mbeddr: an Extensible MPS-based Programming Language and IDE for Embedded Systems

Markus Voelter
independent/itemis
voelter@acm.org

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

Bernd Kolb
itemis
kolb@itemis.de

Abstract

While the C programming language provides very good support for writing efficient, low-level code, it does not offer adequate means for defining higher-level abstractions relevant to embedded software. In this article we present the MPS-based mbeddr technology stack that supports extension of C with constructs adequate for embedded systems. In mbeddr, efficient low-level programs can be written using the well-known concepts from C. Higher-level domain-specific abstractions can be seamlessly integrated into C by means of modular language extension regarding syntax, type system, semantics and IDE. In the article we show how language extension can address the challenges of embedded software development and report on our experience in building these extensions with MPS. We show that MPS delivers on the promise of significantly reducing the effort of language engineering and the construction of corresponding IDEs. Both MPS and mbeddr are open source software.

1. Challenges in Embedded Software

In this section we discuss a set of challenges we address with the mbeddr approach. We label the challenges C_n so we can refer to them from Section 2.2 where we show how they are addressed by mbeddr C. While these are certainly not *all* challenges embedded software developers face, based on our experience with embedded software and feedback from various domains (automotive, sensors, automation) and organizations (small, medium and large companies), these are among the most important ones.

C_1: Abstraction without Runtime Cost Domain-specific concepts provide more abstract descriptions of the system under development. Examples include data flow blocks, state machines, or data types with physical units. On the one hand, adequate abstractions have a higher expressive power that leads to shorter and easier to understand and maintain programs. On the other hand, by restricting the freedom of programmers, domain specific abstractions also enable constructive quality assurance. For embedded systems, where runtime efficiency is a prime concern, abstraction mechanisms are needed that can be resolved before or during compilation, and not at runtime.

C_2: C considered Unsafe While C is efficient and flexible, several of C's features are often considered unsafe. For example, unconstrained casting via `void` pointers, using `int`s as Booleans or the weak typing implied by `union`s can result in runtime errors that are hard to track down. Consequently, the unsafe features of C are prohibited in many organizations. Standards for automotive software development such as MISRA limit C to a *safer* language subset. However, most C IDEs are not aware of these and other, organization-specific restrictions, so they are enforced with separate checkers that are often not well integrated with the IDE. This makes it hard for developers to comply with these restrictions efficiently.

C_3: Program annotations For reasons such as safety or efficiency, embedded systems often require additional data to be associated with program elements. Examples include physical units, coordinate systems, data encodings or value ranges for variables. These annotations are typically used by specific, often custom-built analysis or generation tools. Since C programs can only capture such data informally as comments or `pragma`s, the C type system and IDE cannot check their correct use in C programs. They may also be stored separately (for example, in XML files) and linked back to the program using names or other weak links.

Even with tool support that checks the consistency of these links and helps navigate between code and this additional data, the separation of core functionality and the additional data leads to unnecessary complexity and maintainability problems.

C_4: Static Checks and Verification Embedded systems often have to fulfil strict safety requirements. Industry standards for safety (such as ISO-26262, DO-178B or IEC-61508) demand that for high safety certification levels various forms of static analyses are performed on the software. These range from simple type checks to sophisticated property checks, for example by model checking. Since C is a very flexible and relatively weakly-typed language, the more sophisticated analyses are very expensive. Using suitable domain-specific abstractions (for example, state machines) leads to programs that can be analyzed much more easily.

C_5: Process Support There are at least two cross-cutting and process-related concerns relevant to embedded software development. First, many certification standards (such as those mentioned above) require that code be explicitly linked to requirements such that full traceability is available. Today, requirements are often managed in external tools and maintaining traceability to the code is a burden to the developers and often done in an ad hoc way, for example via comments. Second, many embedded systems are developed as part of product lines with many distinct product variants, where each variant consists of a subset of the (parts of) artifacts that comprise the product line. This variability is usually captured in constraints expressed over program parts such as statements, functions or states. Most existing tools come with their own variation mechanism, if variability is supported at all. Integration between program parts, the constraints and the variant configuration (for example via feature models) is often done through weak links, and with little awareness of the semantics of the underlying language. For example, the C preprocessor, which is often used for this task, performs simple text replacement or removal controlled by the conditions in `#ifdef`s. As a consequence, variant management is a huge source of accidental complexity.

