentially visible to other .c files). However, some of these "improvements" were not sustainable because people really needed to use what we considered a strange, or bad feature of C. For example, we had to add back `for`-loops with more than one counter variable. We ascribe our misjudgements to a lack of detailed, real-world C experience of the mbeddr developers, and to the fact that different user communities use C in different ways. Other changes are subtly different from standard C, which may make them dangerous for experienced C developers. For example, we have switched the order of the dimensions in multi-dimensional arrays to make them easier to understand and implement. However, when experienced C developers look at mbeddr code they might be surprised by the code doing something different. We are in the process of changing this back to the standard order.

Reimplementation of Languages Some extensions had to be reimplemented because we did not have any real requirements and/or we did not get them right the first time. An example is the units extension, whose original implementation was not extensible with generic units (see the hump in summer 2014 in Figure 6).

Also, because we got additional requirements over time, the complexity of some languages and generators grew, and we did not do enough refactoring. For example, the generator for the components language must be rewritten completely. It is too convoluted to support the additional language features required of the components language. We do not attribute this issue to specific problems with the MPS generators; it is a case of delayed refactorings because of pressure to implement new features. This happens in many projects, independent of the implementation technology.

8.2 What we Underestimated

Languages are not Enough In addition to the languages and their editor support, effective IDEs also require other kinds of tools, in particular, various tree views and visualizations. While those can be implemented easily in MPS as part of a language's `plugin` aspect, we have underestimated the users' needs for those. We had to retrofit some of them.

Tool Chrome We underestimated the importance of cleaning up MPS' chrome (menu items, dialogs, actions). MPS "looks" complicated to end users. This is partially because it is used for language development (with "real" developers who can cope with dense UIs) and by mbeddr end users (who appreciate simple UIs). The problem was bad enough that prospective users did not "see" the benefits of the languages, notations and extensibility, since they were put off by the UI. Thus we have recently invested significantly into cleaning up the MPS UI. In particular, we have built DSLs for customizing the actions shown in MPS menus, for developing custom structures for the project view and for adding context actions (similar to a diagram palette, see Figure 14).

Education and Adoption We underestimated the resistance of mbeddr's target audience to change to using mbeddr, as well as the importance of integrating with legacy tools (IBM Doors, Eclipse EMF or Microsoft Excel). Also, the initial level of education expected of mbeddr users is quite high. Many embedded developers have a background in electrical engineering or mechanical engineering with lots of domain knowledge about the systems as part of which the embedded software will run – but often only with relatively basic experience in software engineering and the concepts embodied in mbeddr. While these issues are not directly relevant to the experience of *building* mbeddr, they are relevant considerations when deciding to address a domain problem with language engineering.

8.3 Onboarding of Developers

During the mbeddr project we integrated several new developers into the team, which allows us to evaluate, to some degree, the time it takes to become productive with MPS. In our experience, competent developers become productive with MPS in six to eight weeks of coaching by colleagues, and learning on their own. Considering the capabilities of MPS, we think this number is reasonable. However, the learning effort is not distributed equally between the various aspects of MPS. Structure and behavior are understood readily, since they directly resembles OO-style modeling and programming. Editor definition is also understood quickly, once developers get over some off-putting notational details in the editor definition language (such as the use of the percent sign for child collections). New languages developers struggle most with are the type system and the transforma-

tions. The type system language is hard to learn because of its reliance on declarative equations and solvers. Most developers are not familiar with this programming paradigm. The transformation language is hard because of the way templates, scaffolding and macros interact.

On the plus side, it is very easy to find out how to solve specific issues using MPS' languages. Users can always take an existing language, such as MPS' BaseLanguage, and jump to its definition to investigate how a particular language feature is implemented.

8.4 Conceptual Challenges

Optimizing Generators We underestimated the inherent (and MPS-independent) complexity of writing optimizing generators. For example, the components generator optimizes unnecessary indirections through function pointers (if it determines that polymorphism of interfaces is not used in a given system), avoids the use of extra arguments to pass around component state (if it determines that a component only has one instance and thus the state can be held as a global variable) and optionally generates code to treat interfaces as values (so they can be used as values). We were able to modularize some of these optimizations into preprocessing scripts, but others crosscut the templates and thus cannot be modularized. In both cases they are hard to extend when new language constructs are added.

