# Using C Language Extensions for Developing Embedded Software: A Case Study

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

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

Bernd Kolb, Stephan Eberle
itemis AG, Germany
{kolb|eberle}@itemis.de

## Abstract

We report on an industrial case study on developing the embedded software for a smart meter using the C programming language and domain-specific extensions of C such as components, physical units, state machines, registers and interrupts. We find that the extensions help significantly with managing the complexity of the software. They improve testability mainly by supporting hardware-independent testing, as illustrated by low integration efforts. The extensions also do not incur significant overhead regarding memory consumption and performance. Our case study relies on mbeddr, an extensible version of C. mbeddr, in turn, builds on the MPS language workbench which supports modular extension of languages and IDEs.

***Categories and Subject Descriptors*** D.3.2 [*Extensible languages*]; D.3.4 [*Code Generation*]; D.2.3 [*Program Editors*]; C.3 [*Real-time and embedded systems*]

***Keywords*** Embedded Software, Language Engineering, Language Extension, Domain-Specific Language, Case Study

## 1. Introduction

According to Ebert and Jones [12], 80% of embedded systems companies implement embedded software in C. C is good at low-level algorithms and produces efficient binaries, but it provides only limited support for defining custom abstractions. This can result in code that is hard to understand, maintain and extend. On the other hand, high-level modeling tools make it hard to effectively address the low-level aspects important for embedded software. To address this apparent contradiction, a team at itemis and fortiss has built mbeddr, an extensible version of C that comes with extensions relevant to embedded software development. At the same time, C's native constructs are available to write efficient low-level code if needed. For details on mbeddr see Section 2.

**Contribution** To provide empirical evidence to what extent the kinds of language extensions supported by mbeddr are useful, we report on a case study on the development of a smart meter (SMT). Our contribution is to analyze, in a real-life project, how the extensions affect the complexity, testability, and runtime overhead of embedded software, as well as the effort for its development.

**Audience** We target language engineering researchers (interested in empirical data justifying their work or looking to understand problems they may be able to solve) as well as embedded systems developers (seeking to understand how language extensions can help them in practice).

**Structure** We organize the paper according to the structure for case studies proposed by Runeson et al. [41] and Yin [60]. We begin by outlining the background on embedded software, language engineering, MPS and mbeddr in Section 2. In Section 3 we introduce the research questions and the collected data. Section 4 then describes the relevant context of the case study (as suggested by Dyba et al. [11]) including the hardware and software architecture, the initial artifacts and the development timeline. Section 5 provides an overview over the mbeddr-based implementation of SMT and illustrates the use of the extensions. We answer the research questions in Section 6, complemented with a critical discussion in Section 7. We wrap up the paper with related work and conclusions in Sections 8 and 9, respectively.

## 2. Background

### 2.1 Embedded Software Engineering

Embedded software controls hardware devices, often under time and memory constraints. It can be simple (lighting controls running on an 8-bit microprocessor with a few KB of RAM) or sophisticated (airplanes, missiles and process control). The amount of software embedded in devices is growing and its value for businesses is increasing rapidly [9].

According to our own experience, as well as experiences of others [5, 26, 28, 29, 48], developing embedded software poses challenges. These include meaningful abstraction while incurring little runtime overhead (because unit pricing often prohibits an increase in resources), addressing the safety and security issues incurred by C (because many embedded systems are also safety-critical), integration with various metadata (for analysis, deployment or parametrization), support for testing and monitoring (because updating deployed faulty systems is often expensive), and adhering to development processes and standards regarding requirements tracing or documentation. Together with the need for shorter time-to-market and product-line variability this makes for a challenging field.

### 2.2 Language Engineering with MPS

Language engineering refers to building, extending and composing general-purpose and domain-specific languages (DSLs) [54]. Language workbenches [13, 14] are tools for efficiently implementing languages and their integrated development environments (IDEs). The JetBrains Meta Programming System (MPS)[1] is an open-source language workbench with comprehensive support for specifying structure, syntax, type systems, transformations and generation, debuggers and IDE support (see Figure 2). MPS relies on a projectional editor. Projectional editors avoid parsing the concrete syntax of a language to construct the abstract syntax tree (AST); instead, editing gestures *directly* change the AST, and the concrete syntax is rendered ("projected") from the changing AST.[2] This means that, in addition to text, languages can also use non-parsable notations such as mathematical symbols, tables and diagrams [52]. Since projectional editors never encounter grammar ambiguities, they can support language composition [50]. Traditionally, projectional editors were tedious to use and were hardly adopted in practice. With MPS, in contrast, editing textual syntax is quite close to "normal text editing". It also supports diff-merge on the level of the projected concrete syntax. The study in [57] shows that users are willing and able to work with the editor after getting used to it.