An additional concern is tool integration. The diverse requirements and limitations of C discussed so far often lead to the use of a wide variety of tools in a single development project. Most commercial off-the-shelf (COTS) tools are not open enough to facilitate seamless and semantically meaningful integration with other tools, leading to significant accidental tool integration complexity. COTS tools often also do not support meaningful language extension, severely limiting the ability to define and use custom domain-specific abstractions. By building mbeddr onto MPS, users of mbeddr get access to the same powerful means of building languages and language extensions as the original mbeddr developers. Users who want to extend the tool and the languages are not second-class citizens. This is a game changer.

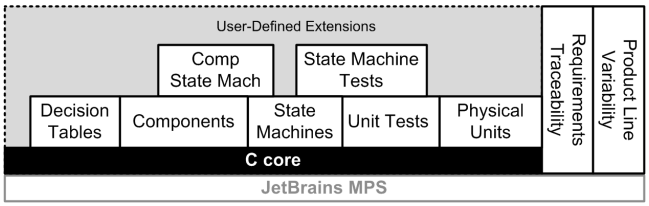

Figure 1. Based on MPS, mbeddr comes with an implementation of the C programming language. On top of C mbeddr defines a set of default extensions (white boxes) stacked on top of each other. Users can use them in their programs, but they don't have to. Support for requirements traceability and product line variability is cross-cutting. Users build their own extensions on top of C or on top of the default extensions. (Note: component/state machine integration and state machine tests are not discussed in this paper.)

2. The mbeddr Approach

Language engineering provides a holistic approach to solve these challenges. In this section we illustrate how mbeddr addresses the challenges with an extensible version of the C programming language, growing a stack of languages extensions (see Fig. 1).

2.1 Language Extension

Traditionally, languages are composed by *referencing*: The partial programs expressed with different languages reside in their own files and refer to each other via references, often using qualified names. There is no *syntactic* integration, where a single program file contains language constructs defined in different languages. While referencing is sometimes useful, syntactic integration is required in many cases, as we will see in the examples provided in Section 2.2. There are two composition approaches that support syntactic integration, both of them supported by MPS: language *embedding* refers to the syntactic composition of two independent languages. The embedded language has no dependency on the host language. Both have been developed independently, and the act of embedding does not require changes to either language. In language *extension*, a dependency from the extending language to the base language is allowed, for example, by inheriting from language concepts defined in the base language. The mbeddr system relies primarily on language extension.

To make language extension useful, it must provide deep syntactic and semantic integration, as well as an

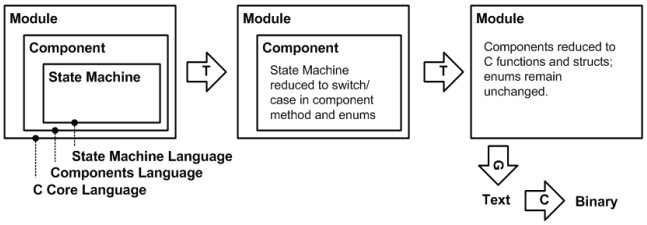

Figure 2. Higher-level abstractions such as state machines or components are reduced incrementally to their lower-level equivalent, reusing the transformations built for lower-level extensions. Eventually, C text is generated which is subsequently compiled with a C compiler suitable for the target platform.

Challenge	Example Extensions
C_1 (Low-Overhead Abstraction)	State machines Components Decision Tables
C_2 (Safer C)	Cleaned up C Safe Modules
C_3 (Annotations)	Physical Units
C_4 (Static Checks, Verification)	Unit Tests State Machines Safe Modules
C_5 (Process Support)	Requirements Traceability Product Line Variability

Figure 3. Embedded software development challenges and the example extensions in this section

IDE that is aware of the language extensions. It is much more than a macro system or an open compiler. MPS supports the flexible definition, extension, composition and use of multiple languages. A language extension defines new structure, syntax, type system rules and semantics, as well, as optionally, support for refactoring, quick fixes and debugging. The semantics of an extension are typically defined by a transformation back to the base language. For example, in an extension that provides state machines, these may be transformed to a `switch/case`-based implementation in C. Extensions can be stacked (Fig. 1), where a higher-level extension extends (and transforms back to) a lower-level extension instead of C. At the bottom of this stack resides plain C in textual form and a suitable compiler. Fig. 2 shows an example where a module containing a component containing a state machine is transformed to C, and then compiled.

