mbeddr: an Extensible C-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 good support for writing efficient, low-level code, it is not adequate for defining higher-level abstractions relevant to embedded software. In this paper we present the 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 paper we show how language extension can address the challenges of embedded software development and report on our experience in building these extensions. We show that language workbenches deliver on the promise of significantly reducing the effort of language engineering and the construction of corresponding IDEs. mbeddr is built on top of the JetBrains MPS language workbench. Both MPS and mbeddr are open source software.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments - Programmer workbench

Keywords language extension, DSLs, development environments, embedded software, formal methods

1. Introduction

The amount of software embedded in devices is growing (see, for example, the German National Roadmap for Embedded Systems [9]). Embedded software development is challenging. In addition to functional requirements, strict operational requirements have to be fulfilled, including reliability (a device may not be accessible for maintenance after deployment), safety (a failure may endanger life or property), efficiency (the resources available to the system may be limited) or real-time constraints (a system may have to run on a strict schedule prescribed by the environment). Addressing these challenges requires any of the following: abstraction techniques should not lead to excessive runtime overhead; programs should be analyzable for faults before deployment; and various kinds of annotations, for example for describing and type checking physical units, must be integrated into the code. Process issues such as requirements traceability have to be addressed, and developers face a high degree of variability, since embedded systems are often developed in the context of product lines.

Current approaches for embedded software development can roughly be distinguished into programming and modeling. The *programming* approach mostly relies on C, sometimes C++ and Ada in rare cases. However, because of C's limited support for defining custom abstractions, this can lead to software that is hard to understand, maintain and extend. Furthermore, C's ability to work with very low-level abstractions such as pointers, makes C code very expensive to analyze statically. The alternative approach uses *modeling* tools with automatic code generation. The modeling tools provide predefined, higher-level abstractions such as state machines or data flow component diagrams. Example tools include ASCET-SD[1] or Simulink[2]. 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[3]). Increasingly, *domain specific* languages (DSLs) are used for embedded software [1, 17, 18]. Studies such

[1] http://www.etas.com/

[2] http://www.mathworks.com/products/simulink

[3] http://www.mathworks.com/products/sldesignverifier

as [7] and [?] show that domain-specific languages substantially increase productivity in embedded software development. However, most real-world systems cannot be described completely and adequately with a single modeling tool or DSL, and the integration effort between manually written C code and possibly several modeling tools and DSLs becomes significant.

A promising solution to this dilemma lies in a much tighter integration between low-level C code and higher-level abstractions specific to embedded software. We achieve this with an extensible C programming language. The advantages of C can be maintained: existing *legacy code* can be easily integrated, reused, and evolved, and the need for *efficient code* is immediately addressed by relying on C's low-level programming concepts. At the same time, domain-specific extensions such as state machines, components or data types with physical units can be made available as C extensions. This improves *productivity* via more concise programs, it helps improve *quality* in a constructive way by avoiding low-level implementation errors up-front, and leads to system implementations that are more amenable to *analysis*. By directly embedding the extensions into C, the mismatch and integration challenge between domain specific models and general purpose code can be removed. An industry-strength implementation of this approach must also include IDE support for C and the extensions: syntax highlighting, code completion, error checking, refactoring and debugging.

Developing such an extensible language and IDE is hard, but modern language engineering approaches promise to make it much simpler. The LW-ES research project, run by itemis AG, fortiss GmbH and Sick AG explores the benefits of language engineering in the context of embedded software development. The open source project is hosted at `http://mbeddr.com`. The code is available via this site.

Contribution In this paper we present mbeddr, an extensible C-based language and IDE for embedded software development. In particular, the paper makes the following contributions:

First, we present the design and implementation of mbeddr, which relies heavily on language engineering techniques. Extensions to C are *modular*, i.e. they can be developed without changing the C base language, and they are *incremental*, since they can be developed at any time and users can include extension modules into programs as the need arises. Extensions address syntax, type systems, semantics (by transformation to lower abstraction levels) as well as IDE support.

Second, the paper serves as a case study for the power and maturity of projectional language workbenches, and MPS in particular. We show how, with very limited effort, we were able to implement a significant set of languages that is ready to be used by embedded developers.

Third, we present a new approach to embedded software development located between programming and modeling. We illustrate a set of extensions to C that address important challenges in the embedded domain. Examples of such extensions include state machines, components and interfaces as well as the possibility of defining different restrictions of C in order to make programs conform to programming standards. We briefly illustrate how these extensions enable the use of advanced analyses such as model checking.

Even though the work presented here is centered on C and embedded software, the approach can be used with other domains and other base languages (we discuss this in Section 7). In this case, the first contribution would serve as a blueprint for identifying, motivating and designing language extensions. The second would serve as comparative reference for future (research) projects that use other language approaches or tools. The third would serve as a baseline for more specialized DSLs in specific subdomains of embedded software.

Outline In the next section we describe in more detail the challenges faced in embedded software development. In Section 3, we provide an overview over our solution approach and identify ways in which C must be extensible to allow the definition of adequate domain specific abstractions. Section 3.3 introduces a number of example extensions that address the challenges outlined in Section 2. We describe the implementation of these extensions, and with it, the design of the extensible C base language in Section 4. We discuss our experience in building mbeddr in Section 5. We wrap up the paper with related work (Section 6), a discussion (Section 7) and an outlook on future work in Section 8.

2. 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 3.3 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 [27] 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 [20]. 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.

3. 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). The following section briefly discusses language extension in general and explores which ways W_m of extending C are necessary to address the challenges C_n. Section 3.3 then shows examples that address each of the challenges and ways of extending C.

3.1 Language Extension

In [35] we classify strategies for language modularization and composition. 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,

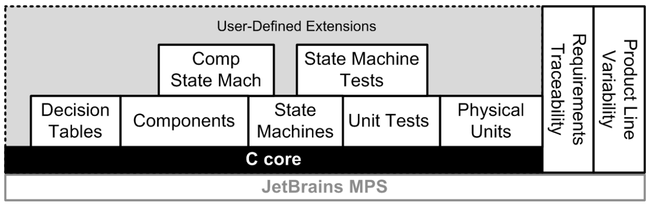


Figure 1. Based on MPS, mbeddr comes with an implementation of the C programming language. On top of C mbeddr defines a set of default extensions (white boxes) stacked on top of each other. Users can use them in their programs, but they don't have to. Support for requirements traceability and product 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.)

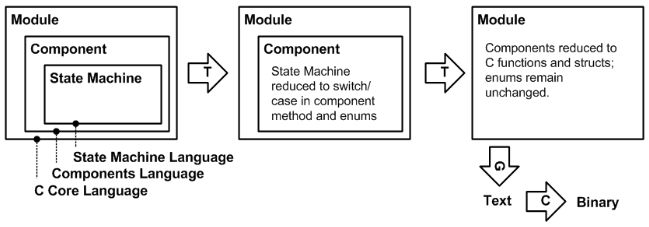

Module
Component
State Machine
T
Module
Component
State Machine reduced to switch/ case in component method and enums
T
Module
Components reduced to C functions and structs; enums remain unchanged.
State Machine Language
Components Language
C Core Language
G
Text
C
Binary

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.

as we will see in the examples provided in Section 3.3. In [35] we identify two strategies that support syntactic integration: 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 IDE that is aware of the language extensions. It is much more than a macro system or an open compiler (cf. Related Work in Section 6). Our implementation technology, the JetBrains MPS open source language workbench, 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, an approach also called assimilation [6]. 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.

3.2 Ways to extend C

In this section we discuss in which particular ways C needs to be extensible to support addressing the challenges discussed above. Section 3.3 shows examples for each of the ways.

W_1**: Top Level Constructs** Top level constructs (on the level of functions or `struct` declarations) are necessary. This enables the integration of test cases or new programming paradigms relevant in particular domains such as state machines, or interfaces and components.

W_2**: Statements** New statements, such as `assert` or `fail` statements in test cases, must be supported. If statements introduce new blocks, then variable visibility and shadowing must be handled correctly, just as in regular C. Statements may have to be restricted to a specific context; for example the `assert` or `fail` statements must *only* be used in test cases and not in any other statement list.

W_3: **Expressions** New kinds of expressions must be supported. An example is a decision table expression that represents a two-level decision tree as a two dimensional table (Fig. 5).

W_4: **Types and Literals** New types, e.g. for matrices, complex numbers or quantities with physical units must be supported. This also requires defining new operators and overriding the typing rules for existing ones. New literals may also be required: for example, physical units could be attached to number literals (as in `10kg`).