### 2.3 C Extensions and mbeddr

mbeddr [55] applies projectional editing to embedded software engineering. Built on MPS, it provides an extensible version of C plus a set of predefined extensions such as physical units, interfaces and components, state machines and unit testing. Since extensions are embedded in C programs, users can mix higher-level abstractions with low-level C code. Developers are not forced to use the extensions; they may use them only when they consider them appropriate. mbeddr also supports product line variability, requirements traces and documentation as well as formal verification [34]. mbeddr is open-source under the Eclipse Public License and is available from `http://mbeddr.com`. Several commercial systems have been developed with mbeddr. It forms the basis for a controls engineering tool by Siemens PLM Software.

Thanks to MPS, each mbeddr extension is modular: no invasive changes to C are required to add a new extension, and multiple extensions can be seamlessly combined in a particular program. Extensions provide concrete syntax, a type system, execution semantics and IDE support. AST transformations reduce extensions to C, possibly in multiple steps. Eventually, textual C code is generated which is compiled with existing (possibly platform-specific) compilers. Users are encouraged to use MPS' facilities to define their own domain-specific C extensions. Details on building languages and language extensions are provided in [54] and [51].

## 3. Case Study Setup

The goal of this research is to find out the degree to which C language extensions (as implemented in mbeddr) are useful for developing embedded software. We adopt the case study method to investigate the use of mbeddr in an actual commercial project because we believe that the true risks and benefits of language extensions can be observed only in such projects. Focussing on a single case allows us to provide significant details about that case. To provide insight beyond this single case, we generalize analytically in section 7.4.

To structure the case study, we introduce four specific research question in Section 3.1. They are aligned with the general challenges for embedded software discussed in Section 2.1 as well as with the key non-functional requirements of the SMT case at hand. The data we collected to evaluate these research questions is introduced in Section 3.2.

The case study is not explicitly comparative. However, the implicit comparison is to the state of the practice in embedded systems development, which is the use of plain C (we briefly mention other, and in particular, model-based approaches in Section 8, Related Work). The comparison is not with an actually built alternative plain C implementation because it would be too expensive to build a production-quality second implementation. An example smart meter implementation that was available to the team was not production-ready for the reasons discussed in Section 4.4, so it would not have been a useful and fair comparison. Instead, the comparison is analytical, based on the substantial experience of several of the authors in creating embedded systems in C.

Finally, this paper does *not* consider the development of mbeddr itself (as an example of language engineering). We refer the reader to Chapter 10 of [51].

### 3.1 Research Questions

C makes it hard to create abstractions for efficiently managing the complexities associated with embedded systems. However, to ensure quality, long-term maintainability and

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

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

the opportunity for reuse, such abstractions are necessary. The first research question is thus:

**RQ-Complexity: Are the abstractions provided by mbeddr beneficial for mastering the complexity encountered in a real-world embedded system? Which additional abstractions would be needed or useful?**

Testing embedded software is challenging because of hardware dependencies, restrictions on on-device debugging and subtle timing and resource constraints. Embedded software often has few or no automated unit tests, which is a problem for quality, productivity and evolvability. Additionally, because of hardware dependencies, some problems are found only during commissioning (the process of getting the code to run on the target device). Hence, our second question is:

**RQ-Testing: Can the mbeddr extensions help with testing the system? In particular, is hardware-independent testing possible to support automated, continuous integration and build? Is incremental integration and commissioning supported?**

Most embedded software is constrained regarding available memory, processor performance or as a consequence of external timing requirements. In a trade-off between efficiency and maintainability, efficiency usually wins because of unit price constraints. Abstractions thus must not come with too much overhead (the exact magnitude of *too much* depends on the context). We capture this in question three:

**RQ-Overhead: Is the low-level C code generated from the mbeddr extensions efficient enough for it to be deployable onto a real-world embedded device?**

Independent of how useful an approach is in terms of the first three research questions, it must not require significant additional effort in the various phases of development, or it will not be adopted. This leads to research question four:

**RQ-Effort: How much effort is required for developing embedded software with mbeddr?**

### 3.2 Data Collected

Below we list the data collected to answer the research questions, taking into account that this is a real, revenue-generating industry project, and some desired data may not be available (cf. Section 7.5 on Reliability).