A set of organizations, such as the departments in a large company, will likely not agree on a *single* set of extensions to C since they typically work in slightly different areas. Also, a language that contains *all* relevant abstractions would become big and unwieldy. Thus, extensions have to be *modular*. They have to be defined independent of each other, without modifying the base language, and unintended interactions between independently created extensions must be avoided (a discussion of automatic detection of interactions is beyond the scope of this paper). Also, users must be able to include *incrementally* only those extensions into any given program they actually need. Ideally, they should be able to do this without requiring the definition of a "combined language" for each combination of used extensions: for example, a user should be able to include an extension providing state machines and an extension providing physical units *in the same program* without first defining a combined language statemachine-with-units.

2.2 Extensions addressing the Challenges

In this section we present example extensions that illustrate how we address the challenges discussed in Section 1. We show at least one example for each challenge. Our aim in this paper is to showcase the extensibility of the mbeddr system, and, by this, language engineering using language workbenches. We will not discuss in detail any particular extension. The table in Fig. 3 shows an overview over the challenges, the examples in this section, and the ways of extension each example makes use of.

A cleaned up C (addresses C_2) To make C extensible, we first had to implement C in MPS. This entails the definition of the language structure, syntax and type system[1]. In the process we changed some aspects of C. Some of these changes are a first step in providing a safer C (C_2). Others changes were implemented because it is more convenient to the user or because it simplified the implementation of the language in MPS. Out of eight changes total, four are for reasons of improved robustness and analyzability, two are for end user convenience and three are to simplify the implementation in MPS. We discuss some of them below.

mbeddr C provides *modules* (Fig. 4). A module contains the top level C constructs (such as `structs`, functions or global variables). These module contents can be `exported`. Modules can *import* other modules, in which case they can access the exported contents of the imported modules. While header files are generated, we do not expose them to the user: modules provide a more convenient means of controlling modularizing programs and limiting which elements are visible globally.

[1] A generator to C text is also required, so the code can be fed into an existing compiler. However, since this generator merely renders the tree as text, with no structural differences, this generator is trivial. We do not discuss it any further

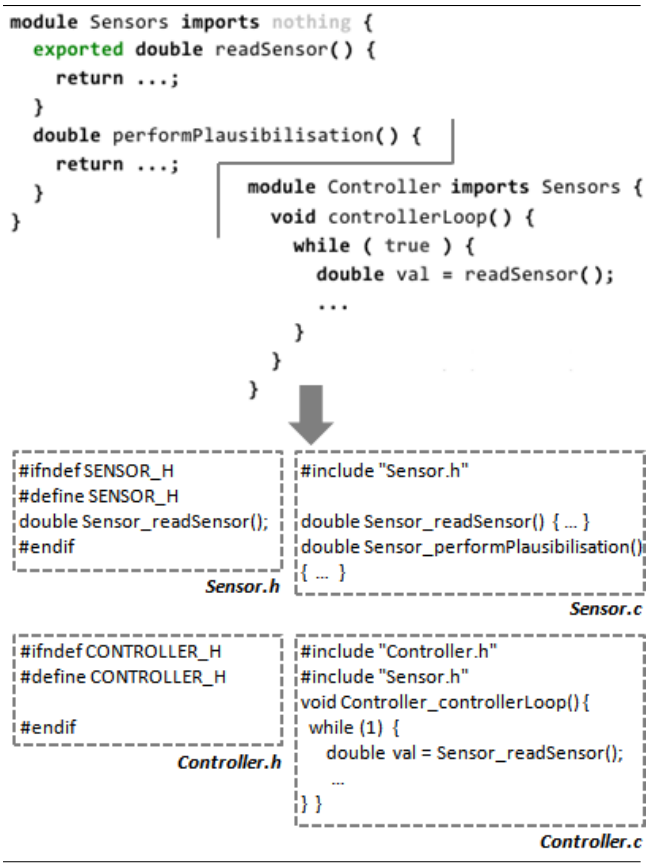

Figure 4. Modules are the top-level container in mbeddr C. They can import other modules, whose exported contents they can then use. Exported contents are put into the header files generated from modules.

mbeddr C does not support the *preprocessor* because it is often used to emulate missing features of C in ad-hoc way, leading to problems regarding maintenance and analyzability. Instead, mbeddr C provides first class support for the most important use cases of the preprocessor. Examples include the modules mentioned above (replacing `#include`) as well as the support for variability discussed below (replacing `#ifdefs`). Instead of defining macros, users can create first-class language extensions including type checks and IDE support. Removing the preprocessor and providing specific support for its important use cases goes a long way in creating more maintainable and more analyzable programs. The same is true for introducing a separate `boolean` type and not interpreting integers as Booleans by default. An explicit cast operator is available.