W_5: **Transformation** Alternative transformations for existing language concepts must be possible. For example, in a module marked as `safe`, `x + y` may have to be translated to `addWithBoundsCheck(x, y)`, a call to an `inline` function that performs bounds-checking besides the addition.

W_6: **Meta Data Decoration** It should be possible to add meta data such as trace links to requirements or product line variability constraints to arbitrary program nodes, without changing the concept of the node.

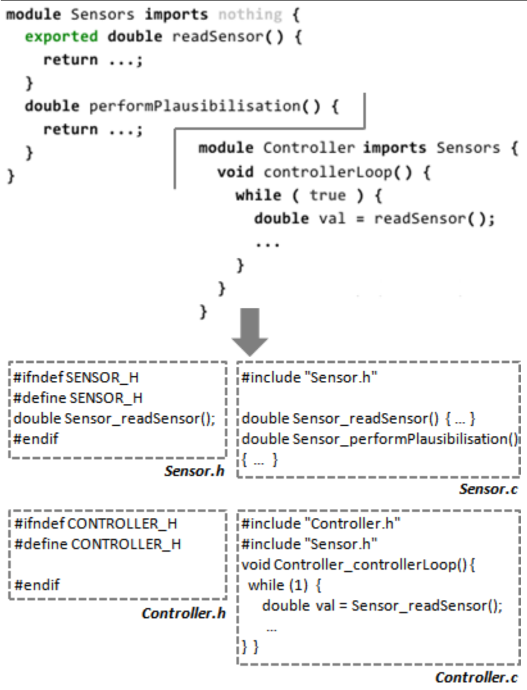
W_7: **Restriction** It should be possible to define contexts that restrict the use of certain language concepts. Like any other extension, such contexts must be definable *after* the original language has been implemented, without invasive change. For example, the use of pointer arithmetic should be prohibited in modules marked as *safe* or the use of real numbers should be prohibited in state machines that are intended to be model checked (model checkers do not support real numbers).

Challenge	Example Extensions
C_1 (Low-Overhead Abstraction)	State machines (W_1, W_2) Components (W_1) Decision Tables (W_3)
C_2 (Safer C)	Cleaned up C (W_7) Safe Modules (W_5, W_7)
C_3 (Annotations)	Physical Units (W_4)
C_4 (Static Checks, Verification)	Unit Tests (W_1, W_2) State Machines (W_1, W_2) Safe Modules (W_2, W_5, W_7)
C_5 (Process Support)	Requirements Traceability (W_6) Product Line Variability (W_6)

Figure 3. Embedded software development challenges, example extensions in this section, and the ways of extending C each example makes use of.

3.3 Extensions addressing the Challenges

In this section we present example extensions that illustrate how we address the challenges discussed in Section 2. We show at least one example for each challenge. How such extensions are built will be discussed in the next section, Section 4. 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.



```
module Sensors imports nothing {
  exported double readSensor() {
    return ...;
  }
  double performPlausibilisation() {
    return ...;
  }
}
```

```
module Controller imports Sensors {
  void controllerLoop() {
    while ( true ) {
      double val = readSensor();
      ...
    }
  }
}
```

Sensor.h
```
#ifndef SENSOR_H
#define SENSOR_H
double Sensor_readSensor();
#endif
```

Sensor.c
```
#include "Sensor.h"

double Sensor_readSensor() { ... }
double Sensor_performPlausibilisation()
{ ... }
```

Controller.h
```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#endif
```

Controller.c
```
#include "Controller.h"
#include "Sensor.h"
void Controller_controllerLoop() {
 while (1) {
   double val = Sensor_readSensor();
   ...
} }
```

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.

A cleaned up C (addresses C_2, uses W_7) To make C extensible, we first had to implement C in MPS. This entails the definition of the language structure, syntax and type system[4]. 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 `struct`s, functions or global variables). These module contents can be `exported`. Modules can *import* other modules, in which

[4] 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

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.

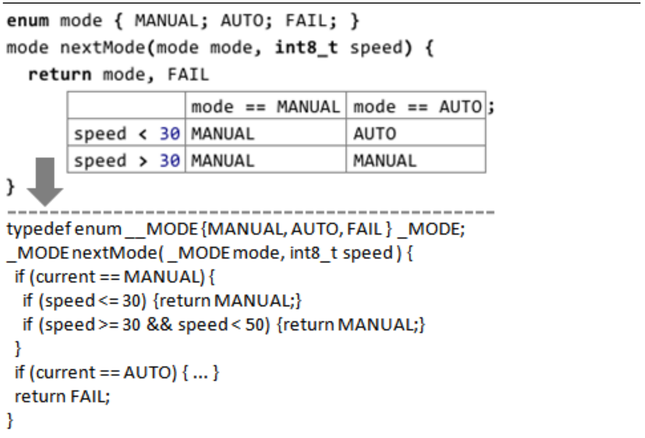
mbeddr C does not support the *preprocessor*. Empirical studies such as [16] show that it is often used to emulate missing features of C in ad-hoc way, leading to problems regarding maintenance and analyzability. Instead, mbeddr C provides first class support for the most important use cases of the preprocessor. 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, uses W_3) 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`).

Unit Tests (addresses C_4, uses W_1, W_2) 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, uses W_1) are new top level constructs that support modularization, encapsulation and 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



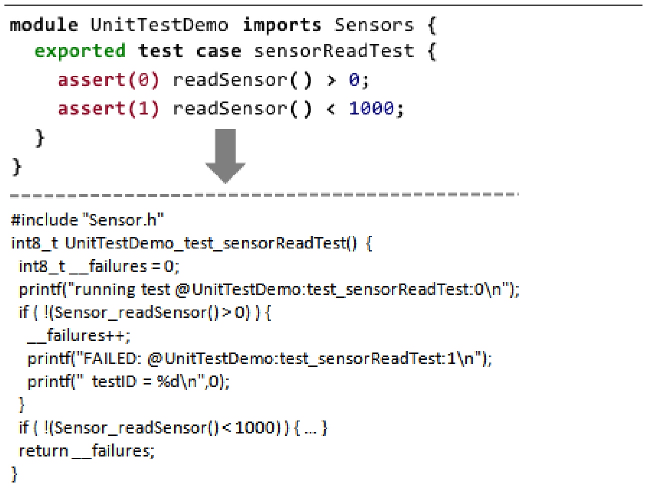
```
enum mode { MANUAL; AUTO; FAIL; }
mode nextMode(mode mode, int8_t speed) {
  return mode, FAIL
```

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

```
}
```

```
typedef enum __MODE {MANUAL, AUTO, FAIL } _MODE;
_MODE nextMode( _MODE mode, int8_t speed ) {
 if (current == MANUAL) {
  if (speed <= 30) {return MANUAL;}
  if (speed >= 30 && speed < 50) {return MANUAL;}
 }
 if (current == AUTO) { ... }
 return FAIL;
}
```

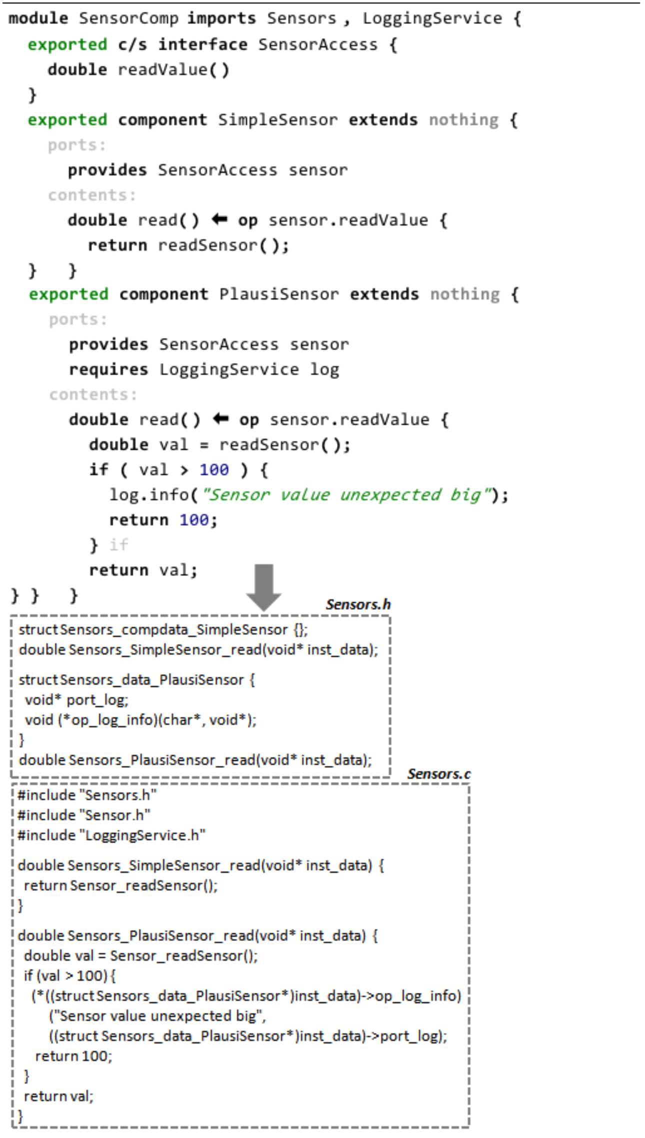
Figure 5. A decision table evaluates to the value in the cell for which the row and column headers are `true`, a default value otherwise (`FAIL` in the example). By default, a decision table is translated to nested `ifs` in a separate function. The figure shows the translation for the common case where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



```
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.