**RQ-Complexity** We look at the mbeddr extensions used in SMT as well as those developed specifically for the project and those identified as still missing. We qualitatively asses their impact on the complexity of the implementation.

**RQ-Testing** We investigate test coverage of the SMT implementation and discuss the test-specific SMT code. We report on the experience with commissioning the system as well as the expected effort for certification by the customer.

**RQ-Overhead** We measure the size of the system and compare it with the resources of the target hardware. We analyze the achieved performance. We also analyze the runtime overhead incurred by some of mbeddr's generators.

**RQ-Effort** We measure and discuss the effort required for developing SMT, distinguishing implementation, testing and commissioning of the system as well as the development of custom language extensions.

## 4. Case Study Context

### 4.1 What is a Smart Meter?

An electricity smart meter continuously senses the instantaneous voltage and current on a mains line using analog front ends and analog-to-digital converters. From the measured raw values, it computes various energy consumption data in physical quantities over time, most importantly RMS (root mean square) voltage and current, active, reactive, and apparent power, power factor, as well as active and reactive energy. The resulting data is displayed on an LCD display, recorded in histories, and analyzed and evaluated with regard to maximum loads, times of use, and billing periods. In addition, a smart meter communicates this data to the outside world over networks. It may also accept commands via the network. The primary success criterion for a smart meter is that it achieves a specified accuracy, verified through a certification process: a prerequisite for this is the real-time performance of the underlying computations (RQ-Overhead). In order to be a viable business, the smart meter has to be reliable, low cost, able to evolve (over time and across variants) and must be developed with an effort at or below industry average; this is reflected in RQ-Complexity, RQ-Testing and RQ-Effort. The specification for the particular smart meter developed in this project can be found in [44].

### 4.2 Hardware Architecture

The SMT target hardware consists of two MSP430 processors[3] clocked at 25 MHz. One variant of the system uses the MSP430F67791 with 256 KB Flash ROM and 32 KB RAM, the other variant uses the smaller MSP430F6736 with 128 KB Flash ROM and 8 KB RAM. One processor performs the real-time metrology, the other performs higher-level application logic and communication; the separation ensures undisturbed execution of the real-time functionality. The two processors communicate using a lightweight implementation of MQTT[4] over UARTs[5]. The application processor communicates with the outside world via RS485 and IrDA interfaces and an industry-specific communication protocol called DLMS/COSEM.[6] The system has a 7-segment LCD to show system status and measurements. The hardware was determined irrespective of the software development approach, so the implementation must cope with this hardware in terms of available resources (RQ-Overhead).

[3] `http://ti.com/ww/en/launchpad/launchpads-msp430.html`

[4] A lightweight communication protocol, see `http://mqtt.org`

[5] Universal Asynchronous Receiver/Transmitter, used in serial comm.

[6] Industry-specific data exchange messages, see `http://dlms.com/`

### 4.3 Software Architecture

The software functionality can be split into two major parts, corresponding to the two processors: low-level measurement and higher-level application functions. Figure 1 shows a more detailed breakdown. Note that the SMT-specific functionality of each of the boxes is not relevant for this paper.

No real-time operating system is used on the processors and the system is fundamentally interrupt-driven: interrupt-triggered background tasks preempt foreground tasks, which are in turn activated cyclically from the `main` function. This approach is known as one-threaded programming [40].

The interrupt-triggered tasks include reading the raw measurements (triggered by the `ADC` interrupt[7]) as well handling recalibration requests (triggered by `UART-Receive`). Interrupt-triggered tasks preempt the foreground tasks and always run to completion, which makes them time-sensitive. This is why the measurement task only performs simple calculations and then uses message passing to hand the data off to a foreground task that performs more sophisticated calculations involving expensive division and square roots. Other foreground tasks include calibrations as well as the UART-MQTT-based inter-processor communication.