Modularity vs. Optimizations Global optimizations are often necessary in embedded software. An example is global lock ordering in the `ext.concurrency` language: for global lock ordering, the whole system must be generated in one generator run so that all locks are known to the optimizer. Another example is the analysis of components to find out if a component is instantiated more than once, and whether interfaces require polymorphism. If some parts of the system are generated independently, such global analyses are, by definition, not possible. At this point we are not sure how to resolve this issue; we currently opt for compromised modularity in such cases, because, in embedded software, compromising on optimizations is almost always the wrong decision. Nonetheless, the situation is unsatisfactory and requires additional conceptual research in the future.

Open vs. Closed World The open-world assumption underlying mbeddr's extensibility is great for flexibility and productivity, because it allows third parties to add their own language extensions to mbeddr, with IDE and tool support that is as good as that provided for mbeddr's native extensions. However, the open-world assumption is at odds with the requirements for tool qualification and certification [21, 41] in safety-critical contexts, where a tool's behavior has to be known and predictable for quality assurance purposes. This issue is particularly critical for tools that create (generate or configure) part of an executable system – in other words, exactly what mbeddr does with its C extensions. Specifically, the ability to add new languages and generators makes it impossible to know what the generated code will look like when a certain fragment of the program is investigated. Other parts of the program or configuration flags that control the code generators can lead to different generated code. A similar problem occurs for some generic tools in the IDE: for example, to build a call graph for mbeddr, we must consider function calls, triggered events in state machines, component runnables calls, and test-calls. In addition, a new DSL might come with its own "call-like" construct that should become part of a call graph. However, this is impossible to know in advance. One way of alleviating this issue to some degree is to provide interfaces – such as `ICallLike` – that act as markers. However, extension developers have to be aware of interfaces like this one and implement it (correctly) for the new construct to integrate effectively. They also require invasive changes to base languages if they are introduced after the fact. At this point, we do not know how to resolve this issue; more research is required.

8.5 Threats to Internal Validity

A factor that affects the findings in this paper is the bias because of the involvement of the authors in mbeddr itself. The first and second authors are the lead creators of mbeddr, and the third and fourth author are long-time contributors. To counter this bias, we focused on aspects that can be objectively measured (size, concept counts, effort, scalability). Furthermore, the fifth author has no connection to mbeddr or the companies involved in the case study, and was brought in primarily for his experience in conducting qualitative research. Finally, mbeddr and MPS are open source software, so interested parties can look at the code and check many of the conclusions, at least through random samples.

8.6 Threats to Conclusion Validity

Conclusion validity raises the question whether there is an *explanation* for our findings, which are positive overall. Several factors can contribute to this explanation:

First, MPS has been designed to support large ecosystems of languages. Specifically, the support for language composition facilitated by the projectional editor is the raison d'être for MPS in the first place; Sergey Dmitriev's 2004 article on Language Oriented Programming [25] articulates this goal.

Second, MPS has been developed since the early 2000s and Jetbrains estimates that a total of 125 person years has been spent on its development over the 15 years since. Additional effort has been spent validating MPS by developing several MPS-based languages inside Jetbrains (including various languages for web development). This huge effort has obviously helped mature MPS to the point where it is now able to effectively implement systems like mbeddr.

Finally, the key members of the team that developed mbeddr have significant experience with language engineering, primarily based on Xtext. The limitations of Xtext regarding language composition and notational flexibility motivated them to develop mbeddr based on MPS; they had a

clear vision of where they wanted to go, and hence were able to drive mbeddr in the direction described in this paper, fully exploiting the capabilities of MPS.

8.7 External Validity

In this section we discuss a key question: to what extent can the results of this case study be generalized?

Beyond mbeddr As of now, no other case study of the same magnitude has been run, neither by the mbeddr team nor by other groups – this is why we consider the mbeddr case study interesting in the first place. So we have no hard facts about generalizing beyond mbeddr. However, only a few of the findings in this case study are specific to embedded software or the mbeddr languages. In fact, the team at itemis has built several other language ecosystems (in requirements engineering, financial systems, health and security analysis), and, while they are not as big as mbeddr, the team has fundamentally had comparable experiences. We thus think that the findings in this study can be generalized to other language ecosystems developed with MPS.