```
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.

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, uses W_1, W_2) 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 representation of the state machine in the input language of the NuSMV model checker[5], 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 in [12]. We discuss the integration of formal verification into mbeddr in more detail in [?].

```
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;
}
```

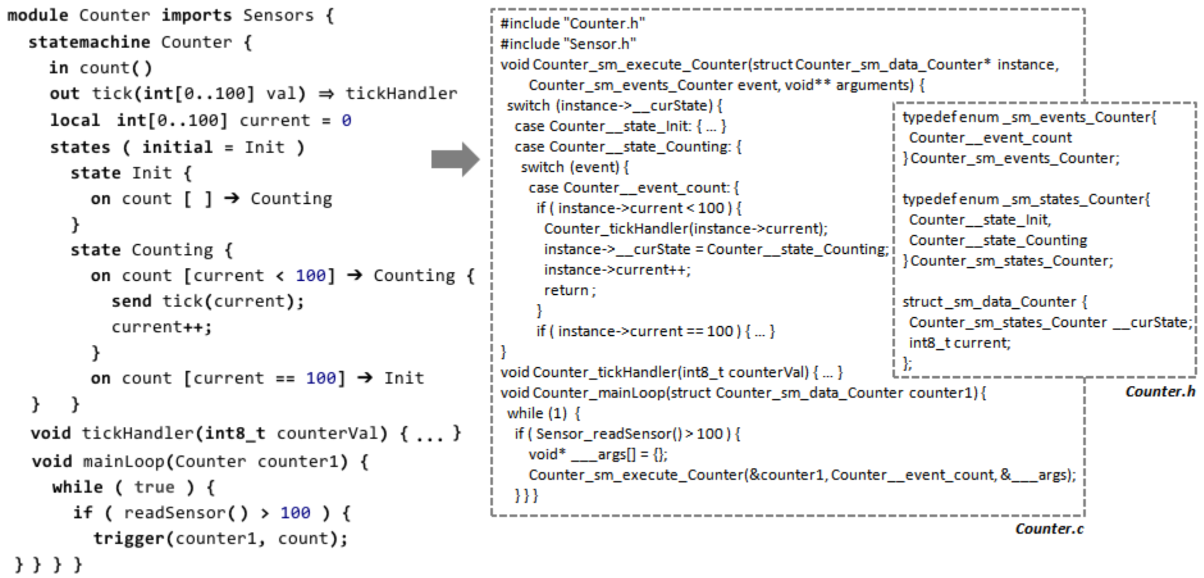
Figure 9. The *units* extension ships with the SI base units. Users can define derived units (such as the `mps` in the example) as well as convertible units that require a numeric conversion for mapping back to SI units. Type checks ensure that the values associated with unit literals use the correct 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, uses W_4) 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, uses W_6) 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 and in Fig. 23).

Presence Conditions (addresses C_5 and W_6) A presence condition determines whether the program element

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



```
module Counter imports Sensors {
  statemachine Counter {
    in count()
    out tick(int[0..100] val) ⇒ tickHandler
    local  int[0..100] current = 0
    states ( initial = Init )
      state Init {
        on count [ ] ➔ Counting
      }
      state Counting {
        on count [current < 100] ➔ Counting {
          send tick(current);
          current++;
        }
        on count [current == 100] ➔ Init
 }   }
  void tickHandler(int8_t counterVal) { ... }
  void mainLoop(Counter counter1) {
    while ( true ) {
      if ( readSensor() > 100 ) {
        trigger(counter1, count);
 } } } }
```

```
#include "Counter.h"
#include "Sensor.h"
void Counter_sm_execute_Counter(struct Counter_sm_data_Counter* instance,
     Counter_sm_events_Counter event, void** arguments) {
 switch (instance->__curState) {
  case Counter__state_Init: { ... }
  case Counter__state_Counting: {
   switch (event) {
    case Counter__event_count: {
     if ( instance->current < 100 ) {
      Counter_tickHandler(instance->current);
      instance->__curState = Counter__state_Counting;
      instance->current++;
      return ;
     }
     if ( instance->current == 100 ) { ... }
}
void Counter_tickHandler(int8_t counterVal) { ... }
void Counter_mainLoop(struct Counter_sm_data_Counter counter1) {
 while (1)  {
  if ( Sensor_readSensor() > 100 ) {
    void* ___args[] = {};
    Counter_sm_execute_Counter(&counter1, Counter__event_count, &___args);
  }}}
```

Counter.c

```
typedef enum _sm_events_Counter{
 Counter__event_count
}Counter_sm_events_Counter;

typedef enum _sm_states_Counter{
 Counter__state_Init,
 Counter__state_Counting
}Counter_sm_states_Counter;

struct _sm_data_Counter {
 Counter_sm_states_Counter __curState;
 int8_t current;
};
```

Counter.h

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.

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[6]. 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, uses W_5, W_7) 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. 14).

[6] 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.

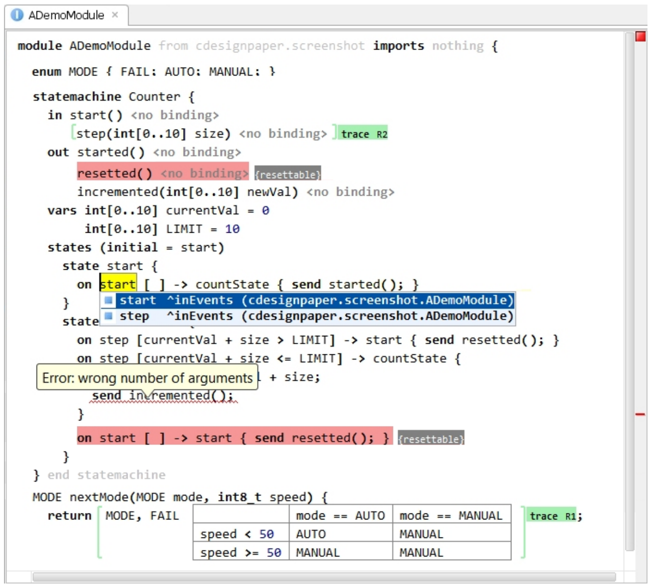
3.4 Addressing the Tool Integration Challenge

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. While we are still working on a debugger (see Section 8), all the other IDE support is available. 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.

4. Design and Implementation

This section discusses the implementation of mbeddr language extensions. For obvious reasons, this section cannot be a comprehensive tutorial of MPS. We refer to the MPS documentation[7]. However, the section provides a good overview of what it takes to build these extensions. We start by explaining in some detail how MPS works (Section 4.1). We then briefly discuss the structure of the C core language (Section 4.2). The major part of this chapter discusses each of the ways W_m of extending C (Section 4.3 through Section 4.9) based on the extensions discussed in the previous section.

