Noname manuscript No.
(will be inserted by the editor)

mbeddr: Instantiating a Language Workbench in the Embedded Software Domain

Markus Voelter · Daniel Ratiu · Bernd Kolb · Bernhard Schaetz

Received: date / Accepted: date

Abstract Tools can boost software developer productivity, but building custom tools is prohibitively expensive, especially for small organizations. For example, embedded programmers often have to use low-level C with limited IDE support, and integrated into an off-the-shelf tool chain in an ad-hoc way.

To address these challenges, we have built mbeddr, an extensible language and IDE for embedded software development based on C. mbeddr is a large-scale instantiation of the Jetbrains MPS language workbench. Exploiting its capabilities for language modularization and composition, projectional editing and multi-stage transformation, mbeddr is an open and modular framework that lets third parties add extensions to C with minimal effort and without invasive changes. End users can combine extensions in programs as needed.

To illustrate the approach, in this paper we discuss mbeddr's support for state machines, components, decision tables, requirements tracing, product line variability and program verification and outline their implementation. We also present our experience with building mbeddr, which shows that relying on language workbenches dramatically reduces the effort of building customized, modular and extensible languages and IDEs to the point where this is affordable by small organizations. Finally, we report on the experience of using mbeddr in a commercial project, which illustrates the benefits to end users.

Keywords Language Workbenches · Domain Specific Languages and Tooling · Synthesis of Tailored Tools · Embedded Systems

Markus Voelter
independent/itemis
E-mail: voelter@acm.org

Bernd Kolb
itemis AG
kolb@itemis.de

Daniel Ratiu; Bernhard Schaetz
Fortiss
{ratiu | schaetz}@fortiss.org

1 Introduction

Tools play an important role in the development of software. Adequate tools support developers in various ways [37,12]. They

- increase productivity through automation of tedious development tasks,
- ensure the well-formedness of the content they produce,
- help verify critical properties of the system and help fix property violations,
- offer support for following a particular development process,
- and guide users by only showing data relevant to the current context.

Domain specific tools enable even better support, since they are aligned with the class of problems relevant in a particular domain. However, developing domain-specific tools is often infeasible because general-purpose tools are not extensible in meaningful ways and developing a tool from scratch is expensive. Justifying the effort may be hard because the tool will only be applicable in a specific domain — the narrower the domain, the harder the justification. Consequently, organizations often fall back on off-the-shelf tools and ad-hoc tool-chains. Often these do not fit the domain very well, and the ad-hoc integration leads to friction losses. *To substantially improve the status quo, it is crucial that the cost for developing domain-specific tools be as low as possible.*

Development tools are often centered around languages used to create content. Hence, building *domain-specific* development tools is closely related to creating *domain-specific* languages. New kinds of views, input forms, or menu items are not enough. Language workbenches are tools that support the efficient development, extension and composition of general-purpose and domain-specific languages and their IDEs. However, reports about their use for building substantially large and practically usable domain-specific tools are rare. In this paper we present our experience with instantiating the JetBrains MPS[1] (Meta Programming System) language workbench for building mbeddr[2], a novel approach to embedded software development that integrates C, domain-specific extensions of C and formal verification (Fig. 1).

Contribution The core contribution of this paper is to demonstrate that language engineering as supported by projectional language workbenches is a solid foundation for building sophisticated domain-specific development tools with reasonable effort. The cornerstone of the approach is *modular* language extension. This means that an extension to a language (including abstract syntax, concrete syntax, type system and IDE services) can be developed without changing the extended language (C in our case). Modular extension also means that several independently developed extensions can be used together in the same program.

To support this argument we report on mbeddr, an industry-strength development environment for embedded software. Based on MPS, mbeddr supports modular extension of C with different domain-specific constructs, offers advanced IDE support for the extensions, and supports formal analyses based

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

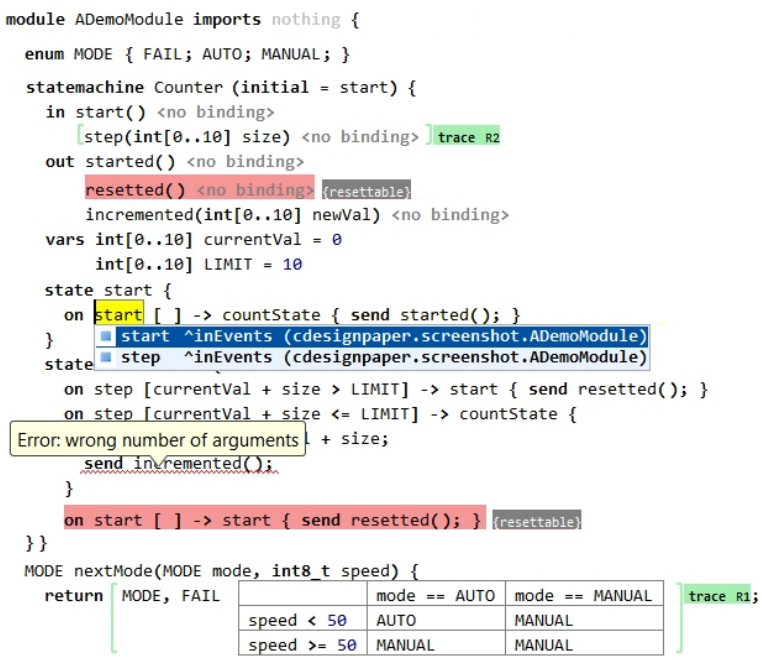

Fig. 1 An mbeddr example program using five separate but integrated languages. It contains a module with an `enum`, a state machine (`Counter`) and a function (`nextMode`) that contains a decision table. Inside state machines and decision tables developers can write regular C code. The IDE provides code completion for all language extensions (see the `start`/`stop` suggestions) as well as static error validation (`Error...` hover). The green `trace` annotations are traces to requirements that can be attached to arbitrary program elements. The red parts with the `{resettable}` next to them are presence conditions: the respective elements are only in a program variant if the configuration feature `resettable` is selected.

on high-level abstractions. In the paper, we illustrate how the MPS language workbench enables this approach based on its comprehensive support for language modularization and extension as well as projectional editing. In particular, we address the following questions:

Q1 **Is it feasible to build sophisticated domain-specific IDEs based on a projectional language workbench?** We are interested in the maturity of the tools and the skills and efforts required for implementing languages and analyses. Only if the efforts are low enough it is feasible to use the approach in the context of industry projects. We investigate this question using MPS as a representative example of projectional language workbenches.

Q2 **Does language modularity really work for realistically complex use cases?** Modular language extension requires syntax, type system, transformation and IDE modularization. We consider *modular* extension (as defined above) essential to keep the overall complexity manageable and to allow third parties to independently build additional extensions on top of the common base language or existing extensions.

Q3 **Does an approach based on language engineering, projectional editing, and formal verification lead to tools that are beneficial for real-world development?** Earlier we emphasized that mbeddr aims to serve as an example of a substantially large and practically usable system built on language engineering with projectional language workbenches. To verify this claim it is essential that real end users find the approach useful to solve real-world problems.

Structure The structure of the paper reflects these questions. In the rest of the introduction we provide a background on language workbenches and embedded software development. In Section 2 we introduce three characteristics of language workbenches we found particularly important for addressing **Q1** and **Q2**: language modularity, projectional editing and multi-stage transformation. To show what we mean by "sophisticated" in **Q1**, Section 3 illustrates mbeddr based on four challenges in embedded software engineering: separating specification and implementation, analysis and verification, requirements tracing and product line variability. We then show how we address these challenges using the three important characteristics to build modular extensions (**Q2**) in Section 4. Section 5 explores the usefulness of the approach from two perspectives: the tool developer-perspective (**Q1**) discusses the experiences made in building mbeddr and the end-user perspective (**Q3**) discusses how end-users benefit from mbeddr. This section is partially based on a real-world application of mbeddr, a project that develops the software for a Smart Meter. We wrap up the paper with the related work (Section 6) and a summary (Section 7).

Previous Work This paper partially overlaps with previous publications. While this paper provides a comprehensive overview over mbeddr and discusses initial real-world experiences, the other publications in contrast address specific aspects of mbeddr. [73] introduces the idea for embedded software development with projectional language workbenches with an early prototype C implementation and a simple example (a Lego robot). In [75] we look at the current mbeddr implementation, focusing purely on language engineering. We show in detail how we have built an extensible version of C and how we integrate domain-specific extensions. [74] defines four approaches for language modularization and composition, and shows how they are implemented with MPS based on a set of simple examples. Finally, [63] introduces our approach for using domain specific formal verification based on language extensions.

1.1 Language Workbenches and MPS

The term language workbench was introduced by Martin Fowler in 2005. In [26] he characterizes a language workbench as follows (slightly paraphrased):

- Users can freely define languages that are fully integrated with each other.
- The primary source of information is a persistent abstract representation.
- A language has three main parts: schema, editor(s), and generator(s).
- Users manipulate a program through a projectional editor.
- A language workbench can persist incomplete or contradictory information in its abstract representation.

The above description explicitly requires projectional editing (discussed below). However, as a consequence of advances in parser technology, the definition can be extended to grammar-based tools. So, essentially, language workbenches are tools that support the efficient definition, composition and use of sets of languages, domain-specific or general-purpose. Fowler's definition also implies IDE support for the defined languages. Although Fowler's article is from 2005, tools that fit his definition have existed for a long time. We discuss some of them in Related Work (Section 6).

mbeddr is based on MPS, which is such a language workbench. It is developed by JetBrains and licensed as open source software under the Apache 2.0 license. MPS uses projectional editing, which is, as we discuss later, an important enabler for its flexibility.

1.2 Embedded Software Development

Although the main contribution of this paper is in language engineering and language workbenches, it is important to show how mbeddr fits into embedded software development. Embedded software is software that is embedded in some kind of mechanical or electronic device, often controlling most of the functionality of that device. Today, embedded software is one of the main innovation drivers and differentiation factors in many kinds of products [17].

Embedded systems are highly diverse, ranging from rather small systems such as refrigerators, vending machines or intelligent sensors over building automation to highly complex and distributed systems such as aerospace or automotive control systems. This diversity is also reflected in the constraints on their respective software development approaches and cost models. For example, flight control software is developed over many years, has a large budget, an expert team and emphasizes safety and reliability. The less sophisticated kinds of embedded systems mentioned above are developed in a few months, often with severe budget constraints and by smaller teams. The requirements for safety and reliability are much less pronounced.

The tools used to develop these systems reflect these differences. Highly safety-critical systems are often developed with tools such as SCADE[3]. Systems that are based on a standardized architecture or middleware, such as AUTOSAR in the automotive domain, are often developed with tools that are specific to the standard (such as Artop[4]). Model-based development and automatic code generation is particularly well suited for systems that are highly structured in terms of state-based behavior or control algorithms. Tools like ASCET-SD[5] or Simulink[6] provide suitable predefined, high-level abstractions (e.g. state machines or data flow diagrams). Using higher-level abstractions leads to more concise programs and simplified fault detection using static analysis and model checking (for example using the Simulink Design Verifier[7]).

[3] http://www.esterel-technologies.com/products/scade-suite/
[4] http://www.artop.org/
[5] http://www.etas.com/
[6] http://www.mathworks.com/products/simulink
[7] http://www.mathworks.com/products/sldesignverifier

However, the state of the practice [20] is that 80% of companies implement embedded software in C, particularly systems that are *not* safety-critical or do *not* implement control algorithms. C is good at expressing low-level algorithms and produces efficient binaries, but its limited support for defining custom abstractions can lead to code that is hard to understand, maintain and extend.

Empirical studies found out that there is a need for tools that are more specific for an application domain yet flexible enough to allow adaptation [29,48]. Domain-specific languages (DSLs) are increasingly used for embedded software [3,32,1]. Studies such as [13] and [48] show that DSLs substantially increase productivity in embedded software development. Examples include Feldspar, [3] a DSL hosted by Haskell for digital signal processing; Hume [32], a DSL for real-time embedded systems as well as [28], which uses DSLs for addressing quality of service concerns in middleware for distributed real-time systems. All these DSLs are *external* DSLs. While they typically generate C code, the DSL program is not syntactically integrated with C. This is useful for some cases, but it is a limitation for others. Extending C to adapt it to a particular problem domain is not new: [59] describes an extension of C for real time applications, [8] proposes an extension for reactive systems, [5] describes an extension for shared memory parallel systems. However, these are all *specific* extensions of C, typically created by invasively changing the C grammar, and they typically do not include IDE support.

mbeddr is fundamentally different. While it builds heavily on domain-specific abstractions, mbeddr is an *open framework* and tool for defining *modular* extensions of C, based on the underlying MPS language workbench. In contrast to essentially all other embedded development tools we are aware of, third parties can use *the same* mechanisms for building their own extensions that were used to implement C and the existing extensions. Third parties are *not* at a disadvantage from having to use limited second-class language extension constructs. This is a fundamental shift in the design of tools and, as Section 5 shows, it has proven very useful in the Smart Meter project.

mbeddr also directly integrates formal analyses based on the domain-specific extensions. Even though formal analysis tools for C programs exist (e. g., deductive verification and abstract interpretation with different plugins for Frama-C[8]; model checkers like SLAM[9] and BLAST[10], or commercial tools such as the Escher C Verifier[11] or Klocwork[12]), they are considered by many practitioners as hard-to-use expert tools and are often avoided. This is because verifying *domain-level* properties (as opposed to low level properties of the code such as read-before-write errors) requires complex code annotations (e. g., Frama-C) or the use of tool-specific property specification languages. It is also difficult to re-interpret the analysis results on the domain level [49]. mbeddr makes formal analyses more accessible by relying on high-

[8] http://frama-c.com
[9] http://research.microsoft.com/en-us/projects/slam/
[10] http://mtc.epfl.ch/software-tools/blast/index-epfl.php
[11] http://www.eschertech.com/products/ecv.php
[12] http://www.klocwork.com/

level, domain-specific language constructs (such as state machines or decision tables), making it easier to specify verification properties and interpret results.

Language extension is much more powerful than the alternatives available to C programmers today. In contrast to libraries, language extensions can lead to low runtime overhead because they are statically translated to C. They also provide extensions to the type system and the IDE. Macros can have similarly low overhead, but extensions of the type system and the IDE are not supported. Also, since macros are low-level text transformations, all kinds of maintainability problems can result from excessive use of macros. Nevertheless, many macro libraries, such as Protothreads [19] (which implements lightweight threads), SynchronousC [33] and PRET-C[66] (both adding constructs for deterministic concurrency and preemption), are good candidates for abstractions that could be reified as language extensions based on mbeddr.

In private communication with the authors, a potential user from a major vendor of heating systems told us that he likes mbeddr because it is right in the middle between programming in C and high-level, rigid modeling tools, with the added benefit of extensibility. He argued that mbeddr lets him add language-level support for the abstractions relevant for his platform ("I can build my own AUTOSAR-like infrastructure for the heating systems domain with very little effort"). This is a nice summary of how mbeddr is to be used.

2 Important Characteristics of Language Workbenches

Based on previous experience with MPS and other language workbenches we found that three characteristics are especially important for language workbenches. Building mbeddr, with its substantial size and complexity, has confirmed our beliefs. The characteristics are: language modularity and composition (implied by Fowler's first item, and the basis of **Q2**), projectional editing (implied by Fowler's second item and important for **Q1**) and multi-stage transformation (also implied by Fowler's first item and relevant for **Q2**). In this section we discuss these characteristics. Section 4 illustrates how these characteristics are supported by MPS, and how they can be used to implement a set of language extensions described in Section 3. In our Related Work (Section 6) we discuss if and how other language workbenches support these characteristics.