Beyond MPS mbeddr has been built with MPS, so the findings in this paper apply primarily to MPS-based language engineering. However, there are some other tools that have (some) comparable features. For example, the Intentional Domain Workbench is a projectional editor as well and should support language composition and notational flexibility in a similar way. Similarly, Spoofax and Rascal support similar (but not identical) composition approaches, as a consequence of their use of GLR parsing [84]. Also, as mentioned before, the Spoofax team is working on better and more formalized DSLs for DSL design and implementation [60, 87, 93]. These should further simplify the implementation of language ecosystems like the one discussed in this paper. We would welcome a reimplementation of mbeddr (or a similarly large set of languages) on one of these platforms to compare the findings in this paper.

Beyond the Team At the beginning of the mbeddr project, some team members had experience in language design (see Section 5.1). However, for all of them, building mbeddr was the first attempt at large-scale language ecosystem design. In particular, they had only very limited experience with language composition and the use of multiple notations. MPS experience was also limited. The team learned about both in the course of the project. Therefore we see no reason why other teams could not do the same.

There was significant collaboration between the mbeddr team and the MPS team at Jetbrains; a risk to external validity is that this access to the Jetbrains team may not be available to other teams. However, we think that this risk is limited for several reasons. First, MPS is now much more mature than five years ago when we started. The necessary amount of help from Jetbrains is now lower. The need for support is also reduced because more documentation is available today; beyond the user guide and all the tutorials on the MPS website, several books cover MPS either exclusively [14] or partially [99, 102]. Furthermore, many of the discussions with Jetbrains have been about extending MPS at a fundamental level to enable us to build the mbeddr.platform, which is now available to other teams who want to build something similar to mbeddr. Finally, the MPS team *is* accessible to users through the MPS discussion forum.[34]

8.8 Reliability (Repeatability)

mbeddr is open source. The sources are all available from `http://github.com/mbeddr/`. The version used for this paper can be retrieved from the `buildingMbeddrPaper` tag. The version of MPS used at this time was 3.3.4; it should also work with subsequent 3.3.x releases of MPS. MPS itself can be obtained from `http://jetbrains.com/mps`. This means that all the measurements, size and performance numbers can be reproduced. The use of the MPS language definition DSLs to build mbeddr can also be observed. The mbeddr implementation in MPS can also be seen as a specification for other teams to reimplement mbeddr itself with another language workbench and compare their findings.

9. Related Work

Evaluating DSLs Several papers evaluate a specific (set of) languages, and whether they are useful relative to some metric or better than some alternative. Van den Bos and Storm evaluate Derric, a DSL for digital forensics [88]. They conclude that the DSL supports the development of typical forensics applications with much reduced effort compared to manual coding. A similar conclusion is drawn by Klint and van Rosen in [48] relative to Micro Machinations, a DSL for game design. Our own case study on using mbeddr to develop a smart meter [107] comes to a similar conclusion relative to mbeddr's suitability to develop embedded software. Often the alternative to developing a DSL is to program against an object-oriented framework. Van Deursen compares the Risla DSL (for financial software) against an alternative use of a framework and concludes positively for the DSL [89]. In particular, the paper states that end users can now define questionnaires (the domain of the DSL) and correctness can be asserted more easily. Another paper [90] by Klint and van Deursen on the same language concludes: "a DSL designed for a well-chosen domain and implemented with adequate tools may drastically reduce the costs for building new applications as well as for maintaining existing ones." Kosar et al. also concludes in favor of DSLs [51]: a 15% higher success rate has been measured (regarding achieving the goals that the DSL or framework was designed to achieve). Finally, the fact that well-designed DSLs can increase productivity or quality for a given domain is also cor-

[34] `https://mps-support.jetbrains.com/hc/en-us/community/topics/200363779-MPS`

roborated by systematic studies such as the one by Hermans, Pinzger and van Deursen [40], which used a survey among 18 students to come to the same positive conclusion.

On the one hand, these papers are only marginally related: they all evaluate the *usefulness of a language*, and not its *implementation*, the objective of the current paper. On the other hand, MPS can be seen as a set of DSLs for language implementation, and the current paper can be seen as an evaluation of these DSLs. In this sense, the findings of the current paper and those listed above are similar: well-designed DSLs make the job they are designed for easier.

Language Implementation Case Studies The History of Programming Languages conference series[35] collects invited papers about the design and implementation of programming languages. Some of the papers focus on the history, influences and design decisions, whereas others also touch on implementation issues; however, even those issues are more like implementation decisions. The papers do not cover the actual implementation of structure, parsers, type systems, compilers or IDE support.