[7] http://www.jetbrains.com/mps/documentation/index.html



ADemoModule

```
module ADemoModule from cdesignpaper.screenshot imports nothing {

  enum MODE { FAIL: AUTO: MANUAL: }

  statemachine Counter {
    in start() <no binding>
       [step(int[0..10] size) <no binding> ] trace R2
    out started() <no binding>
        resetted() <no binding> {resettable}
        incremented(int[0..10] newVal) <no binding>
    vars int[0..10] currentVal = 0
         int[0..10] LIMIT = 10
    states (initial = start)
      state start {
        on start [ ] -> countState { send started(); }
           start  ^inEvents (cdesignpaper.screenshot.ADemoModule)
      }    step   ^inEvents (cdesignpaper.screenshot.ADemoModule)
      state
        on step [currentVal + size > LIMIT] -> start { send resetted(); }
        on step [currentVal + size <= LIMIT] -> countState {
   Error: wrong number of arguments + size;
          send incremented();
        }
        on start [ ] -> start { send resetted(); } {resettable}
      }
  } end statemachine

  MODE nextMode(MODE mode, int8_t speed) {
    return [ MODE, FAIL  |             | mode == AUTO | mode == MANUAL | ] trace R1;
                         | speed < 50  | AUTO         | MANUAL         |
                         | speed >= 50 | MANUAL       | MANUAL         |
```

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

4.1 MPS Basics

MPS is a language workbench, a comprehensive tool for defining, extending, composing and using sets of integrated languages. As we will see, it supports the definition of various language aspects with highly expressive DSLs. MPS' most important characteristic is that it is a projectional editor. In parser-based approaches, users use text editors to enter character sequences that represent programs. A parser then checks the text for syntactic correctness and constructs an abstract syntax tree from the character sequence. The AST contains all the semantic information expressed by the program.

Projectional editors such as MPS do not use parsers. In projectional editors, the process happens the other way round: as a user edits the program, the AST is modified directly. A projection engine then creates a representation of the AST that reflects the changes. The user interacts with this representation. This approach is well-known from graphical editors: when editing a UML diagram, users don't draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates and instance of `uml.Class` as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case drawing a rectangle for the class. This approach can be generalized to work with any notation, including textual.

In projectional editors, every program element is stored as a node with a unique ID (UID) in the AST. References between program elements are based on actual pointers (references to UIDs). The AST is actually an ASG, an abstract syntax graph, from the start because cross-references are first-class rather than being resolved after parsing. The program is stored using a generic tree persistence mechanism, often XML.

The projectional approach can deal with arbitrary syntactic forms such as text, tables, symbols and graphics (graphics expected to be supported by MPS in 2013). Since no grammar is used, grammar classes are not relevant, and no syntactic ambiguities can result from the combination of independently developed languages. If two concepts (possibly defined by different language extensions) with the same syntax are valid in the same location, the user is forced to decide which one to instantiate as she enters the program. In principle, projectional editing is simpler than parsing, since there is no need to "extract" the program structure from a flat textual source. However, the challenge in projectional editing is making the editing experience convenient and productive. Traditionally, projectional editors have had a bad reputation because the user experience in editing programs was unacceptable. MPS has solved this problem, the editing experience is comparable to traditional text editors. Among others, MPS uses the following strategies to achieve this: aliases are used to instantiate language concepts from the code completion menu (e.g. you can just type `for` to instantiate a `ForStatement`); side transformations support entering trees linearly (e.g. you can just type + and 3 on the right side of a 2 to get 2+3); and the code completion menu shows targets of references directly instead requiring users to first instantiate the reference concept (e.g. when pressing Ctrl-Space after the + in 2+3, you will directly see all visible variables and arguments in the code completion menu).

4.2 The mbeddr Core Languages

C can be partitioned into expressions, statements, functions, etc. We have factored these parts into separate language modules to make each of them reusable without pulling in all of C. The `expressions` language is the most fundamental language. It depends on no other language and defines the primitive types, the corresponding literals and the basic operators. Support for pointers and user defined data types (`enum, struct, union`) is factored into the `pointers` and `udt` languages, respectively. `statements` contains the procedural part of C, and the `modules` language covers modularization. Fig. 11 shows an overview over some of the languages and constructs.

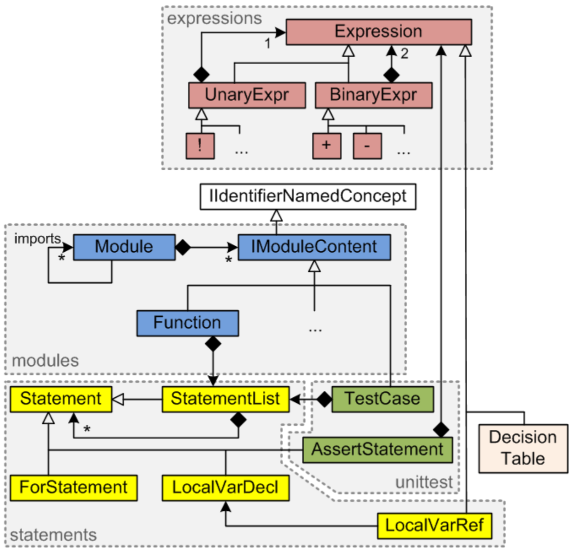


Figure 11. Anatomy of the mbeddr language stack: the diagram shows some of the language concepts, their relationships and the languages that contain them.

4.3 Addressing W_1 (Top-Level Constructs): Test Cases

In this section we illustrate the implementation of the `test case` construct as well as of the `assert` and `fail` statements available inside test cases.

Structure `Modules` own a collection of `IModuleContents`, an interface that defines the properties of everything that can reside directly in a module. All top-level constructs such as `Functions` implement `IModuleContent`. `IModuleContent` extends MPS' `IIdentifierNamedConcept` interface, which provides a `name` property. `IModuleContent` also defines a Boolean property `exported` that determines whether the respective module content is visible to modules that import this module. This property is queried by the scoping rules that determine which elements can be referenced. Since the `IModuleContent` interface can also be implemented by concepts in other languages, new top level constructs such as the `TestCase` in the `unittest` language can implement this interface, as long as the respective language has a dependency on the `modules` language, which defines `IModuleContent`. Fig. 11 shows some of the relevant concepts and languages.

Constraints A test case contains a `StatementList`, so any C statement can be used in a test case. `StatementList` becomes available to the unit test language through its dependency on the `statements` language. `unittest` also defines new statements: `assert` and `fail`. They extend the abstract `Statement` concept defined in the `statements` language. This makes them valid in *any* statement list, for example in a function body. This is undesirable, since the transformation of `asserts` into C depends on them being used in a `TestCase`. To enforce this, a *can be child* constraint is defined (Fig. 12).

```
concepts constraints AssertStatement {
  can be child
    (context, scope, parentNode, link, childConcept)->boolean {
      parentNode.ancestor<TestCase>.isNotNull;
}   }
```

Figure 12. This constraint restricts an `AssertStatement` to be used only inside a `TestCase` by checking that at least one of its ancestors is a `TestCase`.

```
test case exTest {
  int x = add(2, 2);
  assert(0) x == 4;
}
```

```
void test_exTest {
  int x = add(2, 2);
  report
    test.FAIL(0)
    on !(x == 4);}
```

```
void test_exTest {
  int x = add(2, 2);
  if (!(x == 4)) {
    printf("fail:0");
} }
```

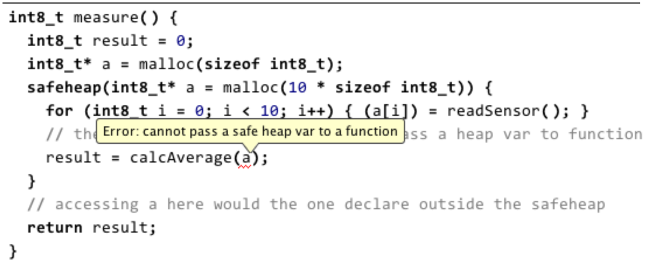
Figure 13. Two-stage transformation of `TestCases`. The `TestCase` is transformed into a C function using the logging framework to output error messages. The `report` statement is in turn transformed into a `printf` statement *if* we generate for the Windows/Mac environment. It would be transformed to something else if we generated for the actual target device (configured by the user in the build configuration).

Transformation The new language concepts in `unittest` are reduced to C concepts: the `TestCase` is transformed to a `void` function without arguments and the `assert` statement is transformed into a `report` statement defined in the logging language. The `report` statement, in turn, it is transformed into a platform-specific way of reporting an error (console, serial line or error memory). Fig. 13 shows an example of this two-step process.

4.4 Addressing W_2 (Statements): Safeheap Statement

We have seen the basics of integrating new statements in the previous section where `assert` and `fail` extended the `Statement` concept inherited from the C core languages. In this section we focus on statements that require handling local variable scopes and visibilities. We implement the `safeheap` statement mentioned earlier (see Fig. 14), which automatically frees dynamically allocated memory. The variables introduced by the `safeheap` statement must only be visible inside its body and they have to shadow variables of the same name declared in outer scopes (such as the `a` declared in the second line of the `measure` function in Fig. 14).