2.1 Language Modularity

Classical tool platforms such as Eclipse[13] support modular tool extension based on plugins. In this paper, however, we emphasize the need for *language* extension in addition to *tool* extension. Many tools essentially provide a user interface for editing and processing data expressed with one or more formalisms or languages. To be able to extend the tool in meaningful ways, extensions to the underlying formalisms and languages are necessary as well (**Q2**).

[13] `http://eclipse.org`

In contrast to general purpose programming languages (GPLs) such as C, DSLs address much narrower domains. By encoding knowledge about the domain in the language and its associated analyzers, generators or interpreters, a DSL is much more *effective* for expressing programs for that particular domain. This effectiveness results in shorter programs, tighter integration of stakeholders into the development process and increased suitability of the code for formal analyses and automatic transformation. Pervasive use of DSLs for real-world systems requires the set of concerns of the system to be described with a set of languages, each optimized for a particular concern. Some of these languages may be general purpose, some may be domain-specific. To implement the whole system, this set of languages has to work together (**Q2**). Traditionally, languages have been combined by using an approach called *referencing* in [74]. There, two separate programs are written with separate languages. Each program resides in its own file, and cross-references are established with by-name references. An IDE may check the two programs for referential integrity. Referencing is useful if the two concerns addressed by the two languages *should* be separated into two different viewpoints, each of them expressed with its own language, and possibly described by different stakeholders. However, as we will see in Section 3, many concerns benefit from a tighter integration that includes *syntactic composition*. In [74], we identify the following composition approaches that provide syntactic integration:

Extension: If a language l_2 extends a language l_1, l_1's concepts can be used in the definition of l_2 (we use the term concept to refer to the elements of programming languages, comprising abstract syntax, concrete syntax and semantics). l_2 has a dependency on l_1 and cannot exist without l_1. A program written in l_2 may embed instances of l_1's concepts. An example would be the extension of the C programming language with state machines (Section 3.4), which adds new top-level concepts, types, expressions and statements to C.

Embedding: As in Extension, programs can use a mix of l_1 and l_2 syntax, but the two languages remain independent, increasing their potential for reuse. Embedding state machines into components is an example we discuss at the end of Section 4.1. Another maybe more obvious example would be embedding SQL in Java: since SQL would have no dependency on Java, it could also be embedded into C. Often a third language will be used to adapt l_2 to l_1 syntactically and semantically.

MPS supports both *modular* language Extension and Embedding: in both cases, no invasive changes to the participating languages are required. End-user can use several independently developed extensions in a single program as needed. This modularity is a critical building block for successfully building extensible domain-specific IDEs. **Q2** addresses this challenge.

2.2 Projectional Editing

Traditionally, text editors are used to enter character sequences that represent programs. Based on a grammar, a parser then checks the text for syntactic correctness and constructs an abstract syntax tree (AST) from the character

sequence. The AST contains all the data expressed by the text, but ignores notational details. It is the basis for all downstream analysis and processing.

Projectional editing does not rely on parsers. As a user edits a program, the AST is modified *directly*. A projection engine then creates some representation of the AST with which the user interacts, and which reflects the resulting changes (Fig. 2). This approach is well-known from graphical editors: when editing a UML diagram, users do not draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates an instance of `uml.Class` as a user drags a class from the palette onto the canvas. A projection engine renders the diagram by drawing a rectangle for the class. This approach can be generalized to work with any notation, including textual.

Fig. 2 Left: In parser-based systems a user sees and manipulates the concrete textual syntax of a program. A parser then (re-)constructs the AST from the text (going from the AST to the concrete syntax requires extra care to retain formatting and is not supported out of the box by most parsers – hence the dotted line). **Right:** In projectional editing, while the user still *sees* a concrete syntax, each editing gesture *directly* changes the AST. No parser is involved and it is not necessary to be able to (re-)construct the AST from a flat text structure. Instead, the concrete syntax is projected from the AST.

In projectional editors, every program element is stored as a node with a unique ID (UID). References between program elements are stored as pointers based on the UID, so the AST is actually a graph. These references are established during program editing by directly selecting reference targets from the code completion menu. This is in contrast to parser-based environments where a reference is expressed as a string in the source text, and a separate name resolution phase resolves the target AST element. In a projectional editor, programs are stored using a generic tree persistence format (such as XML).

What makes projectional editing interesting for language workbenches in general, and for mbeddr in particular, are the following two characteristics. First, the approach can deal with arbitrary syntactic forms including textual, symbolic/mathematical, tabular and graphical[14]. This means that much richer notations can be used *in an integrated fashion*, improving the usability to the end user (**Q3**). The decision tables discussed in this paper are an example. Traditionally the tools for building textual and tabular/symbolic/graphical editors were very different in terms of their architecture and user experience, and integrating them seamlessly was a lot of work, and sometimes impossible.

Second, when independent languages are composed (either in case of Embedding, or in case several independent Extensions of the same base language are used in a single program) the resulting composite language is never syntactically ambiguous (which helps with **Q2**). This is in contrast to mainstream parser-based systems that rely on a limited grammar class such as LR or LL(k),

[14] MPS does not yet support graphical syntax, but will in 2013. Other projectional editors, such as Intentional's Domain Workbench [68] support graphical notations already.

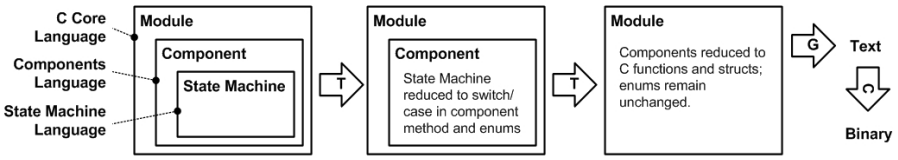

Fig. 3 Higher-level abstractions such as state machines or components are transformed (`T`) to their lower-level equivalent. From the C program we generate (`G`) C text that is subsequently compiled (`C`).

where such compositions are often ambiguous and require invasive change to the composite grammar to resolve the ambiguities (we discuss the capabilities of parser-based systems in Section 6, Related Work).

In principle, projectional editing is simpler than parsing, since there is no need to "extract" the program structure from a flat textual source. The challenge for projectional editors lies in making them convenient to use for end users (**Q3**). Traditionally, projectional editors have had a bad reputation because users had to construct the syntax tree more or less manually instead of "just typing". MPS has solved this problem to a large extent, the editing experience is comparable to traditional text editors. We discuss some of the strategies how MPS addresses the problem in Section 4.2, and look at some of the remaining usability challenges in Section 5.3.

2.3 Multi-Stage Transformation

A transformation maps one program tree or graph on another one. In the context of processing programs expressed with DSLs, the languages used to express these two graphs will usually be different: a more high-level and domain-specific language is mapped to a more general one[15] (Fig. 3).

However, for modular language extension and composition (**Q2**), transformations have to be composable as well[16]. In particular, it must be possible to chain transformations, where the result of one transformation acts as the input to another one. To avoid unnecessary overhead in this case, intermediate transformations should be AST-to-AST mappings. Only if a subsequent tool requires textual input (for example, an analysis tool or a compiler) should textual output be generated. To make a set of transformations extensible, the following features, both supported by MPS, are required:

- Several transformations for the same model have to be supported, either executed in parallel (creating several products from a single input, for example, configuration files and visualizations) or alternatively (creating different, alternative products from a given input, for example, for realizing different non-functional characteristics of an extension; the static component connections in Section 3.3.1 are an example).

[15] Refactoring transformations work with a single language, and in reverse engineering, transformations go from lower to higher levels of abstraction. However, those two cases are outside the scope of this paper.

[16] The static semantics also have to be composed. As discussed in [74], MPS supports the modular definition of type system rules. We provide examples throughout Section 4.

– Dependencies between transformations must be specified in a relative way, and the transformation engine must compute a global transformation sequence based on the transformations configured for a particular program. This supports plugging in additional transformations into the chain without invasive modification of other transformations (see the transformation of mocks at the end of Section 4.3).

Many transformation engines do not support the second item. While they provide languages to express a single transformation, they often do not address the extensible composition of several transformations. Eclipse Xtend[17] is an example that does not explicitly address composition. In contrast, Stratego [9] provides higher-order functions to orchestrate transformations.

3 Example mbeddr C Extensions Relevant to Embedded Software

Question **Q1** asks whether it is feasible to build a sophisticated domain-specific IDE based on language engineering and MPS. In this section we illustrate what we mean by "sophisticated" based on examples from embedded software. We first provide an overview over the mbeddr stack (Section 3.1), then briefly discuss mbeddr's implementation of C (Section 3.2) and finally look at four embedded software engineering challenges and their solution in mbeddr: separating specification and implementation, analysis and verification, requirements tracing and product line variability (see sections 3.3 through 3.6). For each challenge we first describe the challenge, then show how modular language extension addresses the challenge (**Q2**) and wrap up with a brief look of how specific features of language workbenches (from Section 2) enable the implementation. We will provide details of the implementation in Section 4.

3.1 Overview over the mbeddr Stack

As Fig. 4 shows, mbeddr can be seen as a matrix. On the horizontal axis it is separated into an implementation concern (left) and an analysis concern (right). On the vertical axis it consists of a number of layers.

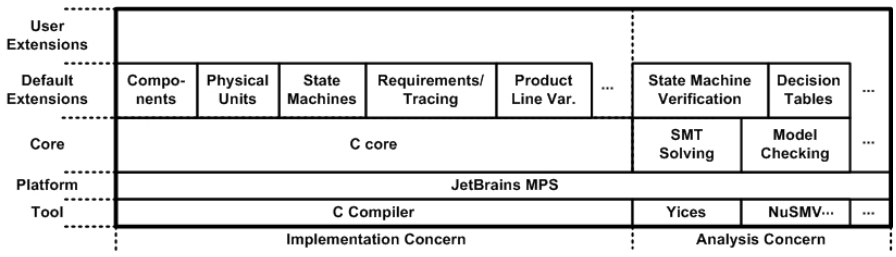

Fig. 4 The mbeddr stack at a glance. Details are explained in the running text (Section 3.1).

At the center is the MPS language workbench. On top of MPS, mbeddr ships with a number of core languages. On the implementation side the core language

[17] `http://eclipse.org/xtext`

is C. On the analysis side, the core comprises languages that represent different analysis formalisms, currently SMT (satisfiability modulo theories) solving [67] and model checking [14]. The next layer up consists of default extensions. On the implementation side mbeddr ships C extensions for interfaces and components, physical units, state machines plus various smaller ones, such as decision tables (an example is at the bottom of Fig. 1). These build on top of C and also translate back to C during generation. On the analysis side the default extensions include support for model checking state machines and for checking the completeness and determinism of decision tables. Below the common platform JetBrains MPS, mbeddr integrates existing tools: a C compiler for the implementation side (`gcc` by default, but it can be exchanged), as well as the NuSMV[18] model checker and Yices[19] and CVC[20] SMT solvers. On top of the default extensions, users can develop their own application level DSLs. These typically rely on the core and default extensions either by directly extending (and translating back to) languages from those layers or by embedding subsets of the languages from these layers into new application level DSLs. We describe the Smart Meter user-level extensions in Section 5.

3.2 mbeddr's Version of C

In order to be able to build extensions for C we first had to implement C in MPS. This entails the definition of the abstract syntax (called *structure* in MPS), the concrete syntax (called *editor*), a type system and a tree-to-text generator (the `G` in Fig. 3) so the code can be piped into existing compilers. While we implemented essentially all of C99 (it is supported by the vast majority of embedded C compilers), we did change some aspects. Some of the changes are a first step to providing a more robust C, one of the goals of the mbeddr project. Other changes were implemented because it is more convenient to the user or because it simplified the implementation of C in MPS. None of these changes constitutes a significant change in C's expressiveness. We provide an overview over all changes and the rationales for them in [75]. We mention the most important ones below.

mbeddr C provides *modules* to act as namespaces. A module contains the top level C constructs (such as `struct`s, functions or variables). These module contents can be `exported`. Modules can *import* other modules, in which case a module can access the exported contents of its imported modules. While header files are generated in the end, we do not expose them to the user: compared to header files, modules provide a more convenient means of controlling which program elements are visible to others. Another use case for headers is the separation of specification (header) from implementation (C file). mbeddr's components provide first-class support for this (see Section 3.3).

More generally, mbeddr C does not support the *preprocessor*. Instead, mbeddr provides first class support for its most important use cases. Exam-

[18] `http://nusmv.fbk.eu`
[19] `http://yices.csl.sri.com`
[20] `http://www.cs.nyu.edu/acsys/cvc3`

ples include the modules mentioned above (replacing `#include`) as well as the first-class support for variability discussed in Section 3.6 (replacing `#ifdef`s). Since the preprocessor is often used to emulate missing features of C in an ad-hoc way, removing it goes a long way in creating more maintainable and more analyzable programs (cf. the empirical study in [24]). The same is true for introducing a native `boolean` type and not interpreting `int` as `boolean`s by default. A cast operator is available for interoperability with legacy code.

3.3 Separating Specification from Implementation

The modules discussed above enable basic program modularization, visibility control and namespaces. However, they do not support the separation of specification from implementation. A specification specifies clearly the service provided to a client, independent of any particular (conforming) implementation, also supporting different implementations of the same specification. Object-oriented and component-based programming exploit this notion for program code. However, C does not support this feature beyond separating sets of functions, `enum`s, `typedef`s etc. into different `.c` and `.h` files.

3.3.1 mbeddr's Solution

mbeddr, in contrast, supports a rich component model including interfaces, components, instantiation, connectors and special support for testing components based on mocks [70].

Interfaces An interface represents a specification and is essentially a set of operation signatures, similar to function prototypes in C:

```
exported interface DriveTrain {
  void driveForwardFor(uint8 speed, uint32 ms)
  void driveContinouslyForward(uint8 speed)
  uint8 currentSpeed() }
```

Components Components represent the implementation. They have *ports*, where each port refers to an interface. A *provided* port declares that the component implements the provided interface's operations, and clients can invoke them. These invocations happen via *required* ports which express an expectation of a component to be able to call operations on the port's interface. The code below shows a component `RobotChassis` that provides the `DriveTrain` interface shown above and requires two instances of `EcRobot_Motor`. The runnable (component method) `dt_driveForwardFor` is triggered by the `driveForwardFor` operation from the `dt` port. Note how regular C code is used together with component-specific extensions for calling operations on a port:

```
exported component RobotChassis {
   provides DriveTrain dt
   requires EcRobot_Motor motorLeft
   requires EcRobot_Motor motorRight

   void dt_driveForwardFor(uint8 speed, uint32 ms) <- op dt.driveForwardFor {
     motorLeft.set_speed(((int8) speed));
     motorRight.set_speed(((int8) speed)); } }
```

Like C++ classes, mbeddr components support polymorphic invocations: a required port only specifies an *interface*, not the implementing *component*. This way, different implementations can be connected to the same required port. This is implemented via a function pointer in the generated C code. However, to optimize performance (an ever present requirement in embedded software), the generator can be configured to connect instances statically. In this case, an invocation on a required port is implemented as a direct function call, avoiding the function pointer overhead. Polymorphism is not supported in this case — users trade flexibility for performance.