Type decorations, such as array brackets or the pointer asterisk must be specified on the type, not on the identifier (`int[] a;` instead of `int a[];`). This has been done for reasons of consistency and to simplify the implementation in MPS: it is the property of a type to be an array type or a pointer type, not the property of an identifier. Identifiers are just names.

Decision Tables (addressing C_1) are a new kind of expression, i.e. they can be evaluated. An example is shown in Fig. 5. A decision table represents nested `if` statements. It is evaluated to the value of the first cell whose column and row headers are `true` (the evaluation order is left to right, top to bottom). A default value (`FAIL`) is specified to handle the case where none of the column/row header combinations is `true`. Since the compiler and IDE have to compute a type for expressions, the decision table specifies the type of its result values explicitly (`int8_t`).

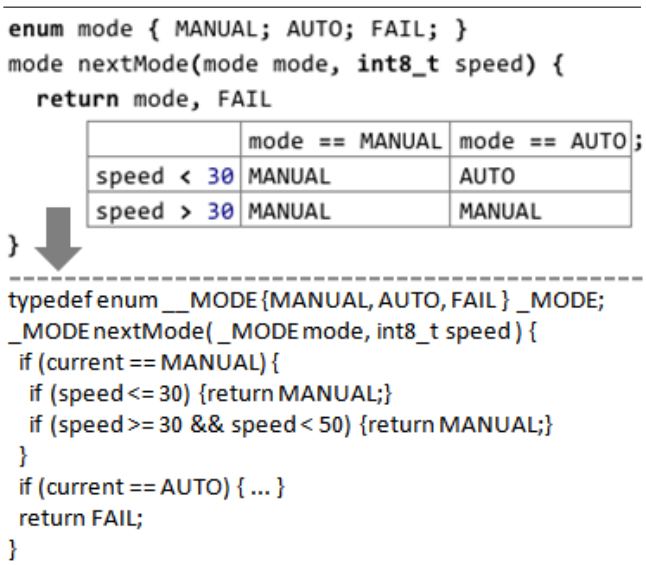

Figure 5. A decision table evaluates to the value in the cell for which the row and column headers are `true`, a default value otherwise (`FAIL` in the example). By default, a decision table is translated to nested `if`s in a separate function. The figure shows the translation for the common case where a decision table is used in a `return`. This case is optimized to not use the indirection of an extra function.

Unit Tests (addresses C_4) are new top-level constructs (Fig. 6) introduced in a separate *unittest* language that extends the C core. They are like `void` functions without arguments. The *unittest* language also introduces `assert` and `fail` statements, which can only be used inside test cases. Testing embedded software can be a challenge, and the *unittest* extension is a first step at providing comprehensive support for testing. mbeddr also provides support for platform-independent logging as well as for specifying stubs and mocks. We do not discuss this in this paper.

Components (addresses C_1) are new top level constructs that support modularization, encapsulation and

```
module UnitTestDemo imports Sensors {
  exported test case sensorReadTest {
    assert(0) readSensor() > 0;
    assert(1) readSensor() < 1000;
  }
}
```

```
#include "Sensor.h"
int8_t UnitTestDemo_test_sensorReadTest() {
 int8_t __failures = 0;
 printf("running test @UnitTestDemo:test_sensorReadTest:0\n");
 if ( !(Sensor_readSensor() > 0) ) {
  __failures++;
  printf("FAILED: @UnitTestDemo:test_sensorReadTest:1\n");
  printf("  testID = %d\n",0);
 }
 if ( !(Sensor_readSensor() < 1000) ) { ... }
 return __failures;
}
```

Figure 6. The *unittest* language introduces test cases as well as `assert` and `fail` statements which can only be used inside of a test case. Test cases are transformed to functions, and the `assert` statements become `if` statements with a negated condition. The generated code also counts the number of failures so it can be reported to the user via a binary's exit value.

the separation between specification and implementation (Fig. 7). In contrast to modules, a component uses interfaces and ports to declare the contract it obeys. Interfaces define operation signatures and optional pre and post conditions (not shown in the example). Provided ports declare the interfaces offered by a component, required ports specify the interfaces a component expects to use. Different components can implement the same interface differently. Components can be instantiated (also in contrast to modules), and each instance's required ports have to be connected to compatible provided ports provided by other component instances. Polymorhphic invocations (different components "behind" the same interface) are supported.