One of the earliest dedicated reports on the implementation of a language with (what would now probably be called) a language workbench is Porter's master thesis [66] on the implementation of the editor for the Prototype Systems Design Language based on the Synthesizer Generator [72, 81]. Porter concludes that "the Synthesizer Generator has great power in its ability to transform input into various forms and is quite capable in the areas of consistency checking and verification of conventions."

Visser discusses the design of WebDSL [94], a set of integrated DSLs for web application development. It looks in detail at the requirements for a DSL for web applications, the design decisions taken as well as implementation alternatives. However, it does not critically review the actual implementation of WebDSL with the Spoofax [45] language workbench, evaluating the language workbench itself.

Basten et al. [10] evaluate the implementation of Oberon-0 with the Rascal [49] language workbench, focusing on modularity. They implement the four levels of Oberon-0 as separate language definition modules, each higher level being an extension of the lower level. At the same time, different aspects, such as abstract syntax, concrete syntax and type checks, have been implemented as separate modules as well. The paper concludes that "Rascal is a suitable language for prototyping languages in a modular fashion with relatively little effort. All five tasks across the four language levels have been implemented in under 1500 source lines-of-code". While the Oberon-0 case study is simpler than mbeddr, and Rascal does not rely on a projectional editor and supports only textual notations, the conclusions are similar to those in the current paper.

[35] https://en.wikipedia.org/wiki/History_of_Programming_Languages

Reviewing the examples above, it becomes obvious that in terms of the number of languages, implementation effort, the support for modular extension, and support for different notations, the mbeddr/MPS system discussed in this paper is the largest, making it a relevant case study.

Projectional Editors Early projectional editors include the aforementioned GANDALF [62], the Incremental Programming Environment (IPE) [58] and the Synthesizer Generator [72]. They all interpret the notion of projectional editing slightly differently. For example, GANDALF and the IPE do not try to make projectional editing "feel like text editing" for textual notations, compromising usability. Others, such as the Synthesizer Generator do not use projection at the fine-grained expression level, where textual input and parsing is used. This compromises language composition at the expression level. Contemporary projectional editors include Más/Concrete [2], Clark's prototype [19], the Whole Platform [6] as well as Intentional Programming [23, 77] and its newer cousin, the Intentional Domain Workbench (IDW) [17, 78]. Only the Whole Platform is reported to have been used for real-world projects[36], and the IDW has demonstrated some language composition and notational capabilities similar to MPS. While the Whole Platform has taken part in the Language Workbench Challenge [32], neither one has been evaluated with a case study comparable to mbeddr. It is hard to judge whether either of them could be used to efficiently build something similar to mbeddr.

Alternative Implementation Techniques Many different techniques exist for DSL implementation. The primary distinction is between internal DSLs and external DSLs. Kosar et al. compares several implementation approaches [50], internal and external included, and conclude: "when small groups of users are going to use a new DSL (error reporting is not that important) and when notation should not be strictly obeyed, then the recommended approach is [internal DSLs]. Otherwise, the recommended solution is to implement a full DSL compiler using compiler generators." This advice, as well as our experience, suggests that mbeddr could not have been implemented with anything other than a language workbench, comparable in features to MPS.

We have discussed other projectional language workbenches above; in terms of parser-based language workbenches, only those that support language modularity are candidates for implementing a version of mbeddr without the non-textual notations. This rules out Xtext. Effectively, only Spoofax and Rascal are candidates; Rascal has demonstrated language modularity in the aforementioned paper about Oberon-0 [10] and the modularity of Spoofax language definitions is discussed in [30]. Both the Spoofax and Rascal team have expressed interest in implementing (parts

[36] The developers, Solmi and Persiani, have talked about commercial projects in finance with the authors of this paper.

of) mbeddr to compare the implementations as part of their future work.

Summing up, we found no evidence of implementations of languages similar to mbeddr in terms of modularity and notational diversity, in any of the other language workbenches.

10. Conclusions

Over the last five years we have built mbeddr, a large set of languages and language extensions of C, targeted to embedded software development. mbeddr comprises roughly 88,000 lines of code, and around 10 person years of development effort. mbeddr continues to be used in real world projects and serves as the basis for Siemens' ESD product. This paper describes and critically evaluates the development of mbeddr using the MPS language workbench.