Instantiation Components can be instantiated. Each component instance must have all its required ports connected to provided ports of other instances that provide the same interface as the required port. This is an example of how linguistic abstraction — making ports first class entities — improves analyzability, a major goal of mbeddr. Another difference of mbeddr components compared to C++ classes is that mbeddr component instances are assumed to be allocated and connected during program startup (embedded software typically allocates all memory at program startup to avoid non-determinism with respect to timing and memory management problems during execution). The following piece of code shows an instance configuration. It defines two instances of `EcRobot_Motor_Impl` (each with a different value for its `motorAddress` configuration parameter) as well as a single instance of `RobotChassis`. The `chassis`' required ports are connected to the provided ports of the two motors.

```
exported instances robotInstances {
  instance RobotChassis chassis
  instance EcRobot_Motor_Impl motorLeft(motorAddress = NXT_PORT_B)
  instance EcRobot_Motor_Impl motorRight(motorAddress = NXT_PORT_C)
  connect chassis.motorLeft to motorLeft.motor
  connect chassis.motorRight to motorRight.motor }
```

Contracts mbeddr interfaces support contracts in the form of pre- and post-conditions (inspired by Eiffel [56]) and sequencing constraints based on protocol state machines. Below is the interface from above with contracts:

```
exported interface DriveTrain {
  void driveForwardFor(uint8 speed, uint32 ms)
    pre(0) speed < 100
    post(1) currentSpeed() == 0
    protocol init -> init
  void driveContinouslyForward(uint8 speed)
    post(1) currentSpeed() == speed
    protocol init -> forward
  void accelerateBy(uint8 speed)
    post(1) currentSpeed() == old(currentSpeed()) + speed
    protocol forward -> forward
  query uint8 currentSpeed() }
```

The `driveForwardFor` operation requires the `speed` parameter to be below 100. After the operation finishes, `currentSpeed` will be zero (note how `currentSpeed()` is marked as `query`, which means that it is idempotent and hence can be invoked any number of times without side effects). The protocol specifies that, in order to call the operation, the protocol has to be in the `init` state (the states are declared implicitly as they are used in the `protocol` specifica-

tion). The post condition for `driveContinouslyForward` expresses that after executing this method the current speed will be the one passed into the operation — in other words, the robot keeps driving. This is also reflected by the protocol which expresses that it will be in the `forward` state after executing the operation. The `accelerateBy` operation can only be called legally while the protocol is in the `forward` state, and it remains in this state. The post condition shows how the value returned by a `currentSpeed()` *before* the execution of the function can be accessed. Pre- and post-conditions reuse regular C expressions and provide additional ones (for example, the `old(..)` expression or the `result` expression to access the return value of non-`void` operations).

Contracts are specified on the *interface*, but the code that checks the contract is generated into the components (i.e. the implementations of the interface operations). The contracts are then checked at runtime. We plan to use C-level model checkers to statically check whether the contracts are fulfilled.

3.3.2 Implementation Characteristics

Components and interfaces rely on modular language extension (**Q2**), they blend in directly with C: interfaces and components live in modules alongside functions or `struct` declarations; and regular C statements and expressions can be used to implement components or specify contracts for interfaces. They also use multi-stage transformation: components are translated to C functions and `struct`s, which are then transformed to text. State machines embedded in components are first transformed to regular component implementations, those are then transformed to C, which is finally transformed to text (see Fig. 3). Components also makes use of multiple *alternative* transformations of the same model: the optional static connection of ports implements the same semantics with different non-functional properties.

3.4 Analysis and Verification

Formal verification of embedded software has traditionally been driven by the safety-critical systems community where system failure can cause considerable damage or hurt people. Safety-critical software must comply with various certification standards that require formal verification to ensure the absence of such defects. Many successes of using formal verification have been reported, but these are isolated, involve experts, big budgets and products that have a relatively long development cycle [60]. Verification can also be useful for non-safety-critical embedded software, since, once deployed in the field, embedded software is often hard to debug and fix: the device may have limited I/O capabilities or may be hard to access physically. In this case, verification can be used pragmatically to find bugs. [60, 49, 16] describe several reasons why verification is not used by mainstream developers. The most important are:

1. Verifying programs written in general purpose languages is expensive because of the low level of abstraction. Users of verification tools feel the accidental complexity along four directions. First, expressing domain-level

properties in terms of low-level program code is challenging. Second, interpreting the low-level verification results at the domain level again requires bridging the abstraction gap, this time in the opposite direction. Third, implementation details irrelevant to the verification itself may be outside the language subset supported by the verification tool, preventing the verification tool from starting up (e. g., pointer casts are unsupported features of the Frama-C Jessie plugin [52] and prevent Jessie from starting). Fourth, verification tools for GPLs are rather slow and hard to use in an agile way. They may also encounter state space explosions or run out of memory.
2. Describing the system and the to-be-verified properties in the specification language of a dedicated verification tool (e. g., model checkers) eliminates the noise introduced by GPLs and makes the verification process more efficient. However, there is a big semantic gap between the domain knowledge and the input languages of verification tools. Translation of application domain problems into the specification language is tedious and error prone, and the interpretation of the tool's output is hard.

Due to these reasons there is a perception among many practitioners that formal methods are only for experts, requiring the use of sophisticated tools and languages. Many practitioners shy away simply because of this perception.

3.4.1 mbeddr's Solution

In mbeddr we rely on automatically generating the input to the verification tool from higher-level and domain-specific abstractions and reinterpreting the results in the context of these higher-level abstractions. We use language engineering to enable this approach: the higher-level abstractions are incrementally added to and tightly integrated with C programs, avoiding the need to work with different tools for different abstractions or analyses. We also locally restrict parts of C to make it easier to analyze. We integrate the verification tools directly into the IDE, hiding the tool complexities from the user [63]. In this section we discuss in detail two examples of analysis and verification: decision tables and state machines. We mention a third one in Section 3.6.

```
enum MODE {AUTO; MANUAL; ERROR;}
MODE nextMode(uint16 speed, MODE mode) {
  return MODE, ERROR
```

	speed < 30	speed > 20
mode == MANUAL	AUTO	MANUAL
mode == AUTO	MANUAL	MANUAL

```
;}
```

Fig. 5 A decision table is an expression that evaluates to that value whose column header and row header are `true`. They are essentially generated into nested `if` statements (column headers will be the outer `if`s, evaluated first).

Decision Tables Fig. 5 shows an example decision table [39]. It determines a new `MODE` based on the current value of the `mode` and `speed` arguments. The table also specifies a default value `ERROR` that is used in case none of the cases expressed by the table actually fits the input data. For a two-dimensional decision table, there are two obvious possible analyses:

- Completeness requires that every behavior of the system is explicitly considered and no case is omitted: this requires listing all possible combinations of the input conditions in the table (avoiding the need for a default value).
- Determinism checks that there are no sets of input values for which several cases are applicable

We perform these analyses for decision tables that are marked as `verifiable`. As long as the conditions in the table only use logical and linear arithmetic expressions, these analyses can be reduced to SMT problems (we check for linearity statically in the IDE and report an error if the condition expressions in a `verifiable` table are not linear). A table with n rows (r_i) and m columns (c_j), can be checked for completeness by checking the satisfiability of the following formula (if satisfiable, the table is incomplete):

$$\neg \bigvee_{i,j=1}^{n,m} (r_i \wedge c_j)$$

Similarly, the determinism of decision tables can be expressed by using the following formula. If it is satisfiable, a non-determinism was found.

$$\bigvee_{i,j=1}^{n,m} \bigvee_{k=i+1,l=j+1}^{n,m} (r_i \wedge c_j \wedge r_k \wedge c_l)$$

For `verifiable` tables, the input for the solver is generated from the table and the integrated Yices SMT solver[21] is run. The result of running the solver is evidence for a satisfiable formula. This evidence is lifted and interpreted with respect to the analyzed decision table, providing the user with the data necessary to understand the combinations of input data that lead to problems with regard to the analyses. The code below shows the result of running the completeness and determinism analyses on the decision table from Fig. 5.

```
FAIL: incomplete. example: speed = 21, prev = ERROR
FAIL: non-determinism betw. cells (1,1) and (1,2). example: speed = 21, prev = MANUAL
FAIL: non-determinism betw. cells (2,1) and (2,2). example: speed = 21, prev = AUTO
```

State Machines The second example for integrated verification concerns proving properties of state machines using model checking [15]. The approach is similar to the one used in decision tables: from a state machine marked as `verifiable` we generate the input to the NuSMV model checker[22], run it, and show the results in the IDE (Fig. 6). As with decision tables, verifiable state machines are limited to make model checking straightforward. For example, a local variable cannot be read and written during a single transition, and all integral types are bounded. We check a set of default properties for every state machine including unreachable states, transitions that cannot be fired, determinism and whether local variables remain inside the specified bounds. In addition, we have implemented a set of well known verification patterns[23] which mbeddr users can use to express custom verification conditions.

[21] `http://yices.csl.sri.com`

[22] `http://nusmv.fbk.eu/`

[23] `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

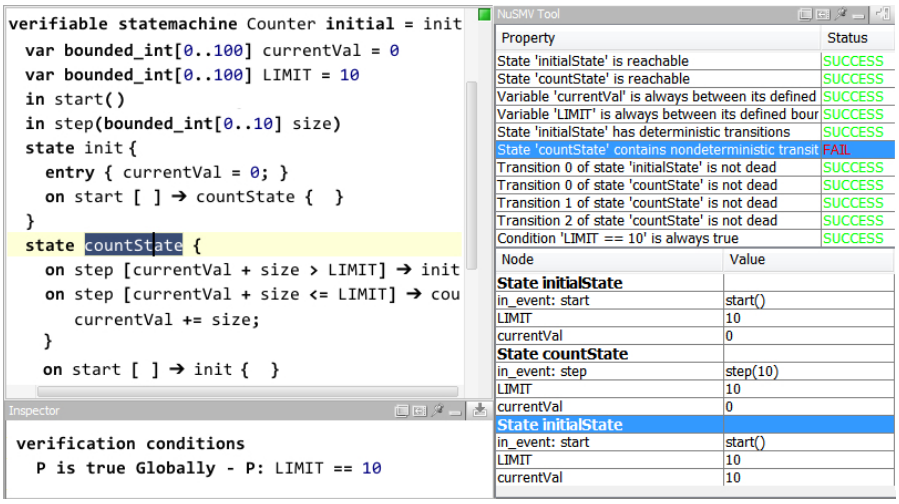

Fig. 6 A `verifiable` state machine has a context menu action that runs the model checker, the results are reported back directly in the IDE. Clicking on a state in the counter example for a failed property (lower part of the table), focuses that particular state in the editor.

In Fig. 6 the checker reported an error: the two `step` transitions in `countState` are non-deterministic for `currentVal + size == LIMIT`. We show the counter example in the lower part of the table. Custom properties are specified in the inspector, shown at the bottom left of Fig. 6. In the example, we claim that `LIMIT` is always 10. The model checker proves this to be true.

3.4.2 Implementation Characteristics

The integration of analysis tools uses all three important characteristics of language workbenches: language modularity (**Q2**) is exploited for providing the language extensions on which the verification tools rely. Projectional editing is exploited in the decision tables, which could not use the intuitive tabular notation in a parser-based environment. Multi-level transformations are used by the language extensions' transformations back to C, but also by creating a C implementation *and* an implementation in NuSMV or Yices from the same source. The integration of analysis tool also makes use of traditional tool extensibility by the mere fact that Yices and NuSMV are integrated, and by the fact that the results can be shown inside a view in the IDE.

3.5 Requirements Tracing

The relationship between requirements and implementation code is important in the context of many development processes and certification standards. If and when a set of requirements are related to a particular part of the implementation, interesting analyses become possible:

– if the rationale for some piece of implementation code is in doubt, it is easy to find out which requirements prescribe what it is supposed to do

– if a set of requirements changes, it is straightforward to find out which parts of the implementation are potentially affected by the change
– test coverage analysis can be performed by finding out which requirements have test code associated with them.

To support these analyses, implementation artifacts must be traced to requirements [40]. A *trace* is essentially a typed pointer (e.g. `implements` or `tests`) from some part of the implementation to one or more requirements. In terms of tooling the challenge is that traces should be attachable not just to whole artifacts (such as C files or a PDF document), but to arbitrary parts of an artifact, expressed with arbitrary languages (C, default extension or a user extension in case of mbeddr). Also, as the respective program parts are moved, copied or edited, these traces should stay attached to the element.

3.5.1 mbeddr's Solution

Specifying Requirements Requirements can be collected in instances of `RequirementsModule`, a concept defined in the `requirements` language (CSV import is also supported). An example from the Smart Meter project is shown in the left part of Fig. 7. Each requirement has an ID, a short prose summary, and a kind (`functional` in the example). A kind may come with its own additional specifications, expressed in a kind-specific DSL. For example, `timing` requires users to enter a `timing` specification. While the hierarchical structure represents refinement (where a child requirement provides additional details for its parent requirement), constraints such as `conflicts with` or `requires also` cross-cut the hierarchy.

The requirements language is extensible with new requirements kinds and new constraints. Also, completely new DSLs can be plugged in as a means of describing additional data for a given requirement kind. A trivial example of an extension used in the Smart Meter project are the `section` and `affected component` elements visible in the right part of Fig. 7. The optionally visible details view contains additional prose and the kind-specific DSL programs.

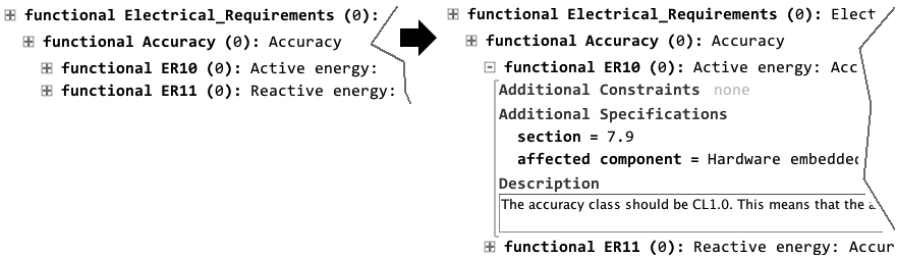

Fig. 7 Left: The editor shows the summary of requirements, organized as a tree. **Right**: The optionally projected details pane allows users to specify an extended prose description, additional constraints among requirements as well as kind-specific data.

Tracing A trace establishes a link between a program element and any target element that implements `ITraceTarget`. `Requirements` implement `ITraceTarget` by default. Importantly, the language that defines the node to which a

trace is attached *does not have to know* about tracing (the two concerns are orthogonal, see Section 4.2). A trace is connected to the traced element, and not just *rendered* next to it: if the element is moved, the trace moves along. In Fig. 8 (left) a trace is attached to an assignment.

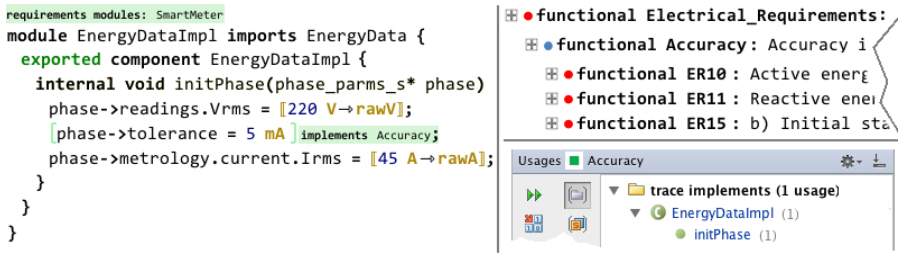

Fig. 8 Left: A component runnable that contains an assignment with an attached requirement trace (the green label `implements Accuracy`). Traces can be attached to arbitrary program nodes, supporting tracing at any level of detail. **Right, Top:** The requirements can be color-coded to reflect whether they are traced at all (grey), implemented (blue) and tested (green). Untraced requirements are red. **Right, Bottom:** The Find Usages dialog shows the different kinds of traces as separate categories. Finally, programs can also be shown without the traces using conditional projection (a mechanism discussed in Section 3.6).