Structure The `safeheap` statement extends `Statement`. It contains a `StatementList` as its body, as well as a list of `SafeHeapVars`. These extend `LocalVarDecl`,



```
int8_t measure() {
  int8_t result = 0;
  int8_t* a = malloc(sizeof int8_t);
  safeheap(int8_t* a = malloc(10 * sizeof int8_t)) {
    for (int8_t i = 0; i < 10; i++) { (a[i]) = readSensor(); }
    // the Error: cannot pass a safe heap var to a function ass a heap var to function
    result = calcAverage(a);
  }
  // accessing a here would the one declare outside the safeheap
  return result;
}
```

Figure 14. A `safeheap` statement declares heap variables which can only be used inside the body of the statement. When the body is left, the memory is automatically freed. Notice also how we report an error in case the variable tries to escape.

so they fit with the existing mechanism for handling variable shadowing (explained below).

Behaviour `LocalVarRef`s are expressions that reference `LocalVarDecl`. A scope constraint, a mechanism provided by MPS, determines the set of visible variables for a given `LocalVarRef`. We implement this constraint by plugging into mbeddr's generic local variable scoping mechanism using the following approach. The constraint ascends the containment tree until it finds a node which implements `ILocalVarScopeProvider` and calls its `getLocalVarScope` method. A `LocalVarScope` has a reference to an outer scope, which is set by finding *its* `ILocalVarScopeProvider` ancestor, effectively building a hierarchy of `LocalVarScope`s. To get at the list of the visible variables, the `LocalVarRef` scope constraint calls the `getVisibleLocalVars` method on the innermost `LocalVarScope` object. This method returns a flat list of `LocalVarDecl`s, taking into account that variables owned by a `LocalVarScope` that is *lower* in the hierarchy shadow variables of the same name from a *higher* level in the hierarchy. So, to plug the `SafeHeapStatement` into this mechanism, it has to implement `ILocalVarScopeProvider` and implement the two methods shown in Fig. 15.

Type System To make the `safeheap` statement work correctly, we have to ensure that the variables declared and allocated in the *safeheap* statement do not escape from its scope. To prevent this, an error is reported if a reference to a `safeheap` variable is passed to a function. Fig. 16 shows the code.

4.5 Addressing W_3 (Expressions): Decision Tables

Expressions are different from statements in that they can be evaluated to a *value* as the program executes. During editing and compilation, the *type* of an expression is relevant for the static correctness of the program. So extending a language regarding expressions requires extending the type system rules as well.

```
public LocalVarScope getLocalVarScope(node<> ctx, int stmtIdx) {
  LocalVarScope scope =
      new LocalVarScope(getContainedLocalVariables());
  node<ILocalVarScopeProvider> outerScopeProvider =
      this.ancestor<ILocalVarScopeProvider>;
  if (outerScopeProvider != null)
    scope.setOuterScope(outerScopeProvider.
                  getLocalVarScope(this, this.index));
  return scope;
}
public sequence<node<LocalVariableDecl>> getContainedLocalVars() {
  this.vars;
}
```

Figure 15. A `safeheap` statement implements the two methods declared by the `ILocalVarScopeProvider` interface. `getContainedLocalVariables` returns the `LocalVarDecl`s that are declared between the parentheses (see Fig. 14). `getLocalVarScope` constructs a scope that contains these variables and then builds the hierarchy of outer scopes by relying on its ancestors that also implement `ILocalVarScopeProvider`. The index of the statement that contains the reference is passed in to make sure that only variables declared *before* the reference site can be referenced.

```
checking rule check_safeVarRef for concept = LocalVarRef as lvr {
    boolean isInSafeHeap =
      lvr.ancestor<SafeHeapStatement>.isNotNull;
    boolean isInFunctionCall =
      lvr.ancestor<FunctionCall>.isNotNull;
    boolean referencesSafeHeapVar =
      lvr.var.parent.isInstanceOf(SafeHeapStatement);
    if (isInSafeHeap && isInFunctionCall && referencesSafeHeapVar)
         error "cannot pass a safe heap var to a function" -> lvr;
}
```

Figure 16. This type system rule reports an error if a reference to a local variable declared and allocated by the `safeheap` statement is used in a function call.

Fig. 5 shows the decision table expression. It is evaluated to the expression in a cell *c* if the column header of *c* and the row header of *c* are `true`[8]. If none of the condition pairs is `true`, then the default value, `FAIL` in the example, is used as the resulting value. A decision table also specifies the type of the value it will evaluate to, and all the expression in content cells have to be compatible with that type. The type of the header cells has to be Boolean.

Structure The decision table extends the `Expression` concept defined in the `expressions` language. Decision tables contain a list of expressions for the column headers, one for the row headers and another one for the result values. It also contains a child of type `Type` to declare the type of the result expressions, as well as a default value expression. The concept defines an alias

[8] Strictly speaking, it is the *first* of the cells for which the headers are `true`. It is optionally possible to use static verification based on an SMT solver to ensure that only one of them will be `true` for any given set of input values

dectab to allow users to instantiate a decision table in the editor. Obviously, for non-textual notations such as the table, the alias will be different than the concrete syntax (in textual notations, the alias is typically made to be the same as the "leading keyword", e.g. assert).

Editor Defining a tabular editor is straight forward: the editor definition contains a table cell, which delegates to a Java class that implements ITableModel. This is similar to the approach to the approach used by Java Swing. It provides methods such as getValueAt(int row, int col) or deleteRow(int row), which have to be implemented for any specific table-based editor. To embed another node in a table cell (such as the expression in the decision table), the implementation of getValueAt simply returns this node.

Type System As mentioned above, MPS uses unification in the type system. Language concepts specify type equations that contain type literals (such as boolean) as well as type variables (such as typeof(dectab)). The unification engine then tries to assign values to the type variables so that all applicable type equations become true. New language concepts contribute additional type equations. Fig. 17 shows those for decision tables. New equations are solved along with those for existing concepts. For example, the typing rules for a ReturnStatement ensure that the type of the returned expression is the same or a subtype of the type of the surrounding function. If a ReturnStatement uses a decision table as the returned expression, the type calculated for the decision table must be compatible with the return type of the surrounding function.

```
  // the type of the whole decision table expression
  // is the type specified in the type field
typeof(dectab) :==: typeof(dectabc.type);
  // for each of the expressions in
  // the column headers, the type must be boolean
foreach expr in dectab.colHeaders {
  typeof(expr) :==: <boolean>;
}
  // ... same for the row headers
foreach expr in dectabc.rowHeaders {
  typeof(expr) :==: <boolean>;
}
  // the type of each of the result values must
  // be the same or a subtype of the table itself
foreach expr in dectab.resultValues {
  infer typeof(expr) :<=: typeof(dcectab);
}
  // ... same for the default
typeof(dc.def) :<=: typeof(dectab);
```

Figure 17. The type equations for the decision table (see the comments for details).

4.6 Addressing W_4 (Types and Literals): Physical Units

To illustrate adding new types and literals we use physical units. We had already shown example code earlier in Fig. 9.

Structure Derived and convertible UnitDeclarations are IModuleContents. Derived unit declarations specify a name (mps, kmh) and the corresponding SI base units (m, s) plus an exponent; a convertible unit declaration specifies a name and a conversion formula. The backbone of the extension is the UnitType which is a composite type that has another type (int, float) in its valueType slot, plus a unit (either an SI base unit or a reference to a UnitDeclaration). It is represented in programs as baseType/unit/. We also provide LiteralWithUnits, which are expressions that contain a valueLiteral and, like the UnitType, a unit (so we can write 100 kmh).

Scoping LiteralWithUnits and UnitTypes refer to a UnitDeclaration, which is a module content. According to the visibility rules, valid targets for the reference are the UnitDeclarations in the same module, and the *exported* ones in all imported modules. This rule applies to *any* reference to *any* module contents, and is implemented generically in mbeddr. Fig. 18 shows the code for the scope of the reference to the UnitDeclaration. We use an interface IVisibleNodeProvider, (implemented by Modules) to find all instances of a given type. The implementation of visibleContentsOfType simply searches through the contents of the current and imported modules and collects instances of the specified concept. The result is used as the scope for the reference.

```
link {unit} search scope:
    (model, scope, refNode, enclosingNode, operationContext)
                               ->sequence<node<UnitDeclaration>> {
      enclosingNode.ancestor<IVisibleNodeProvider>.
              visibleContentsOfType(concept/UnitDeclaration/); }
```

Figure 18. The visibleContentsOfType operation returns all instances of the concept argument in the current module, as well as all exported instances in modules imported by the current module.