Regarding our research questions, we draw generally positive conclusions regarding the experience of developing mbeddr, even though MPS still has a few problems, and various places for optimization. Specifically, we conclude:

- **Language modularity** works well enough for it to be a benefit with regards to managing the overall complexity; it does not introduce too much accidental complexity for it to be infeasible.
- Although **projectional editing** has some drawbacks regarding editor usability, diff/merge and "simple" things like commenting, the benefits regarding multiple notations and simplified language composition outweigh these challenges by far.
- The approach of using aspect-specific DSLs to manage the **complexity** of implementing DSLs works, even though, because of their declarative nature, debugging some of the programs can be painful. We have built a number of additional DSLs for implementing additional language aspects.
- **Performance and scalability**, as with most other programming and modeling tools, needs care: the sizes of roots (editor tabs) and models (units of generation) must be carefully managed for performance to be acceptable.
- In terms of the **development process** – iterative development, testing or CI server integration – we found no significant limitations.

In terms of impact on language engineering research, this paper is the largest case study on language engineering using a language workbench. To corroborate and challenge our findings, additional studies are needed, both for MPS-based systems as well as for similar systems built with other language workbenches. Furthermore, to better understand long term implications regarding maintainability, longitudinal studies should be set up. We will monitor and report on the continued evolution of mbeddr itself as part of our future work.

In terms of impact on industry, this paper demonstrates that systems of significant size can be developed using language workbenches, and, together with [107], demonstrates that *useful* systems can be built this way. The effort that has gone into the development of MPS – roughly 125 person years – suggests that the effort necessary to develop a language workbench that can be used for systems like mbeddr is significant. However, based on the experience with MPS, some of the findings in this paper, and the improved state of frameworks for developing tools in general, we expect that developing a comparably powerful language workbench today would require much less effort than what went into MPS.

As a consequence of the largely positive experiences with MPS, we have branched out into other domains: we are currently running several projects in the financial, health, security and automotive domains based on MPS and the mbeddr.platform. In addition, we have started a new research project in which we explore the use of this approach in requirements engineering and system specification, aiming at combining informal, semiformal and formal specification languages. We will also continue to monitor the evolution of mbeddr itself to understand the long-term consequences of this approach (in terms of maintainability and evolution) that have not been covered in this paper.

Acknowledgements

We thank all mbeddr contributors for their hard work over the years: Domenik Pavletic, Kolja Dumman, Sascha Lisson, Niko Stotz and Zaur Molotnikov. We are also grateful to the MPS team at Jetbrains, and in particular, Alex Shatalin, for their continued support of our work with MPS. We thank itemis for giving us the freedom to work on MPS and mbeddr for the last 6 years; as well as Bernhard Schätz at fortiss for mentoring the LW-ES research project. We acknowledge Laurence Tratt and Jurgen Vinju for their help with related work. We appreciate the feedback on the paper from Niko, Domenik and Kolja. Finally we thank the SOSYM reviewers: they prompted many important additions to the paper and pointed out lots of grammar and spelling issues, even in the second round of reviewing. Thank you!

References

[1] Code Orchestra IDE. `http://codeorchestra.com/ide`.

[2] Concrete. `http://concrete-editor.org`.

[3] Eclipse CDT. `http://www.eclipse.org/cdt/`.

[4] Jetbrains Meta Programming System. `http://www.Jetbrains.com/mps`, .

[5] Jetbrains MPS Documentation. `https://www.jetbrains.com/mps/documentation`, .

[6] Whole Platform. `http://whole.sourceforge.net`.

[7] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8), 1975.

[8] M. Backes, C. Hriţcu, and T. Tarrach. Automatically verifying typing constraints for a data processing language. In *Certified Programs and Proofs*. Springer, 2011.

[9] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLtrans: A Turing incomplete transformation language. In *Software Language Engineering*, pages 296–305. Springer, 2011.

[10] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, and J. Vinju. Modular language implementation in Rascal – experience report. *Science of Computer Programming*, 114:7–19, 2015.

[11] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[12] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *ACM Sigplan Notices*, volume 45, pages 105–116. ACM, 2010.

[13] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. What is the benefit of a model-based design of embedded software systems in the car industry? In J. Rech and C. Bunse, editors, *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.

[14] F. Campagne. *The MPS Language Workbench: Volume I*, volume 1. Fabien Campagne, 2014.

[15] F. Campagne. *The MPS Language Workbench*. CreateSpace Publishing, 2014.

[16] A. Chiş, T. Gîrba, and O. Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In *International Conference on Software Language Engineering*, pages 102–121. Springer, 2014.