Traces can be evaluated in reverse. For example, Fig. 8 (right, top) shows how requirements can be color-coded to reflect their state. In addition, the generic MPS Find Usages facility has been customized: if the user executes Find Usages for requirements, the various kinds of traces are listed separately in the result (Fig. 8 right bottom, shows the result for the `implements` kind).

Note that attaching traces to the respective model elements cannot be automated. We rely on the discipline of users to create a reasonably complete set of traces. However, our experience is that in contexts where tracing is required (for example, for certification) users usually have this kind of discipline. Also, tracing only works well for requirements that are associated with particular elements. Cross-cutting requirements or those that address non-functional characteristics are harder to trace. This is a fundamental limitation of tracing.

3.5.2 Implementation Characteristics

The tracing feature makes use of language extension. Among other things it enables the contribution of new trace targets, trace kinds and requirements kinds. Language embedding enables plugging in arbitrary languages into the additional specifications section. Projectional editing is exploited in the expandable requirements editors, and, importantly, in the ability to attach the traces to arbitrary program elements while keeping the tracing extension independent of the the languages used to express the traced elements (**Q2**).

3.6 Product Line Variability

Product line engineering involves the coordinated construction of several similar, but different products. Managing the differences over the sets of products

in a product line is non-trivial. The industry best practice approach [7] involves two steps. First, variability among products is described on a conceptual level without a relationship to an implementation. Feature models [6] are often used here. In a second step, parts of implementation artifacts are related to the features: a particular part of the implementation is only part of a given product if the referenced features are selected for this product [36].

For a development tool this leads to two challenges. First, it must be possible to associate arbitrary (parts of) implementation artifacts with configuration features to express the dependency of those artifacts on the feature. The challenge is similar to the one for requirements tracing discussed in the previous subsection. The second challenge lies in enabling the user to comprehend the variability in a program. Depending on the number of variable parts, it is not easy to understand what each variant of each artifact will look like, and whether each variant constitutes a well-formed or even correct program.

3.6.1 mbeddr's Solution

Feature Models Feature models express configuration options and the constraints among them. Features are expressed as a tree where a parent feature expresses constraints regarding the combinations of child features, limiting the set of valid configurations to a manageable size[24]. Constraints include:

- **mandatory**: mandatory features have to be in each product. In Fig. 9, each `Stack` has to have the feature `ElementType`.
- **optional**: optional features may or may not be in a product. `Counter` and `Optimization` are examples of optional features in Fig. 9.
- **or**: a product may include zero, one or any number of the features in an `or` group. In Fig. 9, a product may include any number of features from `ThreadSafety`, `BoundsCheck` and `TypeCheck`.
- **xor**: a product must include exactly one of the features grouped into a `xor` group. in Fig. 9 the `ElementType` must either be `int`, `float`, or `String`.

In addition cross-cutting constraints are supported by most feature model implementations. In mbeddr, a feature can declare `conflicts with` and a `requires also` constraints relative to arbitrary other features. Features may also have configuration attributes (see the `size` of the `fixed` feature in Fig. 9).

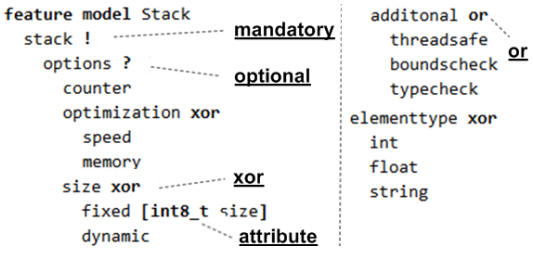

Fig. 9 An example feature model in mbeddr. Until MPS will provide support for graphical notations (planned for 2013), we use a textual notation.

[24] If a product line's variability were just expressed by a set of Boolean options, the configuration space would grow quickly with 2^n, with n representing the number of options.

Instead of using the well-known graphical feature diagram notation, in mbeddr we use a textual notation for representing feature models shown in Fig. 9 (support for graphical syntax in MPS will be available only in 2013).

Configurations A configuration is a named set of selections from the features in a feature model. The selection has to be valid regarding the constraints defined in the underlying feature model. Fig. 10 shows two examples. If an invalid configuration is created, errors are shown in the configuration model.

```
configuration model SimpleStack for Stack
  stack
    options
      counter
      size
        fixed [size = 10]
    elementtype
      int
```

```
configuration model DynamicStack for Stack
  stack
    options
      optimization
        speed
      size
        dynamic
    additonal
      boundscheck
      threadsafe
    elementtype
      float
```

Fig. 10 Two valid configurations of the feature model from Fig. 9.

Presence Conditions A presence condition is an annotation on a program element that specifies under which conditions the program element is part of a variant via a Boolean expression over features. For example, the two red statements in Fig. 1 are only part of a product if the `resettable` feature is selected in the product configuration. The background color of an annotated node is computed from the expression: the same expression results in the same color (an idea borrowed from Christian Kaestner's CIDE [42]).

As a further example of analysis and verification, we use the SMT solver to ensure that a feature model is free from conflicts and that configurations are consistent with the respective feature models. For example, if a feature `A` has a `requires also` constraint to feature `B`, and feature `B` expresses a `conflicts with` constraint regarding `A`, then the feature model contains a conflict.

The fact that one can make arbitrarily detailed program elements depend on features does not mean that no further structuring of the product line is necessary, and all variability should be expressed via fine-grained presence conditions. Instead, presence conditions should be used to configure more coarse grained entities such as the instantiation and wiring of components.

Rendering Variants It is possible to edit the program as a product line (with the annotations, as shown in Fig. 1), undecorated (without annotations) as well as a specific product (in Fig. 1, the `resetted` out event and the `on start` transition would not be shown in the editor if the selected variant did not include the `resettable` feature). The latter is helpful to the programmer to visualize how a given variant will look like in the face of several interacting presence conditions. During transformation, those parts of programs that are not in the product are removed from the model, so no code is generated.

3.6.2 Implementation Characteristics

Just as in requirements traces, we exploit MPS' projectional editing facilities to attach presence conditions to arbitrary program elements, while keeping

the language that defines the presence conditions independent of the language whose concepts are potentially annotated (**Q2**). The ability to show configurations directly in the editor is achieved using conditional projection rules. The product line support also makes use of multi-stage transformation: those element whose presence conditions are `false` are removed from the program by a generic transformation *before* they are transformed any further.

4 Implementation of some mbeddr C Extensions with MPS

To provide a basis for the conclusions regarding feasibility and effort (**Q1**) made in Section 5, this section illustrates the implementation of some of the examples discussed in the previous section. The particular examples are selected to ensure that there are examples for each of the three technical prerequisites we discussed in Section 2: language modularity, projectional editing and multi-stage transformation. We also briefly discuss how traditional tool extensibility fits into the picture. Fig. 11 provides an overview over the most important aspects of MPS language definition.

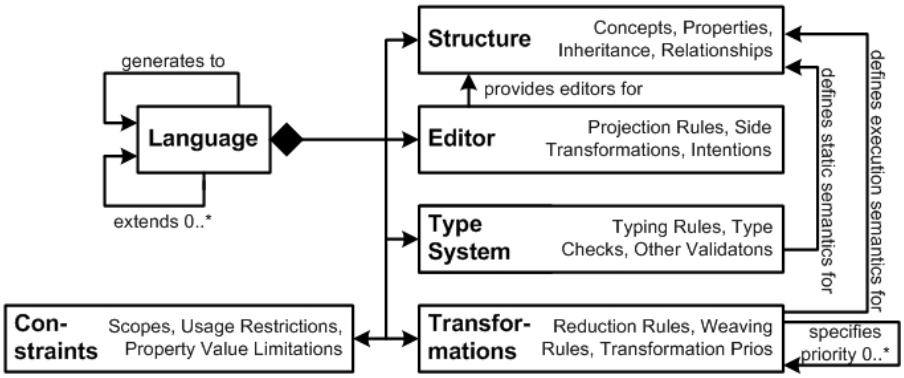

Fig. 11 In MPS, a language consists of several aspects. The figure shows the most important ones (language structure is MPS' term for abstract syntax). In addition, languages can specify refactorings, find-usages strategies, migration scripts and debuggers (all of these are not discussed in this paper). Languages can make use of other languages in their definition and generate down to other languages. Transformations specify priorities relative to other transformations; MPS calculates a global transformation schedule based on these priorities.

4.1 Language Modularity

As we have discussed in Section 2.1, language Extension and language Embedding (both explained in detail in [74]) are the means of language modularity and composition most relevant in language workbenches. In mbeddr we use mostly language Extension: the extending language has a dependency on the base language and can make use of the concepts defined in that base language. Furthermore, it cannot invasively change the base language (**Q2**).

As discussed in Section 2.2, the composability of concrete syntax *never* leads to ambiguities in MPS, so we can discuss the design of modular languages mostly in terms of the abstract syntax (called language structure in

MPS). This, in turn, is essentially object-oriented (OO) design: language concepts have properties and references to other concepts, a concept can extend another concept and can be used polymorphically. Abstract concepts and concept interfaces are supported as well.

Plugging Interfaces into Modules Interfaces are a language concept defined in the `components` extension. However, to integrate with C, they have to be embedded into `module`s, which are defined in the C core language. `Module`s contain a collection of `IModuleContent`s in the `content`s child relationship. Anything that implements the `IModuleContent` interface can be used there:

```
concept Module extends BaseConcept
  children: IModuleContent contents 0..n
```

To allow instances of `Interface` to live inside `Module`s, the `Interface` concept implements `IModuleContent`:

```
concept Interface extends BaseConcept implements IModuleContent
  children: Operation operations 0..n
```

Postconditions An `Operation` has a list of `PrePostCondition`s, an abstract concept that acts as the supertype of `Precondition` and `Postcondition`. A `PrePostCondition` contains a child called `expr` of type `Expression`, which is inherited from the C base language. It is an abstract concept, and all the C expressions (operators, literals, function calls) extend this concept:

```
concept PrePostCondition extends BaseConcept
  children: Expression expr 1
```

In post conditions, the user must have access to the result of the context operation to express things like *the result value is greater than zero*. To make this possible we create a new subtype of `Expression`, the `ResultExpression`:

```
concept ResultExpression extends Expression
```

By making it a subtype of `Expression`, it can be used anywhere an `Expression` is expected. However, this is *not* the behavior we want in this case. We have to restrict the `ResultExpression` to inside of `PostConditions`. And we only want to allow it if the return type of the owning `Operation` is not `void` (there is no meaningful result for `void` operations). Both restrictions are implemented by a constraint. Constraints restrict the usage context of language concepts beyond what is implied by the language structure:

```
can be child constraint for ResultExpression {
  (operationContext, scope, parent, link, childConcept)->boolean {
    boolean isUnderPost = parent.ancestor<PostCondition>.isNotNull;
    boolean isVoid = parent.ancestor<Operation>.returnType.isInstanceOf(VoidType));
    return isUnderPost && !isVoid; } }
```

We still have to define the typing system. In MPS, language concepts specify typing rules, and an inference engine solves the set of equations contributed by the elements in a given program. This way, typing rules are declarative: new rules can be added at any time, and they work together smoothly with rules from the base language. The `ResultExpression` must have the type of the ancestor `Operation`'s return type:

```
rule typeof_ResultExpression for ResultExpression as resultExpr {
   node<Operation> op = res.ancestor<concept = Operation>;
   typeof(resultExpr) :==: typeof(op.returnType); }
```

Embedding State Machines in Components Ideally, independently developed extensions should be usable together in the same program *without* explicitly designing them for any particular combination (**Q2**). For example, any concept that implement the `IModuleContent` interface can be used alongside any other `IModuleContent` in a single program as long as their transformations do not interfere with each other (see Section 6).

However, sometimes it is not so simple. For example, while state machines have been designed to be used as top level concepts in modules (they implement `IModuleContent`), they should also be usable in components. Those, however, expect their contents to implement `IComponentContent`. The mismatch can be resolved by using the Adapter pattern [27]: a new concept `SmCompAdapter` is defined which implements `IComponentContent` and contains a `State Machine`. The editors can be built in a way that users do not see this adapter element when entering or reading the code. The adapter concept lives in a separate language, so neither the `components` nor the `statemachines` languages have a dependency onto the other. This characteristic makes this example an instance of Language Embedding as per Section 2.1 and [74].

4.2 Projectional Editing

MPS' support for language modularization and composition is very much due to its use of projectional editing because concrete syntax introduces very little additional complexity. In this section we describe the definition of editors.

Before we do this, however, we discuss some of the strategies MPS uses to make projectional editing convenient from an end user's perspective (**Q3**). Traditionally, projectional editors were hard to use because the editing gestures known from regular text editors did not work. MPS has solved this issue to a large degree using the following strategies, among others:

- *Aliases* The language concepts legal at a given program location are made available in the code completion menu. For instantiation, a naive implementations requires users to select a concept based on its name. This is inconvenient. In MPS, concepts instead specify an alias. As the user types the alias, the concept is immediately instantiated. For example, a `ForStatement` can be instantiated by typing `for`. Since this is also the starting keyword in `ForStatement`'s projection, it feels like "just typing" a `for` loop.
- *Side transforms* support entering trees linearly. Consider changing `int a = 2;` to `int a = 2 + 3;`. The `2` in the init expression needs to be replaced by an instance of `+`, with the `2` in the left slot and the `3` in the right. Instead of manually removing the `2` and inserting a `+`, users can simply type `+` on the right side of the `2`. This triggers the editor to move the `+` to the root of the subtree, put the `2` in the left slot, and then put the cursor into the right slot to accept the second argument. This way, expressions (or anything else)

can be entered linearly, as in a text editor. Entering expressions linearly requires taking into account precedence rules. To achieve this, each (binary and unary) expression has a numerical `precedence` value associated with it. Side transforms, after constructing a tree fragment, invoke a helper function that reshuffles the tree according to the priorities. This way, a linearly entered expression always reflects precedence in its tree structure, independent of the order in which the subexpressions are entered. Side transforms can also be used to support adding cross-tree parentheses (as in changing `2 + 3*4` to `(2+3) * 4` by pressing `(` on the left of the `2`).
- *Smart delimiters* are used to simplify inputting lists of elements (such as argument lists) separated with a separator (e.g. comma). Users can just press the delimiter key (comma) to add a new element to the list. Smart delimiters can be considered a shortcut for specialized side transforms.
- *Delete actions* are used to similar effect when elements are deleted. Deleting the `3` in `2 + 3` keeps the `+`, with an empty right slot. Deleting the `+` then removes the `+` and puts the `2` at the root of the subtree.
- *Wrappers* Consider `int a;`. Users want to enter a local variable by starting with the type, not by explicitly selecting `LocalVariableDeclaration` from the code completion menu. A wrapper can be used to this effect. It *wraps* the `LocalVariableDeclaration` with `Type`. Once a `Type` is entered, the wrapper implementation creates a `LocalVariableDeclaration`, puts the `Type` into its `type` field and moves the cursor into the `name` slot.
- *Smart references* achieve a similar effect for references (as opposed to children). Consider pressing `Ctrl-Space` after the `+` in `2 + 3` and assume that a couple of local variables are in scope which should be available in the code completion menu. Technically, a `VariableReference` has to be instantiated *first*, before its `variable` slot *then* is made to point to a variable. This is tedious. Smart references trigger special editor behavior: if in a given context a `VariableReference` is allowed, the editor *first* evaluates its scope to find the possible targets and puts them into the code completion menu. If a user selects one of them, *then* the `VariableReference` is created, and the selected element is put into its `variable` slot. This makes the reference object transparent in terms of the user experience.