Type System We have seen how MPS uses equations and unification to specify type system rules. However, there is special support for binary operators that makes overloading for new types easy: overloaded operations containers essentially specify 3-tuples of *(leftArgType, rightArgType, resultType)* plus applicability conditions to match type patterns and decide on the resulting type. Typing rules for new (combinations of) types can be added by specifying additional 3-tuples. Fig. 19 shows the overloaded rules for C's MultiExpression (the language concept the implements the multiplication operator *) when applied to two UnitTypes: the result type will be a UnitType as well, where the exponents of the SI units are added.

While any two units can legally be used with * and / (as long as we compute the resulting unit exponents

```
operation concepts: MultiExpression
  left operand type: new node<UnitType>()
  right operand type: new node<UnitType>()
is applicable:
  (operation, leftOpType, rightOpType)->boolean {
    node<> resultingValueType = operation type(operation,
                  leftOpType.valueType , rightOpType.valueType );
    resultingValueType != null; }
operation type:
  (operation, leftOpType, rightOpType)->node<> {
     node<> resultingValueType = operation type(operation,
               leftOpType.valueType,  rightOpType.valueType );
     UnitType.create(resultingValueType,
                      leftOpType.unit.toSIBase().add(
                         rightOpType.unit.toSIBase(),
                         1 ) );
  }
```

Figure 19. This code overloads the `MultiExpression` to work for `UnitType`s. In the `is applicable` section we check whether there is a typing rule for the two value types (e.g. `int * float`). This is achieved by trying to compute the resulting value type. If none is found, the types cannot be multiplied. In the computation of the `operation type` we create a new `UnitType` that uses the `resultingValueType` as the value type and then computes the resulting unit by adding up the exponents of component SI units of the two operand types.

correctly), this is not true for `+` and `-`. There, the two operand types must be the same (in terms of their representation in SI base units). We express this by using the following expression in the `is applicable` section: `leftOpType.unit.isSameAs(rightOpType.unit)`.

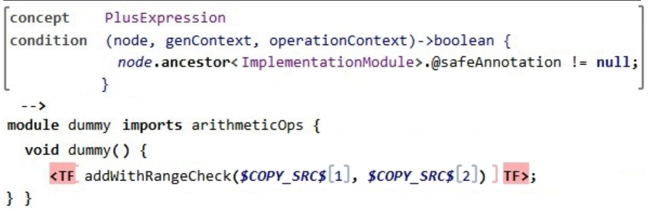
The typing rule for the `LocalVariableDeclaration` requires that the type of the `init` expression must be the same or a subtype of the `type` of the variable. To make this work correctly, we have to define a type hierarchy for `UnitTypes`. We achieve this by defining the supertypes for each `UnitType`: the supertypes are those `UnitTypes` whose unit is the same, and whose `valueType` is a supertype of the current `UnitType`'s value type. Fig. 20 shows the rule.

```
subtyping rule supertypeOf_UnitType
              for concept = UnitType as ut {
  nlist<> res = new nlist<>;
  foreach st in immediateSupertypes(ut.valueType) {
    res.add(UnitType.create(st, ut.unit.copy));
  }
  return res;
}
```

Figure 20. This typing rule computes the direct supertypes of a `UnitType`. It iterates over all immediate supertypes of the current `UnitType`'s value type, wrapped into a `UnitType` with the same unit as the original one.

4.7 Addressing W_5 (Alternative Transformations): Range Checking

The `safemodules` language defines an *annotation* to mark `Modules` as safe (we will discuss annotations in the next subsection). If a module is safe, the binary operators such as `+` or `*` are replaced with calls to functions that, in addition to performing the addition or multiplication, perform a range check.



```
concept    PlusExpression
condition  (node, genContext, operationContext)->boolean {
             node.ancestor<ImplementationModule>.@safeAnnotation != null;
           }
 -->
module dummy imports arithmeticOps {
  void dummy() {
    <TF addWithRangeCheck($COPY_SRC$[1], $COPY_SRC$[2]) TF>;
} }
```

Figure 21. This *reduction rule* transforms instances of `PlusExpression` into a call to a library function `addWithRangeChecks`, passing in the left and right argument of the `+` using the two `COPY_SRC` macros. The `condition` ensures that the transformation is only executed if the containing `Module` has a `safeAnnotation` attached to it. A transformation priority defined in the properties of the transformation makes sure it runs before the C-to-text transformation.

Transformation The transformation that replaces the binary operators with function calls is triggered by the presence of this annotation on the `Module` which contains the operator. Fig. 21 shows the code. The `@safeAnnotation != null` checks for the presence of the annotation. MPS uses priorities to specify relative orderings of transformations, and MPS then calculates a global transformation order for any given model. We use a priority to express that this transformation runs *before* the final transformation that maps the C tree to C text for compilation.

4.8 Addressing W_6 (Meta Data): Requirements Traces

Annotations are concepts whose instances can be added as children to a node N without this being specified in the definition of N's concept. While structurally the annotations are children of the annotated node, the editor is defined the other way round: the annotation editor delegates to the editor of the annotated element. This allows the annotation editor to add additional syntax *around* the annotated element. Optionally, it is possible to explicitly restrict the concepts to which a particular annotation can be attached. We use annotations in several places: the `safe` annotation discussed in the previous section, the requirements traces and the product line variability presence conditions.

Structure We illustrate the annotation mechanism based on the requirements traces. Fig. 22 shows the structure. Notice how it extends the MPS-predefined concept `NodeAttribute` (it sould be named `Node-Annotation`). It also specifies a `role`, which is the name of the property that is used to store `TraceAnnotations` under the annotated node.

```
concept TraceAnnotation extends NodeAttribute implements <none>
  children:
    TraceKind        tracekind   1
    TraceTargetRef   refs        0..n
  concept properties:
    role = trace
  concept links:
    attributed = BaseConcept
```

Figure 22. Annotations have to extend the MPS-predefined concept `NodeAttribute`. They can have an arbitrary child structure (`tracekind`, `refs`), but they have to specify the `role` (the name of the property that holds the annotated child under its parent) as well as the `attributed` concept: the annotations can only be attached to instances of this concept (or subconcepts).

Editor As mentioned above, in the editor, annotations look as if they *surrounded* their parent node (although they are in fact children). Fig. 23 shows the definition of the editor of the requirements trace annotation (and an example is shown in Fig. 10): it puts the trace to the right of the annotated node. Since MPS is a projectional editor, there is base-language grammar that needs to be made aware of the additional syntax in the program. This is key to enabling arbitrary annotations on arbitrary program nodes.

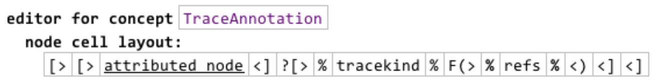
Annotations are typically attached to a program node via an intention. Intentions are an MPS editor mechanism: a user selects the target element, presses `Alt-Enter` and selects `Add Trace` from the popup menu. Fig. 24 shows the code for the intention that attaches a requirements trace.

4.9 Addressing W_7 (Restriction): Preventing Use of Reals Numbers

We have already seen in Section 4.3 how constraints can prevent the use of specific concepts in certain contexts. We use the same approach for preventing the use of real number types inside model-checkable state machines: a `can be ancestor` constraint in the state machine prevents instances of `float` in the state machine if the `verifiable` flag is set.

5. Experiences

This paper is about the design and rationale of an extensible C language for embedded development, based on language engineering techniques. In Section 5.1 we provide a brief overview over our experiences in implementing mbeddr, including the size of the project and the efforts spent. Section 5.2 discusses to what degree this approach leads to improvements in embedded software development.



```
editor for concept TraceAnnotation
  node cell layout:
    [> [> attributed node <] ?[> % tracekind % F(> % refs % <) <] <]
```

Figure 23. The editor definition for the `ReqTrace` annotation (an example trace annotation is shown in Fig. 10). It consists of a vertical list `[/ .. /]` with two lines. The first line contains the reference to the requirement. The second line uses the `attributed node` construct to embed to the editor of the program node to which this annotation is attached. So the annotation is always rendered right on top of whatever syntax the original node uses.

```
intention addTrace for BaseConcept {
  description(node)->string {
    "Add Trace"; }
  isApplicable(node)->boolean {
    node.@trace == null; }
  execute(editorContext, node)->void {
    node.@trace = new node<TraceAnnotation>(); }
}
```

Figure 24. An intention definition consists of three parts. The `description` returns the string that is shown in the intentions popup menu. The `isApplicable` section determines under which conditions the intention is avavailable in the menu — in our case, we can only add a trace if there is no trace yet on the target node. Finally, the `execute` section performs the action associated with the intention. In our case we simply put an instance of `TraceAnnotation` into the `@trace` property of the target node.