State Machines (addresses C_1, C_4) provide a new top level construct (the state machine itself) as well as a `trigger` statement to send events into state machines (see Fig. 8). State machines are transformed into a `switch/case`-based implementation in the C program. Entry, exit and transition actions may only access variables defined locally in state machines and fire out events. Out events may optionally be mapped to functions in the surrounding C program, where arbitrary behaviour can be implemented. This way, state machines are semantically isolated from the rest of the code, enabling them to be model checked: if a state machine is marked as `verifiable`, we also generate a represen-

```
module SensorComp imports Sensors , LoggingService {
  exported c/s interface SensorAccess {
    double readValue()
  }
  exported component SimpleSensor extends nothing {
    ports:
      provides SensorAccess sensor
    contents:
      double read() <- op sensor.readValue {
        return readSensor();
  }   }
  exported component PlausiSensor extends nothing {
    ports:
      provides SensorAccess sensor
      requires LoggingService log
    contents:
      double read() <- op sensor.readValue {
        double val = readSensor();
        if ( val > 100 ) {
          log.info("Sensor value unexpected big");
          return 100;
        } if
        return val;
} }   }
```

Sensors.h

```
struct Sensors_compdata_SimpleSensor {};
double Sensors_SimpleSensor_read(void* inst_data);

struct Sensors_data_PlausiSensor {
 void* port_log;
 void (*op_log_info)(char*, void*);
}
double Sensors_PlausiSensor_read(void* inst_data);
```

Sensors.c

```
#include "Sensors.h"
#include "Sensor.h"
#include "LoggingService.h"

double Sensors_SimpleSensor_read(void* inst_data) {
 return Sensor_readSensor();
}

double Sensors_PlausiSensor_read(void* inst_data) {
 double val = Sensor_readSensor();
 if (val > 100) {
  (*((struct Sensors_data_PlausiSensor*)inst_data)->op_log_info)
    ("Sensor value unexpected big",
    ((struct Sensors_data_PlausiSensor*)inst_data)->port_log);
  return 100;
 }
 return val;
}
```

Figure 7. Two components providing the same interface. The arrow maps operations from provided ports to implementations. An indirection through function pointers enables different implementations for a single interface, enabling OO-like polymorphic invocations.

tation of the state machine in the input language of the NuSMV model checker[2], including a set of property specifications that are verified by default. Examples include dead state detection, dead transition detection, non-determinism and variable bounds checks. In addition, users can specify additional high-level properties based on the well-established catalog of temporal logic properties patterns.

```
derived unit mps = m s^-1 for speed
convertible unit kmh for speed
conversion kmh -> mps = val * 0.27

int8_t/mps/ calculateSpeed(int8_t/m/ length, int8_t/s/ time) {
  int8_t/mps/ s = length / time;
  if ( s > 100 mps ) { s = [100 kmh → mps]; }
  return s;
}
```

Figure 9. The *units* extension ships with the SI base units. Users can define derived units (such as the `mps` in the example) as well as convertible units that require a numeric conversion for mapping back to SI units. Type checks ensure that the values associated with unit literals use the correct unit and perform unit computations (as in speed equals length divided by time). Errors are reported if incompatible units are used together (e.g. if we were to add length and time). To support this feature, the typing rules for the existing operators (such as `+` or `/`) have to be overridden.

Physical Units (addresses C_3) are new types that also specify a physical unit in addition to their actual data type (see Fig. 9). New literals are introduced to support specifying values for these types that include the physical unit. The typing rules for the existing operators (`+`, `*` or `>`) are overridden to perform the correct type checks for types with units. The type system also performs unit computations to deal correctly with `speed = length/time`, for example.

Requirements Traces (addresses C_5) are meta data annotations that link a program element to requirements, essentially elements in other models imported from requirements management tools. Requirements traces can be attached to any program element without that element's definition having to be aware of this (see green highlights in Fig. 10).

Presence Conditions (addresses C_5) A presence condition determines whether the program element to which it is attached is part of a product in the product line. A product is configured by specifying a set of configuration flags and the presence condition specifies a Boolean expression over these configuration switches[3].

[2] http://nusmv.fbk.eu

[3] We use feature models to express product configurations, and the presence conditions are expressions over features. But this aspect is not essential to the discussion here.