The above features are the major contributors to making the user's editing experience very close to editing text files. In addition, like any other modern IDE, projectional editors use intentions (also known as quick fixes). These are small in-place transformations that can be triggered by the user, and, for example, wrap a statement with an `if`, or wrap an expression with a type cast.

As we have seen with the `can be child` constraint for the `ResultExpression` in the previous section, projectional editing also supports restricting the user from entering language constructs in contexts where they are not allowed. Such restrictions can take into account arbitrary structural context, the type system or any other program analysis that can be expressed with Java code. These restrictions effectively guide users towards building programs that are correct with regards to structure and static semantics. However, it may be unintuitive to users in some situations, which is why the traditional approach

(allowing users to enter code and then reporting an error if something are detected to be illegal) is supported as well, as long as the entered concept is structurally valid in the given location.

To support language composition effectively, the mechanisms for improving the editing experience discussed above must also work in the face of modular language composition (**Q2**). For example, if independently developed language extensions define the same alias for different concepts that are valid *at the same location*, then, after typing the alias, MPS opens the code completion menu and forces the user to decide which concept to instantiate. This facilitates modular extension, while degrading the editor experience only slightly instead of failing or requiring invasive changes. This situation is rare, because specific extensions can be restricted to very specific context as mentioned in the previous paragraph. Another example are right transformations. They can also be context-limited with an applicability condition. For example, if a language extension defines a new kind of dot operator (as in `a.b`) that potentially conflicts with `struct` or `union` access in C, the right transformation that accepts the dot can be limited to a particular type of the context expression.

A final benefit of projectional editors is that program parts irrelevant in a specific situation can be hidden using conditions in the projection rules, essentially supporting views on programs[25]. For example, as discussed in Section 3.6, a program can be shown in a variant-specific way by hiding the parts that are not included in a particular variant.

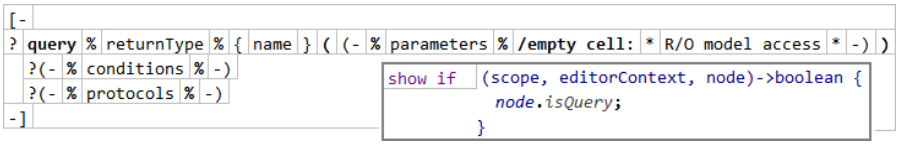

Fig. 12 The editor for `Operation` fundamentally consists of a list of cells (`[- .. -]`). The first element is the `query` cell (marking operations as idempotent) which is projected conditionally: the `?` operator contains a condition that is `true` if the `isQuery` property is `true` (this is specified in the inspector, as shown in the inset). The next cell (`%returnType%`) embeds the editor for the `returnType` child link. We then embed the list of parameters between parentheses, separated by a comma (also specified in the inspector — not shown). Finally, on new lines each, we embed the editors for `conditions` and `protocols`. Note that the indentation level and other stylistic issues are prescribed by the projection.

An Editor for Operations The basic abstraction for defining editors is the *cell*, the smallest unit of projection. A cell may contain a constant text, a collection of other cells laid out in a particular way, or refer to a property, reference or child link of a concept — in which case the editor for that property or link is embedded. The editor for operations, shown in Fig. 12, embeds the editor for the return type, the parameters of the operation, as well as the pre- and post-conditions and the protocols. Cells can be shown optionally: the `query` flag on interface operations is an example.

Requirements Traces Traces use MPS' annotations mechanism. An annotation is a concept whose instances can be added as a child to a node *without*

[25] The upcoming version 3.0 of MPS will support several editors for the same concept.

that node's concept declaring this child. Annotation can be added to any node, unless explicitly restricted. Here is the definition of the `TraceAnnotation`:

```
concept TraceAnnotation extends NodeAnnotation
  children: TraceKind tracekind 1
            TraceTargetRef refs 0..n
  concept properties: role = trace
  concept links: annotated = BaseConcept
```

Annotations must extend the MPS-defined concept `NodeAnnotation`. It can then define any arbitrary structure. For the requirements trace we define two child links: the first one represents the kind of trace (such as `implements` or `tests`), the second one contains a collection of references to trace targets (e.g. requirements). An annotation has to specify two additional properties: the `role` property defines the role name used by the `TraceAnnotation` under the annotated node. In the example, a `TraceAnnotation` would be stored in the `@trace` link. The `annotated` property specifies which concepts this annotation can be attached to. `BaseConcept` is a supertype of all concepts, so the `TraceAnnotation` can be attached to any program node.

```
[> annotated_node ?[> % tracekind % F(> % refs % /empty cell: ... <) <] <]
```

Fig. 13 The editor for the `TraceAnnotation` embeds the editor of the annotated node (represented by the predefined cell type `annotated_node`), resulting in the annotation editor "wrapping around" the editor for the annotated node.

We discuss annotations in the section on projectional editing because the editor, shown in Fig. 13, is the most interesting aspect. The `annotated_node` cell (a predefined cell type), embeds the editor of the element *under* which the `TraceAnnotation` lives. This way, although the `TraceAnnotation` is structurally a child of the annotated node, the annotation's editor is rendered *around* the annotated node. In case of the `TraceAnnotation`, we first embed the annotated node, and then, to its right, we render the `tracekind` and the set of trace target references. The projection of the `tracekind` and the references is conditional: the expression behind the `?` (not visible in the figure) makes sure these cells are only projected if a global flag is set; otherwise the program is shown without traces.

The annotation mechanism is also used in the product line variability presence conditions. In the projection mode that renders a particular product configuration we use a similar conditional projection rule to remove the whole annotated element (not just the annotation itself) in case the presence condition is `false` for the currently rendered variant.

Decision Tables Decision tables are interesting because they use a tabular notation. Defining a tabular editor is straightforward: the definition contains a `table` cell, which delegates to a Java class that implements `ITableModel`. It provides methods such as `getValueAt(int row, int col)` or `deleteRow(int row)`, which have to be implemented for any given editor. To embed another node in a table cell, the implementation of `getValueAt` returns this node.

A Side Transformation for Invocations on Ports In Section 3.3.1 we have shown example code that invokes an operation on a required port:

```
requires EcRobot_Motor motorLeft
void dt_driveForwardFor(uint8 speed, uint32 ms) <- op dt.driveForwardFor {
   motorLeft.set_speed(((int8) speed)); }
```

The `set_speed` invocation on the `motorLeft` port can be used to illustrate side transformations. The user first enters `motorLeft`. This is an instance of `RequiredPortRefExpr`. Then, if the user enters a dot on the right of this expression, it has to be transformed into a `RequiredPortOpCallExpr`, and the user has to enter a reference to the an operation defined by the referenced port's interface. Here is the right transformation code that accomplishes this:

```
right transformed node: RequiredPortRefExpr creates: RequiredPortOpCallExpr
  matching text: .
  do transform (operationContext, scope, model, sourceNode, pattern)->node<> {
    node<RequiredPortOpCallExpr> call = new node<RequiredPortOpCallExpr>();
    sourceNode.replace with(call);
    call.requiredPort = sourceNode; }
```

4.3 Multi-Stage Transformations

In MPS, multi-level language extension is the norm, so support for multi-stage transformation must be available as well. These transformations are model-to-model transformations: a new AST is created from an existing AST. To preserve modularity and composability (**Q2**), transformations specify a ordering relative to other transformations, and MPS computes a global order for the transformation of a program based on the languages used in the program.

In MPS, a transformation is specified as a set of transformation rules. Each rule specifies the node it transforms, an optional condition that determines when it applies, as well as a code template in the target language that describes the result of the transformation. Templates are valid target language program fragments, annotated with macros (the editor provides IDE support for the target language — see below). Macros are replacement rules executed during transformation execution. Different macros exist for replacing whole nodes, for changing the values of properties, and for retargeting a reference. We will see examples of these in the following paragraphs. While this approach guarantees to generate only structurally valid ASTs, the transformations are not hygienic: the developer must make sure that references bind to the correct targets, and newly introduced symbols must have unique names so they are not accidentally targeted by existing references if they are rebound during a transformation or name-resolved in the generated, textual C program.

Transforming Interfaces Interfaces have no equivalent in C, they are used purely on the extension level for checking compatibility of ports. The generator is trivial: it uses an `abandon node` rule to discard the `Interface` input node.

Transforming State Machines State machines are transformed into an `enum` for the states, an `enum` for the events, a `struct` that holds the state machine's data (variables, current state), and a function that implements the behavior. The function takes two arguments: the `struct` that represents a state machine instance as well as the event the instance is supposed to consume (events can have arguments, but this is not discussed for reasons of simplicity).

```
concept    TriggerSMStatement
inheritors false
condition  <always>
    --> module dummy {
          enum eventEnum { e1; e2; }
          struct instanceData { };
          var instanceData theStatemachine;
          void smExecFunc(instanceData* instance, eventEnum event){ }
          void someMethod() {
            <TF [ { ->$[smExecFunc](&$COPY_SRC$[theStatemachine], ->$[e1]);} ] TF>
        } }
```

Fig. 14 Transformation macros are used to replace dummy nodes (such as the reference `e1`) with the code created by the transformation based on the input node. Reference macros (`->$`) are used to wire up references, and `$COPY_SRC$` macros are used to replace entire nodes. Behind each macro is an expression that computes the node that should be used to replace the dummy node. For example, behind the `$COPY_SRC$[theStatemachine]` is an expression that returns the variable that holds the instance data for the current state machine instance. We describe more details about transformations in the running text.

The `trigger` statement, which fires an event into a state machine instance (e.g. `trigger(aStatemachineInstance, anEvent)`), must be transformed to a call to this function, supplying the `struct` instance that corresponds to the instance of the triggered state machine, plus the `enum` literal that represents the event. Figure Fig. 14 shows the respective transformation rule. It has three parts: the part above the `->` specifies that the transformation rule applies to instances of `TriggerSMStatement`. The part enclosed in `<TF .. TF>` is called the *template fragment*. Its content replaces the `TriggerSMStatement` during execution of the transformation. The rest of the code is used for scaffolding.

Scaffolding is necessary for the following reasons: as we have mentioned above, the code inside the template fragment must be valid C code, *even in the template* (this is why MPS can provide IDE support for the code in the template). So to be able to generate a reference, the template must contain a node that can be referenced by the reference, even if we do not intend to generate the reference target, because it already exists in the to-be-transformed tree. So, for example, to be able to write a function call in the template, we first have to have a function (`smExecuteFunction`), to be able to reference an `enum` literal, we first need an `enum`, and so on. During the execution of the transformation, references are "rewired" using the `->$` macro. Its embedded expression returns the target for the reference, typically an element that already exists (or has been created by the transformation) in the output tree.

In our case we want to generate a call to a function with two arguments, so the scaffolding has to contain a function with these two arguments as well — and they must have the correct type to avoid getting type errors *in the template*. Please see the caption of Fig. 14 for further details.

The transformation rule we have seen above is an example of a *reduction* rule. Reduction rules *replace* the input node with the rule's result. MPS also supports various other kinds of rules, including conditional root rules (which create a new node without a specific input element) and weaving rules (which

create a new node at a specified location different from the input node's location in the output tree).

Transforming a Mock Component Mock components are a special kind of component which declaratively express the behavior they expect to see on their provided ports in the context of a test case [70]. Here is an example:

```
mock component PasswordMock {
  total no. of calls is 3
  sequence {
    step 0: energyDataAccess.hasMeterStatus return false;
       assert 0: parameter expectedStatus: expectedStatus == PASSWORD_OK
    step 1: energyDataAccess.hasMeterStatus return false;
    step 2: setPasswordHandler.processCommand return true;
```

These expectations are transformed into implementations of the component operations that track invocations and check whether the expectations are met. For this to work, the mock-to-component transformation has to run *before* the component-to-C transformation. To achieve this, the mock-to-components generator specifies a `strictly before` constraint relative to the components-to-C generator. The following code shows the overall mapping configuration for a progam that uses mocks, components, unit tests, and, of course, C.

```
[1] core.removeCommentedCode
[2] ext.components.mock
[3] ext.components.main, core.unittest
[4] core.ctext
```

Based on the specified relative priorities, MPS has computed an overall order comprising four separate phases. In phase 1, we remove commented code (since it should not end up in the resulting C text file). In phase 2 we run the mock component transformation. As expected, it runs *before* the components-to-C transformation, which runs in phase 3, together with the unit-test-to-C transformation. Phase 4 finally generates the resulting C text.

Implememting Variants Implementing product line variability is a different kind of transformation in that it is generic with regard to the transformed languages (presence conditions can be attached to any arbitrary program element). If the presence condition is `false` for the selected configuration during transformation, the respective program element has to be removed from the program (see the `affectedElement.delete` in the code below). Since this is a generic transformation, it is implemented as a transformation script. In contrast to the template-based approach shown above, a transformation script uses MPS' node API directly to transform the AST.

```
mapping script removePLEStuff pre-process input model, top-priority group: true
(model, genContext, operationContext)->void {
  node<...> config = // the configuration that specifies which variant to generate
  foreach pc in model.nodes<PresenceCondidtion> {
    if (!(pc.condition.isSelectedInTransformationConfiguration(config))) {
      node<> affectedElement = pc.parent;
      affectedElement.delete; } } }
```

The script works as follows. First, it finds the configuration element that specifies which variant should be generated. It then finds all `PresenceCondidtion`s

in a model and evaluates each of them relative to the selected variant configuration. If a presence condition evaluates to `false`, the script removes the element to which the presence condition is annotated — the parent of the presence condition. Notice how the transformation is put into the `top-priority group`, which means it runs in the first phase of the transformation without explicitly specifying priorities relative to other generators. This is important, because we may not even know which other transformations are executed for the program, so we cannot explicitly declare dependencies relative to them.

4.4 Tool Extensions

While we emphasize language extension in this paper because it distinguishes our approach, tool extensibility is also important. There are two dimensions.

The first one is tool extensibility relative to language extensions. The IDE features for a given language have to be extended along with language extensions. In MPS this is mostly automatic: as you define a language extension, you automatically get syntax coloring, error annotation and code completion. For additional features such as refactorings, the customization of syntax highlighting or debugging, custom APIs exist. The second dimension is tool extensibility independent of language extensions. For example, one might want to integrate a view to show graphical overviews of the programs, or to show the results of some analysis. MPS provides a plugin API do define such tool extensions. As an example, the integration of NuSMV comprises the following steps:

1. The input language for NuSMV is implemented in MPS.
2. A transformation is developed that maps state machines marked as `verifiable` to the corresponding NuSMV program.
3. A new window is added to MPS that shows the the verification result.
4. An action `Verify` is contributed to the context menu of state machines. When executed, the action runs NuSMV on the generated input program, parses the textual result, and populates the table in the new window.

Steps 1 and 2 are no different from any other language or transformation definitions and will not be discussed any further. We focus on 3 and 4.