5.1 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. 25 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.

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

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 25. 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.

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[9], 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.

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

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 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.

Who can create Extensions? mbeddr is built to be extended. The question is by whom. This question can be addressed in two ways: who is *able* to extend it from a skills perspective, and who *should* extend it?

Let us address the *skills* question first. We find that it takes about a month for a developer with solid object-oriented programming experience to become proficient with MPS and the structures of the mbeddr core languages. This may be reduced by better documentation, but a steep learning curve will remain. Also, *designing* good languages, independent of their implementation, is a skill that requires practice and experience. So, from this perspective we assume that in any given organization there should be a select group of language developers who build the extensions for the end users. Notice that such an organizational structure is common today for frameworks and other reusable artifacts.

There is also the question of who *should* create extensions. One could argue that, as language development becomes simpler, an uncontrolled growth in languages could occur, ultimately resulting in chaos. This concern should be addressed with governance structures that guide the development of languages. The bigger the organization is, the more important such governance becomes. The modular nature of the mbeddr language extensions makes this problem much easier to tackle. In an large organization we assume that a few language extensions will be strategic: aligned with the needs of the whole organization, well-designed, well tested and documented, implemented by a central group, and used by many developers. In addition, small teams my decide to develop their own, smaller extensions. Their focus is much more local, and the development requires much less coordination. These could be developed by the smaller units themselves.

5.2 Improvements in Embedded Development

Productivity and Quality At this point we have not yet conducted large-scale industry projects with the mbeddr stack. We are currently in the process of setting up two real-world projects. However, two preliminary end-user experiments have been performed. The mbeddr development team itself has created a non-trivial case study based on the OSEK operating system and Lego Mindstorms. Second, a group of students from the University of Augsburg has developed a set of language extensions for controlling a quadcopter. Both cases resulted in much less code, a clearer implementation and fewer bugs compared to what we expected from traditional embedded software development. Both projects also developed extensions of the existing stack: the OSEK/Mindstorms project extended the build language to integrate with the NXT OSEK build system and the quadcopter project has developed languages for controlling and planning the routing for the quadcopter.

The fact that formal verification is directly integrated into mbeddr, and the fact the requirements traceability and product line variability are directly supported promises to improve the overall quality of systems built with mbeddr.

Size and Practicability We have run scalability tests to ensure that the environment scales to at least the equivalent of 100.000 lines of C code. A significant share of embedded software is below this limit and can confidently be addressed with mbeddr. We do not have any data which indicates significant performance degradation for larger systems, and we believe that by structuring systems into separate partitions that are transformed, compiled and linked separately, larger systems are feasible as well. However, to be sure, this requires further scalability testing.

Suitability of the Currently Available Extensions Based on years of experience in embedded software development and dozens of conversations with practitioners in the field we are confident that the extensions we chose provide useful benefits in real-world embedded software development projects. In particular, state machines, interfaces and components as well as traceability and product line support are relevant for almost every developer we talked to and available in several established modeling tools.

However, while we are confident that the default extensions are useful in practice, they mainly serve as a proof-of-concept for the idea of incremental, modular language extension, where end-user organizations build their own custom extensions that fit their domain.

6. Related Work

mbeddr touches several areas of research, so we we have structured the this section accordingly, one paragraph for each area: DSLs in embedded development, specific extensions of C, language and IDE extension and static analysis and formal verification.

DSLs in Embedded Development In addition to the general-purpose embedded software modeling tools mentioned before (Simulink and ASCET), much more specific languages have been developed. Examples include Feldspar [1], a DSL embedded in Haskell for digital signal processing; Hume [18], a DSL for real-time embedded systems as well as the approach described in [17], which use DSLs for addressing quality of service concerns in middleware for distributed real-time systems. Our approach is different because our DSLs are directly integrated into C, whereas the examples mentioned in this paragraph are standalone DSLs that generate C code. As part of our future work we will investigate if and how some of these languages could benefit from a tighter integration with C based on mbeddr.

Specific Extensions of C Extending C to adapt it to a particular problem domain is not new. For example, Palopoli et al. present an extension of C for real time applications [28], Boussinot proposes an extension for reactive systems [3] and Yosi Ben-Asher et al. present an extension for shared memory parallel systems [2]. These are all *specific* extensions of C, typically created by invasively changing the C grammar. These extensions do not include IDE support, and the approach does not provide a framework for modular, incremental extension. However, these are all good examples of extensions that could be implemented as language extensions in mbeddr, if the need arises.

In contrast to these specific extensions of C, the Xoc extensible C compiler described by Cox [8] support arbitrary extensions. It uses a parser-based approach and uses source-to-source translation to transform modular C extensions into regular C code. In contrast to mbeddr, Cox' approach is limited by the fact that is uses a traditional parser based approach and that it does not address IDE extension.

There are also safer dialects of C, basically restricted sub-languages. Examples include Cyclone [21] and the Misra C standard [27]. We are actively working on implementing checks and restrictions to implement the Misra C standard as a language extension using restriction (W_7).

Language and IDE Extension mbeddr itself is not a language engineering tool — we rely on the MPS language workbench. Therefore the discussion of language engineering approaches and tools as part of related work will look at how these tools and approaches address language engineering, and how this differs from the MPS-based approach used in mbeddr.

Language extension is not a new idea. The Lisp community has always considered language extension essential to using Lisp effectively. Guy Steele's OOPSLA 1998 keynote *Growing a Language* (and a related journal article [22]) is maybe the most well-known expression of the idea, and Thrift's extension of Lisp with constructs for logic programming [34] is a concrete example. Obviously, Lisp extension could not have been used as a basis for mbeddr, since it is based on C.

The landmark work of Hudak [19] introduces embedded DSLs as language extensions of Haskell. While Haskell provides advanced concepts that enable such extensions, the new DSLs are essentially just libraries built with the host language and are not first class language entities: they do not define their own syntax, compiler errors are expressed in terms of the host language, no custom semantic analyses are supported and no specific IDE-support is provided. Essentially all internal DSLs expressed with dynamic languages such as Ruby or Groovy, but also those embedded in static languages such as Scala suffer from these limitations.

Several works avoid these limitations by making language definition and extension first class. Early examples include the Synthesizer Generator [31] as well as the Meta Environment [24]. Both generate editors and other IDE aspects from a language definition. The topic is still actively researched. For example, Bravenboer et al. [5] and Dinkelacker [11] provide custom concrete syntax, Bracha [4] provides pluggable type systems and Erweg et al. [15] discuss modular IDE extensions. Eisenberg and Kiczales propose explicit programming [13] which supports semantic extension as well as editing extensions (concrete syntax) for a given base language.

Our approach is similar in that we provide extensions of syntax, type systems, semantics and IDE support for a base language. mbeddr is different in that it extends C, in that we use a projectional editor and in that we address IDE extension including advanced features such as type systems, refactorings and the debugger. The use of a projectional editor is especially significant, since this enables the use of non-textual notations and annotation of cross-cutting meta data.

A particularly interesting comparison can be made with the Helvetia system by Renggli et al. [30]. 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 some of the embedded DSLs discussed above, Helvetia can work with custom grammars for the DSLs). 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 some aspects of the Smalltalk system that are not present for other languages, and in particular, not in 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.

In the same paper, Renggli and his colleagues introduce three different flavors of language extension. A *pidgin* creatively bends the existing syntax of the host language to to extend its semantics. A *creole* introduces completely new syntax and custom transformations back to the host language. An *argot* reinterprets the semantics of valid host language code. mbeddr does not use any pidgins, because C's syntax is not very flexible, and because we have the language workbench at our disposal, so it is easier to implement creoles. W_1 - W_4 are creoles. In contrast, W_5 is an argot. It provides different semantics for existing constructs. W_6 is yet different. New syntax is introduced, but it can be attached to any language concept. The semantics is only relevant to additional tools, not to the core C program — no translation back to C takes place. W_7 *removes* concepts in new contexts and hence also does not fit with the categorization.