In terms of performance, the challenge is to ensure that the background tasks finish within less than a $\frac{1}{4{,}096}$ of a second to maintain the required 4,096 Hz sample rate, and to leave enough time for the foreground processes to finish their tasks within their own time budgets (one second for the calculations mentioned above).

### 4.4 Smart Meter Example Code

The SMT development team had access to an existing example smart meter implementation for the MSP430, made available by the processor vendor.[8] The purpose of this example smart meter code (ESC) is to serve as a realistic, but incomplete implementation of a smart meter on the MSP430. The ESC comprised only a subset of the functionality needed for SMT and required significant extension (for example, to run it on two processors and to support more flexible communication stacks; see Section 6.4), so a thorough understanding of the code was necessary. A document describing the high-level structure of the ESC was also available.

The SMT team decided that the code quality (understandability, modularity, maintainability, testability and test coverage) of the ESC was unacceptable for sustained SMT development; they decided to build a completely new implementation of SMT using mbeddr. Only the core algorithms were taken from the ESC; SMT is otherwise new software.

### 4.5 Development Timeline and Process

Development started in July 2012. As of February 2015, most of the required functionality is implemented, but development and certification are ongoing. The project used an iterative process based on a specification [44] that is updated approximately once per year. Integration with the target hardware started in February 2014 and was spread over 2 months. So far, 300 person days (PD) were spent, spread over 31 months, a 50% developer utilization. Fulltime work was not feasible because of constraints in project funding, decision gateways and requirements elicitation.

| Criterion | Common | Metro | App | Total |
|---|---|---|---|---|
| # of Files | 134 | 101 | 105 | 340 |
| Total LOC | 8,209 | 10,447 | 10,908 | 29,564 |
| Code LOC | 4,397 | 5,900 | 5,510 | 15,807 |
| Comment LOC | 950 | 2,402 | 2,620 | 5,972 |
| Whitespace LOC | 2,852 | 2,145 | 2,778 | 7,775 |

**Table 1.** Size of SMT. *Common* code runs on both processors, *Metro* runs on the metrology processor and *App* runs on the application/communication processor.

### 4.6 Tools

In addition to mbeddr, SMT used gcc and gdb (for compiling and debugging on the PC) and the IAR Embedded Workbench[9] and associated hardware-specific compilers.

## 5. The mbeddr SMT Implementation

### 5.1 Overall Structure

Figure 1 shows the structure of SMT. Each of the small boxes represents one or more mbeddr components in the source code. SMT consists of a hardware-dependent Hardware Abstraction Layer (HAL) as well as hardware-independent communication stacks (COMM), utilities (UTIL) and functionalities for the actual measurement (METROLOGY) and higher level computations and external communication (APPLICATION). The separation into hardware-dependent and hardware-independent layers is a prerequisite for testing of components on the PC (RQ-Testing).

### 5.2 Size of the System

The SMT implementation consists of code that is deployed onto the target as well as code that is used only for testing. Table 1 shows the size of the deployed code in terms of *generated* C; it is ca. 22,000 non-empty lines of code (LOC). The additional test code is roughly similar in size, resulting in 44,000 LOC in total. While this is small compared to automotive, aerospace or defense systems, its size is typical for software found in industrial sensors, AUTOSAR basic software or Internet-of-Things devices.

Table 2 shows the number of instances of important language concepts. Because projectional editing can use non-textual notations, counting lines is not easily possible and we use a conversion factor similar to the one in [53] to calculate the LOC for language constructs; this leads to ca. 42,000 non-empty LOC total, a size roughly similar to the generated