Like requirements traces, presence conditions can be attached to any program element. For example, in Fig. 10, the `resetted` out event and the `on start...` transition in the second state have the `resettable` presence condition, where `resettable` is a reference to a configuration flag. Upon transformation, program elements whose presence condition evaluates to `false` for a particular product configuration are simply removed from the program (and hence will not end up in the generated binary). This program customization can also be performed by the editor, effectively supporting variant-specific editing.

Safe Modules (addresses C_2) Safe modules help prevent writing risky code. For example, runtime range checking is performed for arithmetic expressions and assignments. To enable this, arithmetic expressions are replaced by function calls that perform range checking and report errors if an overflow is detected. As another example, safe modules also provide the `safeheap` statement that automatically frees dynamic variables allocated inside its body (see Fig. **??**).

2.3 Addressing the Tool Integration Challenge

We have not highlighted tool integration as an explicit challenge, because it is a cross-cutting issue that affects all of the challenges above. Nonetheless, in a project intended to be used by practitioners, this needs to be addressed. We do that by providing an *integrated environment* that provides state-of-the-art IDE support for C and all of its extensions. This includes syntax highlighting, code completion, static error checking and annotation, quick fixes and refactorings. Fig. 10 shows a screenshot of the tool, as we edit a module with a decision table, a state machine, requirements traces and presence conditions.

3. Experiences

In this section we provide a brief overview over our experiences in implementing mbeddr based on MPS, including the size of the project and the efforts spent.

3.1 Language Extension

Size Typically, lines of code are used to describe the size of a software system. In MPS, a "line" is not necessarily meaningful. Instead we count important elements of the implementation and then estimate a corresponding number of lines of code. Fig. 11 shows the respective numbers for the core, i.e. C itself plus unit test support, decision tables and build/make integration (the table also shows how many LOC equivalent we assume for each language definition element, and the caption explains to some extent the rationale for these factors). According to our metric the C core is implemented with less than 10,000 lines of code.