Cedalion [10] is a host language for defining internal DSLs. It uses a projectional editor and semantics based on logic programming. 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 base languages with the same tool, optional tight integration with general purpose host languages. We could not have used Cedalion as the platform for mbeddr tough, since we implemented our own base language (C), and the logic-based semantics would not have been a good fit.

Our work relates to macro systems such as Open Java [33]in that mbeddr customizes the translation of language extensions. However, mbeddr uses non-local transformations as well; those are not easily expressible with macros. Also, traditionally, macros have not addressed IDE extension.

Finally, open compilers such as Jastadd [14] are related in that they support language extension and custom transformation. However, while open compilers can typically be extended with independent modules, the input language often requires invasive adaptation. Also, open compilers do not address IDE extension.

Static Analysis and Formal Verification Static analysis of C programs is an active research area (as exemplified by [23, 26, 29]), and several commercial tools are available, such as the Escher C Verifier[10] or Klocwork[11]. We believe that we can simplify some of the analyses provided by these tools by providing extensions to C which embody relevant semantics directly, avoiding the need to reverse engineer the semantics for static analysis. For example, by expressing state-based behavior directly using state machines instead of a low level C implementation, the state space relevant to a model checker can be reduced significantly, making model checking less costly.

Another class of tools (such as Frama-C[12]) requires users to annotate C code with "semantic hints" to reduce the state space and enable meaningful verification. We plan to integrate Frama-C into mbeddr, expecting the following benefits: first, we will provide a language extension for the hints, so users don't have to use comments to specify them. IDE support will be provided. Second, we will provide C extensions on a higher level of abstraction with semantics that can be used to generate the verification hints. This way, users don't have to deal with the hints explicitly.

7. Discussion

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[13] and its accompanying tool 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[14] concise type system definition. Intentional Software[15] supports extremely flexible syntax [32] 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

[10] http://www.eschertech.com/products/ecv.php

[11] http://www.klocwork.com/

[12] http://frama-c.com/

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

[14] http://spoofax.org

[15] http://intentsoft.com

by very reasonable effort necessary for implementing mbeddr (see Section 5.1).

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.

Other Base Languages The technology described in this paper can be applied to other base languages. JetBrains, for example, is extending Java for building web applications. The advantage of using a heterogeneous approach (see the Helvetia discussion in Related Work) is that the tools built for language engineering are independent of the extended languages. No new frameworks or tools have to be developed or learned. Of course the to-be extended language has to be implemented in the tool stack first. We have discussed the effort for doing this in the case of C in Section 5.

Other Application Domains mbeddr's domain is embedded systems. However, the same approach can be used in other domains as well. As mentioned in the previous paragraph, JetBrains are developing Java extensions for web application development. These extensions include support for object-relational mapping, web page templating, and portability of application logic between the client and server by translating the same code into Java and Javascript. In internal communications with the authors, JetBrains have reported significant improvements in productivity and significantly reduced time (days and weeks instead of months) for getting new developers up to speed in web application development. JetBrains use this approach to develop the Youtrack bug tracking software, among others.

8. Conclusion and Future Work

In this paper we presented the mbeddr system, a large scale use of language engineering technologies in general and language workbenches in particular. We show how domain-specific extensions of C language can be used to address important challenges in embedded software development. To illustrate these ways of extension we provide a set of concrete examples and their implementation in the mbeddr system. The feedback on mbeddr received from practitioners so far convinces us that language engineering approaches have great potential to dramatically improve the development of embedded software. The mbeddr project also serves as a strong validation of the power and maturity of projectional language workbenches, in particular, MPS. The effort for building the C language and IDE and especially the incremental effort of building extensions is significantly lower than we expected when we started the project.

To realize the full potential of the mbeddr approach, more research is required in the following two major directions:

Extension of the Approach We are almost finised with a debugger that can be extended together with language extensions. While there is existing research (such as [25, 36]), there are still open questions such as how to calculate custom watches and how to avoid generating debug-specific code into the resulting C. We will also work more on formal analyses, including mapping higher-level DSLs to state machines and reinterpreting the verification results in the context of the higher-level DSL, exploring the relationship between general program analysis and language extensions as well as using SAT solvers to verify the structural integrity of variant-aware programs. In addition, we will add support for graphical notations for state machines and data flow block diagrams once MPS' support for graphical editors becomes available in the MPS 3.0 version.

Real-World Feasibility Since mbeddr is intended to be used for real-world software development, a major part of our future work is the validation of the approach in real-world embedded development projects. We are currently building a set of extensions specific to our application partner Sick AG who use mbeddr to build systems in the sensors domain. We are also setting up a project to develop a smart metering device. We will measure the increase in productivity and maintainability in order to provide solid data about the full potential of this approach. This line of future work will also include an automatic importer for functions, `struct`s, constants, `enum`s and `typdef`s defined in existing header files to simplify working with legacy code. We are also considering an importer for C implementation code (as long as it does not contain preprocessor statements). This will not be fully automatic, since some of the changes to mbeddr C require user decisions.

Acknowledgements We thank Marcel Matzat and Bernhard Merkle for their work on mbeddr. Supported by the German BMBF, FKZ 01/S11014.

References

[1] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegard, A. Persson, M. Sheeran, J. Sven-

ningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE 2010*.

[2] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. ParC - An Extension of C for Shared Memory Parallel Processing. *Software: Practice and Experience*, 26(5), 1996.

[3] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software: Practice and Experience*, 21(4), 1991.

[4] G. Bracha. Pluggable Type Systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages.*, 2004.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects: DSL embedding and assimilation without restrictions. *SIGPLAN Not.*, 39, October 2004.

[6] M. Bravenboer and E. Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In *MoDELS 2007*, volume 5002 of *LNCS*. Springer, 2007.

[7] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*. ICI, 2011.

[8] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS 2008*.

[9] W. Damm, R. Achatz, K. Beetz, M. Broy, H. Dämbkes, K. Grimm, and P. Liggesmeyer. *Nationale Roadmap Embedded Systems*. Springer, Mar. 2010.

[10] David H. Lorenz, Boaz Rosenan. Cedalion: A Language for Language Oriented Programming. In *Proceedings of OOPSLA/SPLASH 2011*, 2011.

[11] T. Dinkelaker, M. Eichberg, and M. Mezini. Incremental concrete syntax for embedded languages. In *Proceeding of the Symposium for Applied Computing 2011*.

[12] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE 1999*.

[13] A. D. Eisenberg and G. Kiczales. Expressive programs through presentation extension. In *Proceedings of AOSD 2007*.

[14] T. Ekman and G. Hedin. The Jastadd extensible Java compiler. In *Proceedings of OOPSLA 2007*.

[15] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. In *GPCE'11*, 2011.

[16] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28, December 2002.

[17] A. S. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1), 2008.

[18] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *GPCE 03*, GPCE '03.

[19] P. Hudak. Modular Domain Specific Languages and Tools. In *ICSR '98*, jun 1998.

[20] F. Ivanicic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model Checking C Programs Using F-SOFT. In *ICCD'05*.

[21] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX 2002*. USENIX Association.

[22] G. L. S. Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3), 1999.

[23] S. Karthik and H. G. Jayakumar. Static Analysis: C Code Error Checking for Reliable and Secure Programming. In *International Enformatika Conference '05*.

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

[25] R. T. Lindeman, L. C. L. Kats, and E. Visser. Declaratively Defining Domain-Specific Language Debuggers. In *GPCE 2011*, 2011.

[26] A. Mine. Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In *ESOP 2011*, volume 6602 of *LNCS*. Springer, 2011.

[27] MISRA. Guidelines for the Use of the C Language in Critical Systems.

[28] L. Palopoli, P. Ancilotti, and G. C. Buttazzo. A C Language Extension for Programming Real-Time Applications. In *6th International Workshop on Real-Time Computing and Applications (RTCSA 99)*. IEEE CS.

[29] A. Puccetti. Static Analysis of the XEN Kernel using Frama-C. *J. UCS*, 16(4), 2010.

[30] L. Renggli, T. Girba, and O. Nierstrasz. Embedding Languages Without Breaking Tools. In *ECOOP'10*.

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

[32] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *OOPSLA 2006*. ACM, 2006.

[33] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-Based Macro System for Java. In *1st Workshop on Reflection and Software Engineering, OOPSLA '99*, volume 1826 of *LNCS*.

[34] P. R. Thrift. Common Lisp relations: an extension of Lisp for logic programming. In *1988 Internation Conference on Computer Languages*. IEEE.

[35] M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[36] H. Wu, J. G. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, editors, *SAC 2005*.