[7] `ADC` is short for Analog-Digital Converter

[8] `http://www.ti.com/tool/msp430-energy-library`

[9] `http://www.iar.com/Products/IAR-Embedded-Workbench/`

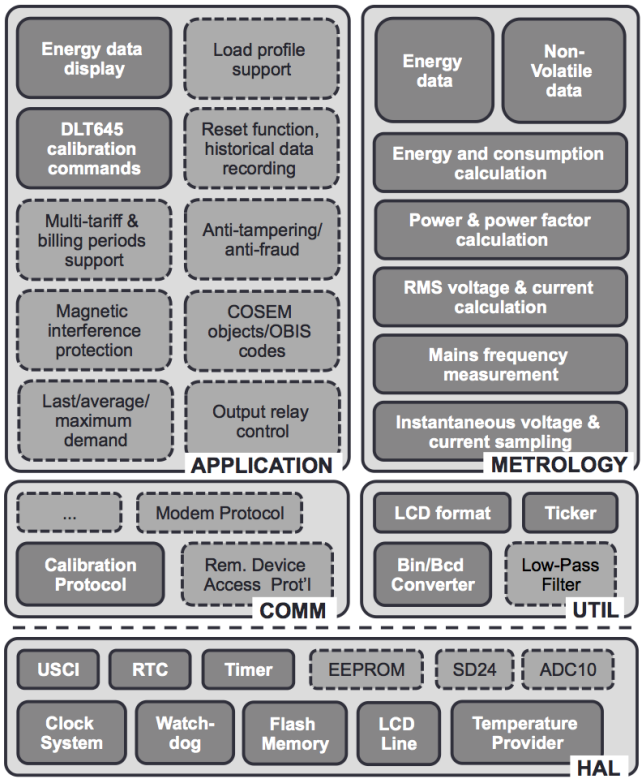


**Figure 1.** Layers, subsystems and components in SMT. The dashed-border components are optional. Only the HAL subsystem (below dotted line) is hardware-dependent.

C code. This demonstrates that mbeddr's extensions do not lead to a significant reduction in code size: they trade boilerplate in some places for well-structuredness, readability, analyzability and maintainability in others.

### 5.3 Use of mbeddr's Built-in Extensions

Table 2 shows that SMT makes use of all major mbeddr C extensions, indicating their relevance for embedded software, as well as their composability. The rest of this subsection introduces mbeddr's extension in some detail. The code examples in this subsection are kept simple for reasons of space and do not show all features of the respective extension; more details on and bigger examples of the extensions can be found in [55] and [51]. Figure 2 shows a screenshot of the mbeddr IDE with some of the languages and notations.

**Chunks** mbeddr structures code into chunks; a chunk can be seen as (and is often generated into) a single file. There are chunks for units and conversion rules, chunks for requirements, chunks for feature models, and chunks for (extended) C code called implementation modules (generated to a .c and a .h file each). Chunks also act as namespaces and are the primary means for structuring mbeddr code. SMT has 382 implementation modules and 46 other chunks.

**C Constructs** mbeddr supports almost all C language constructs (the few exceptions are discussed in [53]). 310 functions, 144 structs, 334 global variables and 8,500 constants

| Category | Concept | Count |
|---|---|---|
| Chunks | Implementation Modules | 382 |
| (≈ Files) | Other (Req, Units, etc.) | 46 |
| C Constructs | Functions | 310 |
| | Structs / Members | 144 / 270 |
| | Enums / Literals | 150 / 1,211 |
| | Global Variables | 334 |
| | Constants | 8,500 |
| Components | Interfaces / Operations | 80 / 197 |
| | Atomic Components | 140 |
| | Ports / Runnables | 630 / 640 |
| | Parameters / Values | 84 / 324 |
| | Composite Components | 27 |
| | Component Config Code | 1,222 |
| State Machines | Machines | 2 |
| | States/Transitions/Actions | 14 / 17 / 23 |
| Physical Units | Unit Declarations | 122 |
| | Conversion Rules | 181 |
| | Types / Literals with Units | 593 / 1,294 |
| Product Line Variability | Feature Models / Features | 4 / 18 |
| | Configuration Models | 10 |
| | Presence Condition | 117 |
| Custom Extensions | Register Definition | 387 |
| | Interrupt Definitions | 21 |
| | Protocol Messages | 42 |
| Statements | Statements total | 16,840 |
| | Statements in components | 6,812 |
| | Statements in test cases | 5,802 |
| | Statements in functions | 3,636 |
| Testing | Test Cases / Suites | 107 / 35 |
| | Test-Specific Components | 56 |
| | Stub / Mock Components | 9 / 8 |
| | `assert` Statements | 2,408 |

**Table 2.** Number of instances of language concepts in the mbeddr SMT sources (before generation to C text).

are used in SMT, the large number of constants being typical for embedded software. As discussed below, most of the SMT implementation is factored into components; however, 3,636 statements (ca. 22%) remain in functions. These are mostly mathematical utilities and filters, conversions, safe access to memory, and test helper functions.