[17] M. Christerson and H. Kolk. Domain Expert DSLs, 2009. talk at QCon London 2009, available at `http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk`.

[18] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NUSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[19] T. Clark. A declarative approach to heterogeneous multi-mode modelling languages. In *Proc of the GEMOC Workshop*, 2014.

[20] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[21] M. Conrad, G. Sandmann, and P. Munier. Software tool qualification according to ISO 26262. Technical report, SAE, 2011.

[22] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering*, pages 422–437. Springer, 2005.

[23] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[24] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[25] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2):1–13, 2004.

[26] B. Dutertre and L. De Moura. The Yices SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[27] T. Dybå, D. I. Sjøberg, and D. S. Cruzes. What works for whom, where, when, and why?: on the role of context in empirical software engineering. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012.

[28] J. Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4(2):183–192, 1975.

[29] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *ACM SIGPLAN Notices*, volume 47, pages 533–544. ACM, 2012.

[30] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *Proc. of OOPSLA 2011*, volume 47, pages 167–176. ACM, 2011.

[31] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of LDTA*, 2012.

[32] S. Erdweg, T. Storm, M. Völter, et al. The state of the art in language workbenches. In M. Erwig, R. Paige, and E. Wyk, editors, *Software Language Engineering*, volume 8225 of *LNCS*. Springer, 2013.

[33] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *European Conference on Object-Oriented Programming*, pages 489–514. Springer, 2014.

[34] M. Fowler. Language workbenches: killer-app for dsls? *ThoughtWorks, http://www.martinfowler.com/ articles/languageWorkbench.html*, 2005.

[35] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[37] J. Gray and G. Karsai. An examination of DSLs for concisely representing model traversals and transformations. In *Proc. of HICSS*, 2003.

[38] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of c. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[39] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN*, 24(11), 1989.

[40] F. Hermans, M. Pinzger, and A. Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009.

[41] J. Hillebrand, P. Reichenpfader, I. Mandic, H. Siegl, and C. Peer. Establishing confidence in the usage of software tools in context of ISO 26262. In *Computer Safety, Reliability, and Security*, pages 257–269. Springer, 2011.

[42] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, (5):279–295, 1997.

[43] M. Jimenez, F. Rosique, P. Sanchez, B. Alvarez, and A. Iborra. Habitation: a domain-specific language for home automation. *Software, IEEE*, 26(4):30–38, 2009.

[44] C. Jones and O. Bonsignour. *The economics of software quality*. Addison-Wesley Professional, 2011.

[45] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*. ACM, 2010.

[46] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proc. ICSE*, 1996.

[47] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2 (2), 1993.

[48] P. Klint and R. Van Rozen. Micro-machinations. In *Software Language Engineering*, pages 36–55. Springer, 2013.

[49] P. Klint, T. van der Storm, and J. Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*. Springer, 2011.

[50] T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[51] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, C. D. Da, and R. P. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[52] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

[53] A. Kuhn, G. Murphy, and C. Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*. Springer, 2012.

[54] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[55] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *Proc. MODELS*, 2014.

[56] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE Softw.*, 26, May 2009.

[57] V. Lussenburg, T. Van Der Storm, J. Vinju, and J. Warmer. Mod4j: a qualitative case study of model-driven software development. In *Model Driven Engineering Languages and Systems*, pages 346–360. Springer, 2010.

[58] R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, 7(5), 1981.

[59] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific C verification with mbeddr. In *Proc. of the 29th ACM/IEEE Intl. Conference on Automated Software Engineering*. ACM, 2014.

[60] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *24th European Symposium on Programming, ESOP 2015*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. ISBN 978-3-662-46668-1. .

[61] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly, 1996.

[62] D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2), 1985.

[63] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008.

[64] D. Pavletic and K. Haßlbauer. Interactive debugging for extensible languages in multi-stage transformation environments. In *2nd International Workshop on Executable Modeling at MODELs 2016*, 2016.

[65] D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer. Extensible debugger framework for extensible languages. In *Reliable Software Technologies–Ada-Europe 2015*, pages 33–49. Springer, 2015.

[66] S. W. Porter. Design of a syntax directed editor for PSDL. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1988.

[67] D. Ratiu and M. Voelter. Automated testing of DSL implementations. In *11th IEEE/ACM International Workshop on Automation of Software Test (AST 2016)*, 2016.