Additional windows (such as the tables in the right half of Fig. 6) can be defined as part of a language. If a program uses that language, the additional window is available. A window is essentially a Java class that has a predefined structure and specifies a caption, an icon, and a position in the UI frame. A window also defines a set of predefined methods, the most important of which creates the Java Swing-based UI. Finally, MPS also supports actions, which define UI aspects (mnemonic, caption, icon, menu location) as well as the code to be executed (running NuSMV and populating the UI in our example).

5 Discussion and Lessons Learned

In this section we address the research questions stated in Section 1 and discuss our experience with developing mbeddr from two perspectives: the usefulness of MPS and language workbenches for building customized domain specific

tools (tool developer perspective) and the implications of the mbeddr stack for the embedded software developers (end user perspective). The latter is important to validate the overall approach. This section relies heavily on the experiences from the first commercial use of mbeddr, a project that develops the software for a 3-phase smart meter. A smart meter is an electrical meter that continuously records the consumption of electric power in a home and sends the data back to the utility for monitoring and billing. The software comprises ca. 20.000 lines of mbeddr code, has several time-sensitive parts that require a low-overhead implementation and will have to be certified by the future operator. This leads to an emphasis on testing, formal analyses and requirements tracing. The software exploits existing code in the form of header files, libraries and code snippets. While the project is still going on, we can already report some experiences and draw some conclusions.

5.1 Addressing Q1: Feasibility of building sophisticated IDEs with MPS

Tool Scalability The scalability of MPS as a language workbench can be measured in different ways including its ability to manage the complexity associated with large or many languages, the learning curve, working in teams and in terms of supported language sizes and tool performance. In this section we look at language size and tool performance. The others are discussed below.

Typically, lines of code (LOC) are used to describe the size of a program. In a projectional editor like MPS, a "line" is not necessarily meaningful. However, it is feasible to estimate the equivalent LOC number by counting the occurrences of certain language definition ingredients and associating a LOC-factor with them. For example, the statements that are used in the imperative parts of a language definition (e.g. in scopes or type system rules) have a LOC factor of 1.2 since many statements embed higher order functions and would span more than one line. 1.2 turned out to be a reasonable average. Another example for a LOC-factor is an intention: an intention declares the concept it applies to, a label and an applicability condition. These are one line each. It also contains a number of statements which are counted separately, as statements. Hence, the LOC factor for intentions is 3 (a similar argument holds for constraints or reference scopes). As a final example, consider editor cells. An editor definition contains a large number of cells, and we found that on average, 4 occur per "line" of editor definition, leading to a LOC-factor of 0.25. The third column of the table in Fig. 15 shows the factors for all kinds of ingredients involved in language definition. While this approach is an approximation, we have made several manual checks and found that it is accurate enough to get an feel for the size of various language implementations.

Fig. 15 shows the result of the LOC count for the mbeddr core, i.e. C itself plus unit test support, decision tables and build/make integration. According to the metric discussed above, the core comprises about 8,640 lines of code. This includes all aspects of language definition (including syntax, type system, to-text-generators) as well as the IDE (code completion, syntax highlighting, quick fixes). Using the same metric, the components extension (interfaces, components, pre- and post-conditions, support for mock components and a

generator back to plain C) is ca. 3,000 LOC. The state machines extension is ca. 1,000 LOC. We consider these numbers an indication that MPS supports very concise definition of languages. While we have not implemented C with other language workbenches, some of the authors have experience with other tools. For example, implementing C and its IDE with Xtext would require significantly more code, since many language aspects are not supported first class (e.g., type systems) or must be implemented using much more verbose Java code that relies on Xtext APIs.

Element	Count	LOC-Factor	LOC-Equivalent
Language Concepts	260	3	780
Property Declarations	47	1	47
Link Declarations	156	1	156
Editor Cells	841	0.25	210
Reference Constraints	21	2	42
Property Constraints	26	2	52
Behavior Methods	299	1	299
Type System Rules	148	1	148
Generation Rules	57	10	570
Statements	4,919	1.2	5,903
Intentions	47	3	141
Text Generators	103	2	206
Total Approximate LOC:			8,640

Fig. 15 We count various language definition elements and then use a factor to translate them into equivalent lines of code.

Efforts The core C implementation has been developed in ca. four person months divided between three people, resulting in roughly 2,200 LOC per person month. Extrapolated to a year, this would be 26,400 LOC per person. According to McConnell[26], in a project up to 10,000 LOC, a developer can typically do between 2,000 and 25,000 LOC per year, so we are just slightly above the typical range. The state machines extension (including the generator and the integration with the NuSMV model checker) and components extension (including a generator to C with polymorphic and static wiring options, testing support, pre- and post conditions and protocol state machines) have both been implemented in about a month. The unit testing extension and the support for decision tables have been implemented in a few days.

MPS Learning Curve MPS is a comprehensive environment for building and composing languages. In addition to defining the structure, syntax and an IDE, it also supports advanced features such as type systems, refactorings and debuggers. Consequently, the learning curve for the language developer (not the end user/C programmer) is significant. Our experience with several novice MPS language developers is that it takes around four weeks of full time training and practice to become a decent MPS language implementor. With improved documentation and some cleanup of MPS itself, this effort may be reduced to three weeks, but it is still a significant investment.

[26] `http://codinghorror.com/blog/2006/07/diseconomies-of-scale-and-lines-of-code.html`

However, once a developer has mastered the learning curve, MPS scales well: increasingly large and complex languages are *not* overly more complex to build. This is in sharp contrast to our experiences with other, parser-based language workbenches, where, with increasing language complexity, the accidental complexity of the language implementation increases significantly.

5.2 Addressing Q2: Language Modularity for non-trivial Use Cases

Language modularity, extension and composition is central to mbeddr in two ways (**Q2**). First it enables third parties to create C extensions without agreeing on how to invasively change C. Second, modular language extension also helps scale the system from the perspective of the language engineer. At this point, mbeddr consists of 51 separate languages with clear dependencies on each other. Putting all the language concepts from these languages into one single language would quickly become unmaintainable.

The integration of formal verification, a problem typically associated with *tool* extension and integration, has been reduced mostly to a *language* integration problem. We implemented the NuSMV and Yices input languages in MPS, reusing part of the C expression language. Then we implemented a transformation from domain-specific abstractions to these input languages. Only the execution of the verification tool, the lifting of the verification results and their representation in the UI remained as a *tool* integration problem. This approach substantially reduced the effort for the integration.

While users can make use of the existing extensions that come with mbeddr (see next subsection), they are encouraged to build their own modular extensions specific to their system context. So far we made use of this possibility in the Smart Meter project in the following ways:

- *Units:* A major part of the Smart Meter application logic performs computations on physical quantities (time [`s`], current [`A`] or voltage [`V`]). We have created a language extension that adds units to types and literals (as in `int8/V/ voltage = 10V;` see also the left side of Fig. 8). The type system has been extended with unit checks and computations (for example, adding `V` and `A` results in a type error and multiplying `V` and `A` results in `W`). The benefits of this extensions are mostly in type checking, but some conversions are also generated into the resulting C code. Using types with units also improves the readability and comprehensibility of the code (important in Smart Meter, which relies on a significant existing code base). While still a modular extension, this extension is not specific to the Smart Meter project and has since been migrated into the mbeddr default extensions.
- *Registers:* Our target processor has special-purpose registers: when a value is written to such a register, a hardware-implemented computation is automatically triggered based on the value supplied by the programmer. The result is then stored in the register. If we want to run code that works with these registers on the PC for testing, we face two problems: first, the header files that define the addresses of the registers are not valid for the PC's processor. Second, there are no special-purpose registers on the PC,

so no automatic computations would be triggered. We solved this problem with a language extension that allows us to define registers first class and access them from C code (see code below). The extension also supports specifying an expression that performs the computation. When the code is translated for the real device, the real registers are accessed using the processor header files. In testing we use generated `structs` to hold the register data and insert the expression into the code that updates the struct, simulating the hardware-based computation.

```
exported register8 ADC10CTL0 compute as val * 1000

void calculateAndStore( int8 value ) {
  int8 result = // some calculation with value
  ADC10CTL0 = result; // actually stores result * 1000}
```

– *Interrupts:* Many aspects of the Smart Meter system are driven by interrupts. To integrate the component-based architecture used in Smart Meter with interrupts, it is necessary to be able to trigger component runnables (methods) via an interrupt. To this end, we have implemented a language extension that allows us to declare interrupts. In addition, the extension provides runnable triggers that express that a runnable is triggered by an interrupt. The following example declares to interrupts (left) and the component runnable `interruptHandler` (right) is declared to be triggered by an interrupt.

```
module Processor {
  exported interrupt USCI_A1
  exported interrupt RTC }
```

```
exported component RTCImpl {
  void interruptHandler() <- interrupt {
    hw->pRTCPS1CTL &= ~RT1PSIFG; } }
```

Note that we do not specify *which* interrupt triggers the runnable because this is done as part of component instantiation (not shown). There, we also check that each interrupt-triggered runnable has at least one interrupt assigned. In addition, for testing purposes on the PC, we have language constructs that simulate the occurrence of an interrupt: the test driver simulates the triggering of interrupts based on a test-specified schedule and checks whether the system reacts correctly.

Based on these extensions, we can draw preliminary conclusions regarding the feasibility of incremental, modular language extension:

– Building a language extension should not require changes to the base language. This, in turn, requires that the base language is built with extension in mind to some degree. Just like in OO programming, only things of a certain granularity can be extended or overwritten (in OO you cannot override lines 10 to 12 in a 20 line method). In addition to being useful in their own right (see below), the implementation of the default extensions also served to verify that the C core language is in fact extensible and the extensions for Smart Meter further demonstrate this point. The registers extension discussed above requires new top level module contents (the register definition themselves), new expressions (for reading and writing into the registers), and embedding expressions into new contexts (the code that simulates the

hardware computation when registers are written). All of those have been built without changing C. Similarly, the interrupt-based runnable triggers have been hooked into the generic trigger facility that is part of the components language. The latter is an example of where the base language (the components extension in this case) has been built with extensibility in mind: an abstract concept `AbstractTrigger` had been defined, which has been extended to support interrupts. Even the units extension, which provides new types, new literals, overloaded typing rules for operators and some adapted code generators has been developed in a modular way, without changing the C base language[27].

- Once a language is designed in a reasonable way (as discussed in the previous item), the language (or parts of it) should be reusable in contexts that had not been specifically anticipated in advance. Embedding state machines into components (discussed at the end of Section 4.1) is an example. We also reuse the C expression language inside the guard conditions in a state machine's transitions, where we use constraints to prevent the use of those C expression that are not allowed inside transitions (for example, references to global variables). We also used decision tables in components. The Smart Meter system contains more examples: expressions have been embedded in the register definition for simulating the hardware behavior, and types with measurement units have been used in decision tables. Again, no change to the existing languages has been necessary.
- Ideally, independently developed extensions should not interact with each other in unexpected ways. We have not seen such interactions so far, in the default extensions or in Smart Meter. While there is no automatic way to detect such interactions or declare incompatibility between languages or extensions, the following steps can be taken to minimize the risk of unexpected interactions. Names of generated C elements (variables, functions) should be qualified to make sure that no name clashes occur. Also, an extension should avoid making specific assumptions about or changing the environment in which it is used. For example, it is a bad idea for a new `Statement` to change the return type of the containing function during transformation because two such badly designed statements could not be used together in a single function (they may require *different* return types for that function). Finally, in traditional parser-based systems, there may be syntactic interactions between independently developed extensions. As we have discussed at length, this *never* happens in MPS.

The efforts are also interesting. The registers and interrupt extensions have been built in 3 hours each. The extension for types with physical units took about 4 days. In the context of a development project which, like Smart Meter, is planned to run a few person years, these efforts can easily be absorbed and well worth the effort in terms of the improved safety and testability.

[27] During the implementation of the default extensions we found a few bugs in the C base language that prevented modular extension. These were not conceptual problems, but real bugs. They have been fixed, so C can now be extended meaningfully in a modular way.

5.3 Addressing Q3: Relevancy and Usefulness for End Users

Usefulness of the existing Extensions While the ability to simply define extensions specific to a platform, architecture or system is paramount to mbeddr, it is also useful to evaluate whether the existing default extensions are useful in practice. If so, this proves that the (relatively low) efforts invested into implementing those extensions leads to tools that are relevant in real-world contexts. In Smart Meter, we use the following default extensions:

- *Components:* We use mbeddr's components to encapsulate the hardware dependent parts. By exchanging the hardware-dependent components with stubs and mocks that provide the same interfaces, we can run integration tests on a PC without using the actual target device. As a side effect we can now debug the software on a normal PC, using the mbeddr debugger. While this does not cover all potential test and debugging scenarios, a significant share of the application logic can be handled this way.
- *State Machines:* The smart meter communicates with its environment via several different protocols. So far, one of these protocols has been refactored to use a state machine. This has proven to be much more readable than the original C code. We combined components and state machines which allowed us to decouple message assembly and parsing from the application logic in the server component.
- *Requirements Tracing:* Smart Meter also make use of requirements traces. During the upcoming certification process, these will be extremely useful for tracking if and how the customer requirements have been implemented. Because of their orthogonal nature, the traces can be attached to the new language concepts specifically developed for Smart Meter.
- *Analyses:* We use decision tables to replace nested `if` statements and used the completeness and determinism analysis to uncover bugs in the code base. We also model checked the protocol state machines. This uncovered bugs introduced when refactoring the protocol implementation from C to state machines. We plan to extend the existing model checking support to be able to check the compatibility of two or more collaborating state machines, which is relevant for the client and server parts of the protocols.

Summing up, the mbeddr default extensions have proven extremely useful in the development of Smart Meter. The fact that the extensions are directly integrated into C (as opposed to the classical approach of using external DSLs or separate modeling tools) reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.

Scalability of mbeddr Let us first look at the scalability regarding the ability to work with large or many programs. We have performed scalability tests and found that mbeddr scales to at least the equivalent of 100,000 lines of C code in the developed system. These tests were based on automatically generated sample code and measured editor responsiveness and transformation times. While there are certainly systems that are substantially larger, a significant share of embedded software is below this limit and can be addressed

with mbeddr. For example, the Smart Meter system is 20.000 lines of mbeddr code. Since there is a factor of ca. 1.5 between the mbeddr code and generated C, the Smart Meter system corresponds to ca. 30.000 lines of C.

One criticism that has been used against language extension is that the language will grow large and that it is hard for users to learn all its constructs. In our experience, this is not a problem in mbeddr for the following three reasons: first, the extensions provide linguistic abstractions for concepts that are well known to the users: state-based behavior, interfaces and components or test cases. Second, the additional language features are easily discoverable because of the IDE support. Third, and most important, these extensions are modularized, and any particular end user will only use those extensions that are relevant to whatever his current program addresses. This avoids overwhelming the user with too much "stuff" at a time.

Usability of Projectional Editing As we have seen, projectional editing has advantages: it contributes to enabling the modularization, extension and composition of languages, it supports mixing textual and non-textual notations (decision tables), allows annotations of programs (as in product line and traceability support) and it supports partial projection of programs (as in the product line support). However, projectional editing also has drawbacks.

First, while MPS' user experience comes very close to real text editing (see Section 4.2), there are some idiosyncrasies users have to get used to (e.g. selecting parts of programs). Experience shows that after a few days the editor is not perceived as a disadvantage anymore, some people actually prefer it over normal text editors. However, users have to get through the first few days of getting used to the editor. We expect this to be less of an issue in the future as the MPS team is working on solving the few remaining issues.