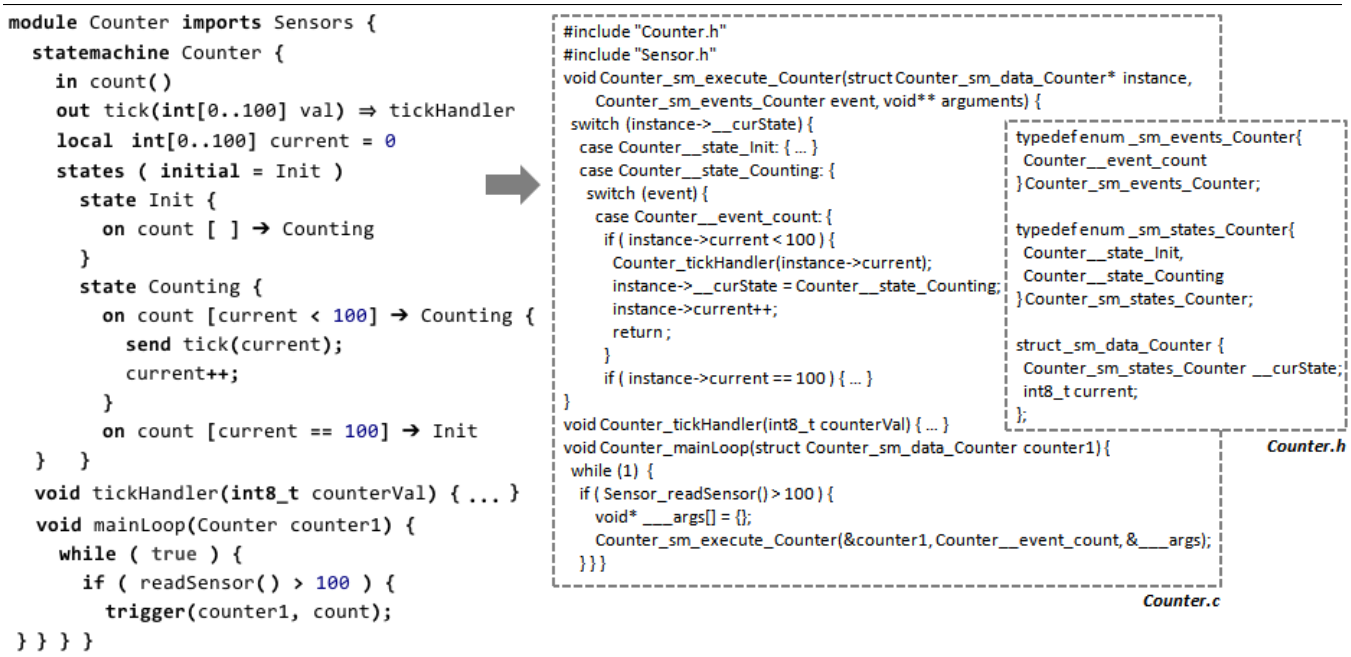

Figure 8. A state machine is embedded in a C module as a top level construct. It declares `in` and `out events` as well as local variables, states and transitions. Transitions react to `in events`, and `out events` can be fired in actions. Through bindings (e.g. `tickHandler`), state machines interact with C code. State machines can be instantiated. They are transformed to `enum`s for states and events, and a function that executed the state machine using `switch` statements. The `trigger` statement injects events into a state machine instance by calling the state machine function.

Let us look at an incremental extension of C. The components extension (interfaces, components, pre and post conditions, support for mock components in testing and a generator back to plain C) is ca. 3,000 LOC equivalent. The state machines extension is ca. 1,000. Considering the fact that these LOC equivalents represent the language definition (incl. type systems and generators) and the IDE (incl. code completion, syntax coloring, some quick fixes and refactorings), this clearly speaks to the efficiency of MPS for language development and extension.

Effort In terms of effort, the core C implementation has been ca. 4 person months divided between three people. This results in roughly 2,500 lines of code per person month. Extrapolated to a year, this would be 7,500 lines of code per developer. According to McConnell[4], in a project up to 10,000 LOC, a developer can typically do between 2,000 and 25,000 LOC. The fact that we are at the low end of this range can be explained by the fact that MPS provides very expressive languages for DSL development: you don't have to write a lot of code to express a lot about a DSL. Instead, MPS code is relatively dense and requires quite a bit of thought. Pair programming is very valuable in language development.

Once a developer has mastered the learning curve, language extension can be very productive. The state machines and components extension have both been developed in about a month. The unit testing extension or the support for decision tables can be implemented in a few days.

Language Modularity, Reuse and Growth Modularity and composition is central to mbeddr.

Building a language extension should not require changes to the base languages. This requires that the extended languages are built with extension in mind. Just like in object-oriented programming, where the only methods can be overridden, only specific parts of a language definition can be extended or overwritten. The implementation of the default extensions served as a test case to confirm that the C core language is in fact extensible. We found a few problems, especially in the type system and fixed them. None of these fixes were "hacks" to enable a specific extension — they were all genuine mistakes in the design of the C core. Due to the broad spectrum covered by our extensions, we are confident that the current core language provides a high degree of extensibility.

Independently developed extensions should not interact with each other in unexpected ways. While MPS

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

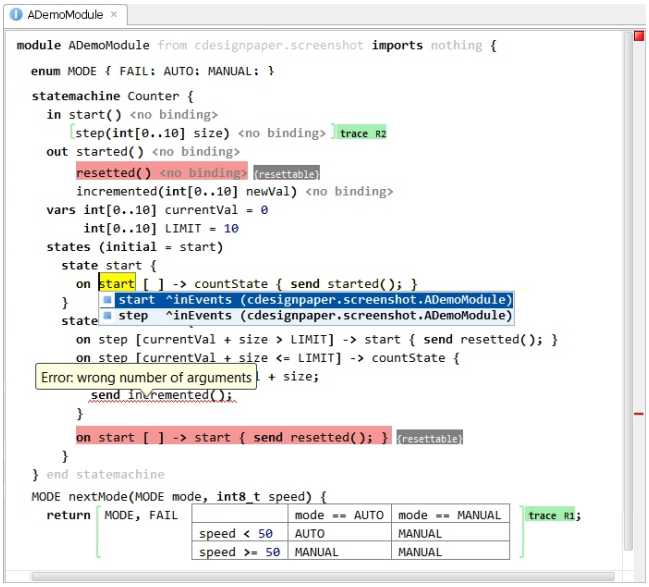

Figure 10. A somewhat overloaded example program in the mbeddr IDE (an instance of MPS). The module contains an `enum`, a decision table and a state machine. Requirements traces are attached to the table and the `step` in event, and a presence condition is attached to an out event and a transitions

Element	Count	LOC-Factor
Language Concepts	260	3
Property Declarations	47	1
Link Declarations	156	1
Editor Cells	841	0.25
Reference Constraints	21	2
Property Constraints	26	2
Behavior Methods	299	1
Type System Rules	148	1
Generation Rules	57	10
Statements	4919	1.2
Intentions	47	3
Text Generators	103	2
Total LOC		**8,640**