[68] D. Ratiu, M. Voelter, Z. Molotnikov, and B. Schaetz. Implementing modular domain specific languages and analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. ACM, 2012.

[69] D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language engineering as enabler for incrementally defined formal analyses. In *Proceedings of the Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FORMSERA'2012)*, 2012.

[70] D. Ratiu, M. Voelter, B. Kolb, and B. Schaetz. Using language engineering to lift languages and analyses at the domain level. In *Proceedings the 5th NASA Formal Methods Symposium (NFM'13)*, 2013.

[71] D. Ratiu, M. Zeller, and L. Kilian. Safety.Lab: Model-based domain specific tooling for safety argumentation. In *Proceedings of the 3rd International Workshop on Assurance Cases for Software-intensive Systems*, 2015.

[72] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. ACM, 1984.

[73] G. Rosu and T. F. Serbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[74] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case study research in software engineering: Guidelines and examples*. Wiley, 2012.

[75] G. M. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes. Specification and verification of graph-based model transformation properties. In *Graph Transformation*, pages 113–129. Springer, 2014.

[76] M. Simi and F. Campagne. Composable languages for bioinformatics: The NYoSh experiment. *PeerJ PrePrints*, 1:e112v2, 2013.

[77] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[78] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. *SIGPLAN Not.*, 41(10), Oct. 2006.

[79] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[80] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM, 2001.

[81] T. Teitelbaum and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[82] D. Thomas and A. Hunt. Mock objects. *Software, IEEE*, 19(3), 2002.

[83] J.-P. Tolvanen and S. Kelly. MetaEdit+: defining and using integrated domain-specific modeling languages. In *OOPSLA 2009*, OOPSLA '09. ACM, 2009.

[84] M. Tomita. *Generalized LR parsing*. Springer Science & Business Media, 2012.

[85] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

[86] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.

[87] H. van Antwerpen, P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In M. Erwig and T. Rompf, editors, *2016 Workshop on Partial Evaluation and Program Manipulation, PEPM 2016*, pages 49–60. ACM, 2016. .

[88] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 671–680. ACM, 2011.

[89] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. *Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39, 1997.

[90] A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of software maintenance*, 10(2), 1998.

[91] A. Van Deursen, P. Klint, and F. Tip. Origin tracing. *Journal of Symbolic Computation*, 15(5):523–545, 1993.

[92] O. van Rest, G. Wachsmuth, J. R. Steel, J. G. Süß, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *Theory and Practice of Model Transformations*, pages 92–107. Springer, 2013.

[93] V. Vergu, P. Neron, and E. Visser. DynSem: A DSL for dynamic semantics specification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[94] E. Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, LNCS.

[95] E. Visser. Program transformation with Stratego/XT. In *Domain-specific program generation*, pages 216–238. Springer, 2004.

[96] M. Voelter. Embedded software development with projectional language workbenches. In *MODELS 2010*, Lecture Notes in Computer Science. Springer, 2010.

[97] M. Voelter. Language and IDE development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[98] M. Voelter. Integrating prose as first-class citizens with models and code. In *7th International Workshop on Multi-Paradigm Modeling MPM 2013*, 2013.

[99] M. Voelter. *Generic tools, specific languages*. Delft University of Technology, 2014.

[100] M. Voelter and S. Lisson. Supporting diverse notations in MPS' projectional editor. *Proc. of the GEMOC Workshop*, 2014.

[101] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012.

[102] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering*. dslbook.org, 2013.

[103] 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.

[104] M. Voelter, D. Ratiu, and F. Tomassetti. Requirements as first-class citizens: Integrating requirements closely with implementation artifacts. In *ACESMB@MoDELS*, 2013.

[105] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *7th International Conference on Software Language Engineering (SLE)*, 2014.

[106] M. Voelter, B. Kolb, and J. Warmer. Projecting a modular future. *IEEE Software*, Volume 32, Issue 5, 2015.

[107] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using C language extensions for developing embedded software: A case study. In *OOPSLA 2015*, 2015.

[108] M. Voelter, T. Szabo, S. Lisson, B. Kolb, S. Erdweg, and T. Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 9th Conference on Software Language Engineering (SLE)*, 2016.

[109] S. Winkler and J. Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9, 2010.

[110] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374. ACM, 2005.

[111] R. K. Yin. *Case study research: Design and methods*. Sage publications, 2014.

TUD-SERG-2016-025
ISSN 1872-5392