Second, a few things are just not possible with projectional editors. One of them is putting comments around *certain* code segments. Commenting is easily supported for entire subtrees (MPS provides a way of suppressing errors in commented code, override the syntax coloring and prevent elements from being referenced). However, cross-tree comments, as in `boolean b = true /*|| false*/;` are not possible since the `true` node is a child of the `OrExpression`. It remains to be seen whether this is a significant issue in practice.

Third, since models are not stored as readable text but rather as an XML document, infrastructure integration can be challenging. MPS provides an integration with mainstream version control systems including CVS, Subversion and git, and also supports diff/merge based on the projected syntax. However, the projected diff/merge is only supported *inside* MPS, so a diff shown in the browser (for example as part of the gerrit code review tool) will show the XML and is hence not useful. We have a lot of experience with using MPS in a team of eight people in the research project during language development. Except for a few bugs in MPS (fixed in the meantime), teamwork works well. Since end users use the same approach, we assume that this will be the case as well for end users. The Smart Meter team only has three developers, so we cannot yet draw significant experiences from this project.

Interoperability with Textual Code Additional effort is required to integrate with existing legacy code. As a consequence of the projectional editor, we have to parse the C text (with an existing parser) and construct the MPS AST. mbeddr provides an importer for header files as a means of connecting to existing libraries. However, mostly as a consequence of C's preprocessor which allows all kinds of mischief to be done to otherwise well-structured C code, this importer is not trivial. For example, we currently cannot import all alternatives expressed by `#ifdef`s. Users have to specify a specific configuration to be imported (in the future, we will support importing of all options by mapping the `#ifdef`s to mbeddr's product line variability mechanism). Also, header files often contain platform-specific keywords or macros. Since they are not supported by the mbeddr C implementation, these have to be removed before they can be imported. The header importer provides a regular expression-based facility to remove these platform specifics before the import. The Smart Meter project, which is heavily based on an existing code base, also drives the need for a complete source code importer (including `.c` files, and not just header files), which we are currently in the process of developing.

The parser behind this importer will also be integrated into MPS' paste handler, so textual C source can be pasted into the projectional editor. While copy and pasting from MPS to text works by default (if the syntax of the code is textual), the reverse is not true and has to be built specifically.

The integration of legacy code describe in this paragraph is clearly a disadvantage of projectional editing. However, because of the advantages of projectional editors discussed in this paper, we feel that it is a good trade-off.

Leaky Abstraction A fundamental problem with higher-level abstractions is that in case of an error (or unacceptable performance or resource consumption), a user may have to deal with the underlying implementation. However, the user may not understand this implementation, because he did not write the code and had previously just relied on the higher-level abstraction. This problem exists for macros, libraries and also for mbeddr's language extensions. We try to limit this problem as far as possible. Primarily we make sure that every valid extension-level program leads to a valid C program by relying on automated unit tests and (initially maybe overly) strict type system rules and constraints. mbeddr also provides an extensible debugger that lets users debug programs at the extension level and a tool to find the source of a log statement on the extension level based on the log output. As a last resort, MPS provides a tracing facility where low-level code is traced back through the transformation process to its high-level source. Based on the experience with Smart Meter, we feel that we have addressed the issue to a degree that works for users.

Runtime Overhead Generating code from higher-level abstractions may introduce performance and resource consumption overhead. While we have not yet performed a systematic analysis of the overhead incurred by the mbeddr extensions, it is low enough to run the Smart Meter system on the hardware intended for it. Some extensions (registers, interrupts or physical units) have no runtime overhead at all since they have no representation in the generated

C code. Others, such as the components, incur a very small overhead as a consequence of indirections from function pointers. We will conduct a systematic investigation in the future.

6 Related Work

In this section we discuss the related work regarding the core contribution of this paper: language engineering, language workbenches and extensible IDEs. Section 1.2 has already addressed related work regarding embedded systems.

Parsers and Grammars In [44] Kats, Visser and Wachsmut describe the trade-offs with non-declarative grammars. Grammar formalisms that cover only subsets of context-free grammars are not closed under composition and composed grammars are likely to be outside of the respective grammar class. Composition (without invasive change) is prohibited. Formalisms that implement full context-free grammars avoid this problem and compose much better.

Most mainstream parser generators (such as ANTLR [61]) do not support the full set of context-free grammars and hence face problems with composition. In contrast, the Syntax Definition Formalism [35] (SDF) does support full context-free grammars. Based on a scannerless GLR parser, it parses tokens and characters in a context-aware fashion. There will be no ambiguities if grammars are composed that both define the same token or production *in different contexts*. This allows, for example, to embed SQL into Java (as Bravenboer et al. discuss in [50]). However, if the same syntactic form is used by the composed grammars *in the same location*, manual disambiguation becomes necessary. In SDF, disambiguation is implemented via quotations and antiquotations ("escape characters") which are defined in a third grammar that defines the composition of two other independent grammars [10]. The SILVER/COPPER system described by van Wyk in [76] instead uses disambiguation functions written specifically for each combination of ambiguously composed grammars. In MPS disambiguation is never necessary — in the worst case, the user makes the disambiguating decision by picking the correct concept from the code completion menu. Given a set of extensions for a language, SILVER/COPPER allows users to include a subset of these extensions into a program as needed (demonstrated for Java in AbleJ [77] and for SPIN/Promela in AbleP [51]). A similar approach is discussed for an SDF-based system in [11]. However, ad-hoc inclusion only works as long as the set of included extensions (presumably developed independently from each other) *are not ambiguous* with regards to each other. Otherwise disambiguation has to be used. Again, MPS does not have this limitation.

Polyglot, an extensible compiler framework for Java [58] also uses an extensible grammar formalism and parser to supports adding, modifying or removing productions and symbols defined in a base grammar. However, since Polyglot uses the LALR subset of context-free grammars, users must make sure *manually* that the base language and the extension remains LALR.

Monticore is another parser-based tool that generates parsers, metamodels, and editors based on extended grammar. Languages can extend each other and

can be embedded within each other [47]. An important idea is the ability to not regenerate the parsers or any of the related tools for a composed language. However, ambiguities have to be avoided manually.

Macro systems support defining additional syntax for existing languages. The new syntax is reduced in place to valid base language code. The definition of the syntax and the transformation is expressed with special host language constructs. Macro systems differ with regard to the degree of freedom they provide for the extension syntax, and whether they support extensions of type systems and IDEs. The most primitive macro system is the C preprocessor which performs pure text replacement during macro expansion. The Lisp macro system is more powerful because it is aware of the syntactic structure of Lisp (see Guy Steele's Growing a Language keynote [41]). An example of a macro system with limited syntactic freedom is the Java Syntactic Extender [4] where each macro has to begin with a unique keyword, and only a limited set of syntactic forms is supported. In OpenJava [69], the locations where macros can be added is limited. More fine-grained extensions, such as new operators, are not possible. Some of the C extensions developed in mbeddr are macro-style (they are reduced in place to the corresponding C code). However, MPS enforces no limitations on the granularity, syntax or location of such extensions, and supports extending the type system and the IDE.

In Section 4.3 we mentioned that MPS' template language provides IDE support for the target language *in the template*. In traditional text-generation template languages this is not possible because it requires support for language composition: the target language must be embedded in the template language. However, there are examples of template languages that support this, built on top of modular grammar formalisms. An example is the Repleo template language [2] which is built on SDF. However, as explained in the discussion on SDF above, SDF requires the definition of an additional grammar that defines how the host grammar (template language in this case) and the embedded grammar (target language) fit together (quotations). In MPS, any target language can be marked up with template annotations. No separate language has to be defined for the combination of template and target language.

Projectional Editing In this section we discuss other tools, that, like MPS, are based on a projectional editor. We focus on flexibility and in particular on usability, since we think that MPS is groundbreaking in this space.

An early example of a projectional editor is the Incremental Programming Environment (IPE, [53]). It provides a projectional editor and an integrated incremental compiler. It supports the definition of several notations for the same program (supported by MPS from late 2012) as well as partial projections. However, the projectional editor forces users to build the program tree top-down. For example, to enter `2+3`, users first have to enter the `+` and then fill in the two arguments. This is tedious and forces users to be aware of the language structure at all times. In contrast, as we have seen in Section 4.2, MPS supports editing that resembles text editing, particularly for expressions. IPE also does not address language modularity. In fact it comes with a fixed,

C-like language and does not have a built-in facility to define new languages. Another projectional system is GANDALF [57]. Its ALOEGEN component generates projectional editors from a language specification. It has the same usability problems as IPE. This is nicely expressed in [62]: *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.*

The Synthesizer Generator described in [65] also supports projectional editing. However, at the fine-grained expression level, textual input and parsing is used. This destroys many of the advantages of projectional editing in the first place, because simple language composition *at the expression level* is prohibited. We have seen in this paper that extensions of expressions are particularly important to tightly integrate an embedded language with its host language. MPS does not use this parsing "trick", and instead supports projectional editing also on expression level, with convenient editing gestures. The Intentional Domain Workbench [68] is another contemporary projectional editor that has been used in real projects. While not too much has been published about it, it is well-known that it supports mixing graphical, tabular and textual notations.

Modular Compilers As we have seen in this paper, language extension does not just include modular concrete syntax. It also requires the extension or composition of static semantics and transformations.

Many systems (including SILVER [76] mentioned above, JastAdd [34] and LISA [54]) describe static semantics using attribute grammars. These associate attributes with AST elements. An attribute can hold arbitrary data about the element (such as its type). Forwarding [78] is a mechanism that improves the modularity of attribute grammars by delegating the look-up of an attribute value to another element. While MPS' type system can be seen as associating a type attribute with AST elements using the `typeof` function, it is different from attribute grammars. Attribute values are calculated by *explicitly* referring to the values of other attributes, often recursively. MPS' type system rules are declarative. Developers specify typing rules for language concepts and MPS "instantiates" each rule for each AST element. A solver then solves all type equations in that AST. This way, the typing rules of elements contributed by language extensions can *implicitly* affect the overall typing of the program.

For language extension the execution semantics is usually defined by a transformation to the base language. In [76], van Wyk shows that this is valid only if the changes to the AST are local, avoiding unintended interactions between independently developed extensions used in the same program. In MPS such local changes are performed with reduction rules. In our experience it is also feasible to add additional elements to the AST *in select places*. In MPS, this is achieved using weaving rules. However, in both cases (local reduction and selective adding) there is no way to detect in advance whether using two extensions in the same program will conflict semantically or not.

As mentioned before, the Stratego [9] term rewriting-based transformation engine separates the transformations themselves from the orchestration of sets of transformations. The latter is achieved with several predefined strategies

that can be parametrized with the actual transformations. This way, the same transformations can be reused in different contexts. The facility can also be used to define the global order of independently developed transformations.

Extensible Tools and Language Workbenches While projectional tools always requires an IDE for editing programs, textual languages can be used with any text editor. We have already discussed modular languages above, and we discuss modular IDEs and full-blown language workbenches here.

Early examples include the Synthesizer Generator [65] (mentioned above) and the Meta Environment [45] which provides an editor for languages defined via ASF+SDF. Rascal [46] and Spoofax [43] provide Eclipse-based IDE support for SDF-based languages. In both cases the IDE support for composed languages is still limited (for example, at the time of this writing, Spoofax only provides syntax highlighting for an embedded language, but no code completion), but improving rapidly. To implement semantics, Rascal uses a Java-like language that has been extended with features for program construction, transformation and analysis. Spoofax uses term rewriting based on Stratego [9] which supports transformation composition based on higher-order strategies. An interesting tool is SugarJ [23] also based on SDF, which supports library based language extension (which can be seen as a sophisticated macro system). Spoofax-based IDE support is available as well [22].

LISA [54] (mentioned earlier) supports the definition of language syntax and semantics (via attribute grammars) in one integrated specification language. It then derives, among other things, a syntax-aware text editor for the language, as well as various graphical and structural viewing and editing facilities. Users can use inheritance and aspect-orientation to define extended languages. The use of this approach for incremental language development is detailed in [55]. However, users have to make sure manually that sub-grammars remain unambiguous with respect to the base grammar. The same is true for the combination of independently developed grammars.

Eclipse Xtext[28] generates sophisticated text editors from an EBNF-like language specification. Syntactic composition is limited since Xtext is based on ANTLR [61] which is a two phase LL(k) parser. It is possible for a language to extend *one* other language. Concepts from the base language can be used in the sub-language and it is possible to redefine grammar rules defined in the base language. Combination of independently defined extensions or Embedding is not supported. Xtext's abstract syntax is based on EMF Ecore[29], so it can be used together with any EMF-based model transformation and code generation tool (such as Xtend, Xpand, ATL, and Acceleo, all part of Eclipse Modeling[30]). Static semantics is based on constraints written in Java or on third-party frameworks that support declarative description of type systems such as XTS[31]

[28] `http://eclipse.org/Xtext`
[29] `http://eclipse.org/emf`
[30] `http://eclipse.org/modeling`
[31] `http://code.google.com/a/eclipselabs.org/p/xtext-typesystem`

or XSemantics[32]. Xtext comes with Xbase, an expression language that can be used as the basis for custom DSL [21]. Xbase also comes with a framework that simplifies the creation of interpreters and compilers for Xbase-based DSLs.

An interesting comparison can be made with the Renggli et al.'s Helvetia [64]. It supports language embedding and extension of Smalltalk using *homogeneous* extension, which means that the host language (Smalltalk) is also used for *defining* the extensions. In contrast to macro systems, it can embed languages with full-blown grammars. The authors argue that the approach is independent of the host language and could be used with other host languages as well. While this is true in principle, the implementation strategy heavily relies on the unique aspects of the Smalltalk system which are not available for other languages, and in particular, not for C. Also, since extensions are defined in the host language, the complete implementation would have to be redone if the approach were used with another language. This is particularly true for IDE support, where the Smalltalk IDE is extended using this IDE's APIs. mbeddr uses a *heterogeneous* approach which does not have these limitations: MPS provides a language-agnostic framework for language and IDE extension that can be used with any language, once the language is implemented in MPS.

Cedalion [18] is a host language for defining internal DSLs, based on a projectional editor and logic programming semantics. Both Cedalion and language workbenches such as MPS aim at combining the best of both worlds from internal DSLs (combination and extension of languages, integration with a host language) and external DSLs (static validation, IDE support, flexible syntax). Cedalion starts out from internal DSLs and adds static validation and projectional editing, the latter avoiding ambiguities resulting from combined syntaxes. Language workbenches start from external DSLs and add modularization, and, as a consequence of implementing GPLs with the same tool, optional tight integration with GPL host languages. We could not use Cedalion for mbeddr though, since we implemented our own base language (C), and the logic-based semantics would not have been a good fit.

An older line of work is focused on meta-CASE tools that aim at rapid development of CASE tools in order to support customised development methodologies [25]. They support specifying a meta-model and a typically *visual* notation, editors are then synthesized. Tools that implement this approach range from academic tools such as Pounamu [31], to industry quality tools based on Eclipse [30]. MetaEdit+ [71] is one of the most well known tools used in this space. It was and is used in several industry projects. The focus of mbeddr is different. We focus on mixed-notation languages and on the incremental extension of languages, general-purpose and domain-specific. This goes far beyond the creation of (often relatively high-level) graphical modeling languages.