Figure 11. We count various language definition elements and then use a factor to translate them into lines of code. The reasons why many factors are so low (e.g. reference constraints or behavior methods) is that the implementation of these elements is made up of statements, which are counted separately. In case of editor cells, typically several of them are on the same line, hence the fraction. Finally, the MPS implementation language supports higher order functions, so some statements are rather long and stretch over more than one line: this explains the 1.2 in the factor for statements.

provides no automated way of ensuring this, we have not seen such interactions so far. The following steps can be taken to minimize the risk of unexpected interactions. Generated names should be qualified to make sure that no symbol name clashes occur in the generated C code. An extension should never consume "scarce resources": for example, it is a bad idea for a new `Statement` to require a particular return type of the containing function, or change that return type during transformation. Two such badly designed statements cannot be used together because they will likely require *different* return types. Note that unintended *syntactic* integration problems between independently developed extensions (known from traditional parser-based systems) can *never* happen in MPS. This was one of the reasons to use MPS for mbeddr.

Modularity should also support reuse in contexts not anticipated during the design of a language module. Just as in the case of language extension (discussed above), the to-be-reused languages have to be written in a suitable way so that the right parts can be reused separately. We have shown this with the state machines language. State machines can be used as top level concepts in modules (binding out events to C functions) and also inside components (binding out events to component methods). Parts of the transformation of a state machine have to be different in these two cases, and these differences were successfully isolated to make them exchangeable. Also, we reuse the C expression language inside the guard conditions in a state machine's transitions. We use constraints to prevent the use of those C expression that are not allowed inside transitions (for example, references to global variables). Finally, we have successfully used physical units in components and interfaces.

Summing up, these facilities allow different user groups to develop independent extensions, growing the mbeddr stack even closer towards their particular domain. The fact that all of them are possible with MPS clearly illustrates MPS' suitability for defining non-trivial language ecosystems.

Why MPS? A central pillar to our work is MPS. Our choice of MPS is due to its support for all aspects of language development (structure, syntax, type systems, IDE, transformations), its support for flexible syntax as a consequence of projectional editing and its support for advanced modularization and composition of languages. The ability to attach annotations to arbitrary program elements without a change to that element's definition is another strong advantage of MPS (we we use this for presence conditions and trace links, for example). No other freely available tool provides support for all those aspects, but some are supported by other tools. For example, Eclipse Xtext[5] and its accompanying tool

[5] http://eclipse.org/xtext

stack supports abstract and concrete syntax definition, IDE support and transformations, but it is weak regarding non-textual syntax and modularization and composition of languages. TU Delft's Spoofax[6] concise type system definition. Intentional Software[7] supports extremely flexible syntax and language composition (it is a projectional editor) but is not easily available.

Another important reason for our choice is the maturity and stability of MPS and the fact that it is backed by a major development tool vendor (JetBrains).

While the learning curve for MPS is significant (a developer who wants to become proficient in MPS language development has to invest at least a month), we found that is scales extremely well for larger and more sophisticated languages. This is in sharp contrast to some of the other tools the authors worked with, where implementing simple languages is quick and easy, and larger and more sophisticated languages are disproportionately more complex to build. This is illustrated by very reasonable effort necessary for implementing mbeddr.

Projectional Editing Projectional editing is often considered a drawback because the editors feel somewhat different and the programs are not stored as text, but as a tree (XML). We already highlighted that MPS does a good job regarding the editor experience, and we feel that the advantages of projectional editors regarding syntactic freedom far outweigh the drawback of requiring some initial familiarization. Our experience so far with about ten users (pilot users from industry, students) shows that after a short guided introduction (ca. 30 minutes) and an initial accomodation period (ca. 1-2 days), users can work productively with the projectional editor. Regarding storage, the situation is not any worse than with current modeling tools that store models in a non-textual format, and MPS does provide good support for diff and merge using the projected syntax.

4. Summary

Based on our experience with mbeddr as well as experiences with other language workbenches, MPS is currently clearly the most powerful and flexible one. It is also reasonably mature and stable. Considering the published MPS roadmap, we are confident that MPS will remain a leader in language workbenches for the forseeable future and we are looking forward to many interesting additions to mbeddr based on new features provided by MPS.

[6] http://spoofax.org

[7] http://intentsoft.com