Domain-Specific Tools based on Language Workbenches mbeddr is an example of instantiating a language workbench to build a domain-specific tool. While we believe that mbeddr is one of the largest and most sophisticated examples of this class of tools, it is not the only one. For example, WebDSL [72]

[32] `http://xsemantics.sourceforge.net`

is a set of DSLs for building (form-based) web applications based on Spoofax. Mobl [38] is a similar approach for mobile web applications. WebDSL in particular has proven to be useful for realistically-sized applications, as exemplified by the `researchr.org` website. In contrast to mbeddr, both WebDSL and Spoofax are not incremental extensions of a general purpose language, and they are based on a parser-based language workbench. The only other example that uses a projectional workbench is Intentional's Pension Workbench discussed in a presentation on InfoQ titled *Domain Expert DSLs*[33].

7 Summary, Conclusion and Future Work

Adequate tools can boost the productivity of software development, for example in the embedded software domain. Unfortunately, developers often have to live with off-the-shelf tools and ad-hoc tool chains, since, due to high costs, building customized tools for small and medium-sized organizations has been hard to justify and hence often not even considered as a realistic option. In this paper we investigate the usefulness of extending the languages that underlie such tools as a means of adapting the tool to particular domains. In particular, we presented our experience with instantiating the MPS language workbench for building the mbeddr stack in the field of embedded software. At first we introduced three important concepts of language workbenches: language modularity, projectional editing, and multi-stage transformations. We then presented the mbeddr stack from the perspective of the end-user by illustrating our solutions to four important challenges in embedded software engineering: separation of specification and implementation, formal verification, requirements traceability and product line variability. Next we discussed how the three pillars of language workbenches have been used to elegantly implement solutions for these challenges. Finally, we reviewed our experience in building mbeddr and in using it in the Smart Meter project. Regarding the questions we posed in the introduction, we conclude:

Q1 **It is feasible to build sophisticated domain-specific IDEs based on MPS as a representative example of projectional language workbenches.** mbeddr is a significantly large and complex system, and it has been implemented with very reasonable effort (10,000 lines of code for all of C and its IDE, implemented in four person months). Also, the efforts for incrementally extending the system for a particular project (as in the Smart Meter example) can easily be covered by such a project (a few days have been spent to develop Smart Meter-specific extensions so far). This shows that building domain-specific tools on top of language workbenches (as opposed to the current state of the art of developing such tools from scratch) is a very productive approach, in particular, because tools built this way can be extended in meaningful ways with reasonable effort.

Q2 **Language modularity works for realistically complex use cases.** mbeddr shows that it is feasible to build realistically complex extensions

[33] `http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk`

at various levels of granularity (down to expressions), addressing syntax, type systems, transformations and IDE support. As Smart Meter shows, independently developed extensions can be integrated. The amount of pre-planning for such extensions is comparable to the pre-planning in OO design. We have shown several cases where different extensions have been used together in the same program without pre-planning, and without conflicts.

Q3 **The approach leads to tools that are beneficial for real-world development.** Our preliminary experience with Smart Meter shows that the extensions lead to improvements in productivity as well as comprehensibility and maintainability. They help prevent and uncover bugs, and the runtime overhead incurred by the abstractions is acceptable. The integrated formal verifications have been used successfully to find non-trivial bugs as well. The feedback from those Smart Meter developers that have not been involved in mbeddr itself leads us to believe that, while projectional editing takes some time to get used to, the trade-off is worthwhile.

Summing up, we conclude that the approach discussed in this paper makes a significant contribution to improving the status quo with regard to building domain-specific development tools from scratch, or to support domain-specific extension of general-purpose tools.

Our future work involves providing better support for working with legacy code (one of the challenges of projectional editing), integrating more formal analyses (further exploiting the benefits of higher levels of abstraction) and running a larger project in the automotive industry to further evaluate the acceptance of the approach by end users. We will also make use of MPS' upcoming support for graphical notations for state machines and data flow diagrams. To validate the notion of building domain-specific tools on top of language workbenches more generally, mbeddr C could be extended for a domain other than embedded software, such as financial data analysis, 3D rendering or scientific data processing. We are currently looking for partners who would be interested in trying mbeddr in one of these domains.

Acknowledgements We thank Marcel Matzat and Domenik Pavletic for their work on mbeddr. mbeddr has been supported by the German BMBF, FKZ 01/S11014.

References

1. Andalam, S., Roop, P., Girault, A., Traulsen, C.: PRET-C: A new language for programming precision timed architectures. In: Proocedings of the Workshop on Reconciling Performace with Predictability (RePP), Embedded Systems Week (2009)
2. Arnoldus, J., Bijpost, J., van den Brand, M.: Repleo: a syntax-safe template engine. In: Proc. of the 6th ACM Int. Conference on Generative Programming and Component Engineering (GPCE 2007)
3. Axelsson, E., Claessen, K., Devai, G., Horvath, Z., Keijzer, K., Lyckegard, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: MEMOCODE 2010
4. Bachrach, J., Playford, K.: The Java syntactic extender. In: Proceedings 16th conference on Object-oriented programming, systems, languages, and applications, OOPSLA '01
5. Ben-Asher, Y., Feitelson, D.G., Rudolph, L.: ParC - An Extension of C for Shared Memory Parallel Processing. Software: Practice and Experience **26**(5) (1996)

6. Beuche, D., Papajewski, H., Schroeder-Preikschat, W.: Variability management with feature models. Science of Computer Programming **53**(3) (2004)
7. Birk, A., Heller, G., John, I., Schmid, K., von der Massen, T., Mueller, K.: Product Line Engineering: The State of the Practice. IEEE Software **20**(6) (2003)
8. Boussinot, F.: Reactive C: An Extension of C to Program Reactive Systems. Software: Practice and Experience **21**(4) (1991)
9. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation
10. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proc. of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004
11. Bravenboer, M., Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: MoDELS 2007, *LNCS*, vol. 5002. Springer (2007)
12. Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D.: Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. Proc. of the IEEE **98**(4) (2010)
13. Broy, M., Kirstan, S., Krcmar, H., Schätz, B.: What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In: Emerging Technologies for the Evolution and Maintenance of Software Models. ICI (2011)
14. Clarke, E.: Model checking. In: Foundations of Software Technology and Theoretical Computer Science, *LNCS*, vol. 1346, pp. 54–56 (1997)
15. Clarke, E.M., Heinle, W.: Modular Translation of Statecharts to SMV. Tech. rep., Carnegie Mellon University (2000)
16. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Proc. of the Int. Conference of Software Engineering (ICSE) (2000)
17. Damm, W., Achatz, R., Beetz, K., Broy, M., Dämbkes, H., Grimm, K., Liggesmeyer, P.: Nationale Roadmap Embedded Systems. Springer (2010)
18. David H. Lorenz, Boaz Rosenan: Cedalion: A Language for Language Oriented Programming. In: Proc. of OOPSLA/SPLASH 2011 (2011)
19. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: Proc. of the 4th Int. Conference on Embedded networked sensor systems, SenSys 2006. ACM
20. Ebert, C., Jones, C.: Embedded Software: Facts, Figures, and Future. Computer **42**(4) (2009)
21. Efftinge, S., Eysholdt, M., Khnlein, J., Zarnekow, S., Hasselbring, W., von Massow, R., Hanus, M.: Xbase: Implementing DSLs for Java. In: Proc. of the 11th Conference on Generative Programming and Component Engineering (GPCE 2012)
22. Erdweg, S., Kats, L.C.L., Kastner, C., Ostermann, K., Visser, E.: Growing a Language Environment with Editor Libraries. In: Proc. of the 10th ACM Int. Conference on Generative programming and component engineering (GPCE 2011). ACM (2011)
23. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: library-based syntactic language extensibility. In: OOPSLA 2011. ACM
24. Ernst, M.D., Badros, G.J., Notkin, D.: An Empirical Analysis of C Preprocessor Use. IEEE Trans. Softw. Eng. **28** (2002)
25. Ferguson, R., Parrington, N., Dunne, P., Hardy, C., Archibald, J., Thompson, J.: MetaMOOSE - an object-oriented framework for the construction of CASE tools. Information and Software Technology **42**(2) (2000)
26. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html (2005)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
28. Gokhale, A.S., Balasubramanian, K., Krishna, A.S., Balasubramanian, J., Edwards, G., Deng, G., Turkay, E., Parsons, J., Schmidt, D.C.: Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. Science of Computer Programming **73**(1) (2008)
29. Graaf, B., Lormans, M., Toetenel, H.: Embedded Software Engineering: The State of the Practice. IEEE Softw. **20**(6) (2003)

30. Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Model-Based Customization and Deployment of Eclipse-Based Tools: Industrial Experiences. In: Proc. of the 2009 Int. Conference on Automated Software Engineering. IEEE CS
31. Grundy, J., Hosking, J.: Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool. In: Proc. of the 29th Int. Conference on Software Engineering, ICSE '07. IEEE Computer Society (2007)
32. Hammond, K., Michaelson, G.: Hume: a domain-specific language for real-time embedded systems. In: Proc. of GPCE 2003
33. von Hanxleden, R.: SyncCharts in C - A Proposal for Light-Weight, Deterministic Concurrency. In: Proc. of the Int. Conference on Embedded Sofware (EMSOFT'09)
34. Hedin, G., Magnusson, E.: JastAdd–an aspect-oriented compiler construction system. Science of Computer Programming **47**(1) (2003)
35. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN **24**(11) (1989)
36. Heidenreich, F., Wende, C.: Bridging the Gap Between Features and Models. In: 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07)
37. Heitmeyer, C.: Developing Safety-Critical Systems: the Role of Formal Methods and Tools. In: Australian Computer Society, Inc (2006)
38. Hemel, Z., Visser, E.: Declaratively programming the mobile web with mobl. In: Proc. of the 2011 ACM Int. Conference on Object oriented programming systems languages and applications, OOPSLA 2011, pp. 695–712. ACM (2011)
39. Janicki, R., Parnas, D.L., Zucker, J.: Tabular representations in relational documents. Springer-Verlag New York, Inc. (1997)
40. Jirapanthong, W., Zisman, A.: Supporting Product Line Development through Traceability. In: 12th Asia-Pacific Software Engineering Conference (APSEC '05). IEEE CS
41. Jr., G.L.S.: Growing a Language. Higher-Order and Symbolic Computation **12**(3) (1999)
42. Kästner, C.: CIDE: Decomposing Legacy Applications into Features. In: 11th Int. Conference on Software Product Lines (SPLC 2007), Proceedings. Vol 2, Workshops
43. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proc. of the 25th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010
44. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Proc. of the 25th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010. ACM
45. Klint, P.: A Meta-Environment for Generating Programming Environments. ACM Transactions on Software Engineering Methodology **2**(2) (1993)
46. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009. IEEE Computer Society
47. Krahn, H., Rumpe, B., Vlkel, S.: MontiCore: a framework for compositional development of domain specific languages. STTT **12**(5), 353–372 (2010)
48. Liggesmeyer, P., Trapp, M.: Trends in Embedded Software Engineering. IEEE Softw. **26** (2009)
49. Loer, K., Harrison, M.: Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems. In: In Proc. of the Int. Conference on Automatic Software Engineering (ASE) (2002)
50. M. Bravenboer and E. Dolstra and E. Visser: Preventing injection attacks with syntax embeddings. In: Generative Programming and Component Engineering, 6th International Conference, GPCE 2007. ACM (2007)
51. Mali, Y., Wyk, E.V.: Building Extensible Specifications and Implementations of Promela with AbleP. In: Model Checking Software - 18th International SPIN Workshop, Proceedings, *LNCS*, vol. 6823. Springer (2011)
52. Marche, C., Moy, Y.: The jessie plugin for deduction verification in frama-c - tutorial and reference manual - version 2.30. Tech. rep., INRIA (2012)
53. Medina-Mora, R., Feiler, P.H.: An Incremental Programming Environment. IEEE Trans. Software Eng. **7**(5) (1981)
54. Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Compiler Construction, 11th Int. Conference, ETAPS/CC 2002, Proceedings, *LNCS*, vol. 2304. Springer (2002)

55. Mernik, M., Zumer, V.: Incremental programming language development. Computer Languages, Systems & Structures **31**(1) (2005)
56. Meyer, B.: Design by Contract: The Eiffel Method. In: TOOLS 1998: 26th Int. Conference on Technology of Object-Oriented Languages and Systems, p. 446. IEEE CS
57. Notkin, D.: The GANDALF project. Journal of Systems and Software **5**(2) (1985)
58. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: 12th Int. Conference on Compiler Construction, ETAPS/CC 2003, *LNCS*, vol. 2622. Springer
59. Palopoli, L., Ancilotti, P., Buttazzo, G.C.: A C Language Extension for Programming Real-Time Applications. In: 6th Int. Workshop on Real-Time Computing and Applications (RTCSA 99). IEEE CS
60. Parnas, D.: Really Rethinking 'Formal Methods'. IEEE Computer **43**(1) (2010)
61. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software: Practice and Experience **25**(7) (1995)
62. Porter, S.W.: Master's thesis, Naval Postgraduate School, Monterey, CA, USA (1988)
63. Ratiu, D., Voelter, M., Schaetz, B., Kolb, B.: Language Engineering as Enabler for Incrementally Defined Formal Analyses. In: Proc. of the Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FORMSERA'2012)
64. Renggli, L., Girba, T., Nierstrasz, O.: Embedding Languages Without Breaking Tools. In: European Conference on Object Orient Programming, ECOOP 2010
65. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator. In: First ACM SIGSOFT-/SIGPLAN software engineering symposium on Practical software development environments. ACM (1984)
66. S. Andalam P. S. Roop, A.G.: Predictable multithreading of embedded applications using PRET-C. In: Proc. of ACM-IEEE Int. Conference on Formal Methods and Models for Codesign (MEMOCODE), (2010)
67. Sheini, H., Sakallah, K.: From Propositional Satisfiability to Satisfiability Modulo Theories. In: Theory and Applications of Satisfiability Testing - SAT 2006, *LNCS*, vol. 4121, pp. 1–9. Springer Berlin / Heidelberg (2006)
68. Simonyi, C., Christerson, M., Clifford, S.: Intentional Software. In: Proc. of the 21st ACM conference companion on Object oriented programming systems languages and applications, OOPSLA '06
69. Tatsubori, M., Chiba, S., Itano, K., Killijian, M.O.: OpenJava: A Class-Based Macro System for Java. In: Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA '99 Workshop on Reflection and Software Engineering, *LNCS*, vol. 1826
70. Thomas, D., Hunt, A.: Mock Objects. IEEE Software **19**(3), 22–24 (2002)
71. Tolvanen, J.P., Kelly, S.: MetaEdit+: defining and using integrated domain-specific modeling languages. In: Proc. of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, OOPSLA '09
72. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, *Lecture Notes in Computer Science*, vol. 5235, pp. 291–373
73. Voelter, M.: Embedded Software Development with Projectional Language Workbenches. In: Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Proceedings, LNCS. Springer
74. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: 4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011), LNCS. Springer (2011)
75. Voelter, M., Ratiu, D., Schaetz, B., Kolb, B.: mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In: Proc. of SPLASH 2012
76. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. ENTCS **203**(2) (2008)
77. Wyk, E.V., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. In: ECOOP 2007 - 21st European Conference on Object-Oriented Programming, Proceedings, *LNCS*, vol. 4609. Springer (2007)
78. Wyk, E.V., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: Proc. of the 11th Int. Conference on Compiler Construction (ETAPS/CC 2002), *LNCS*, vol. 2304