**Components** Components form the backbone of the SMT implementation (and most other mbeddr-based systems). Components are modularized units of behavior, specified via interfaces. Interfaces either define operations (callable through required ports and implemented via provided ports) or data items (received and sent through provided and required ports). Here is an interface that defines one operation:

```
// ADC is the analog-digital converter
interface IADC {
  int16 read(uint8 addr)
}
```

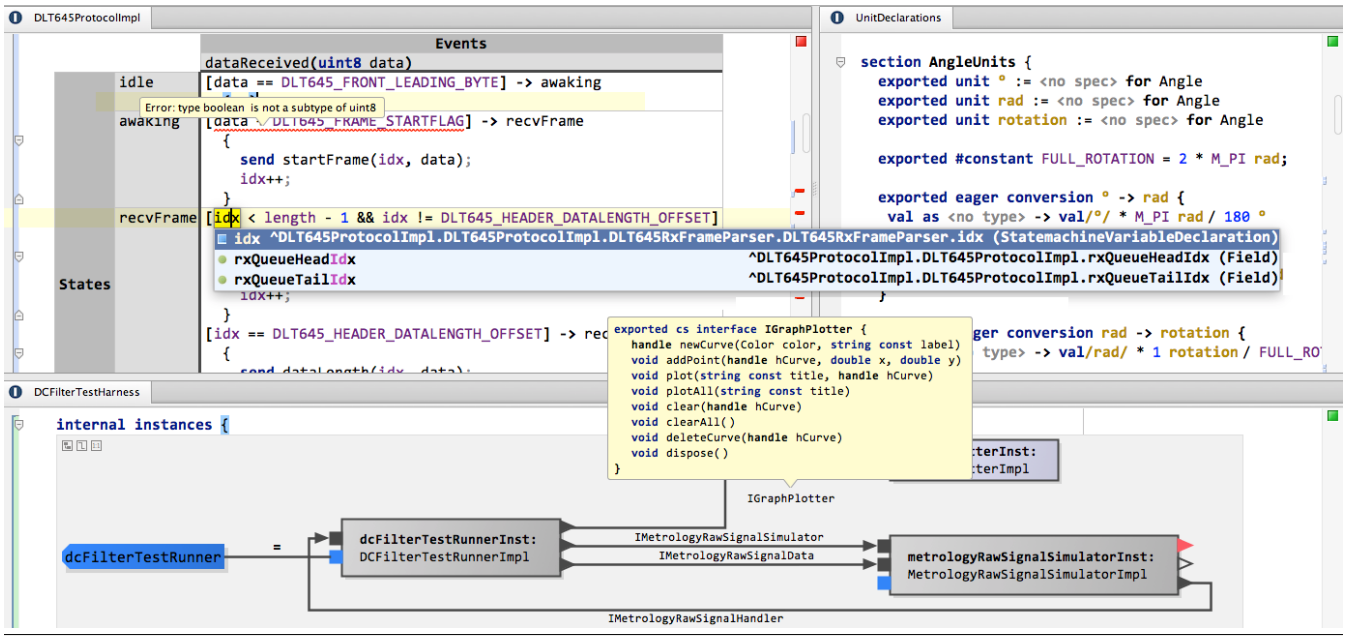


**Figure 2.** The screenshot shows various parts of the SMT implementation: a part of the protocol parser state machine (top left), unit declarations (top right) and component wiring for a test case (bottom). It also illustrates how mbeddr provides IDE support for C and its extensions, including syntax highlighting, code completion, error markup, refactorings, quick fixes and tooltips. The screenshot also showcases the support for mixed notations (text, tables, diagrams).

Components provide and require ports. Each port is associated with an interface. Components implement the operations of the interfaces associated with provided ports in runnables, essentially C functions inside components. SMT has 80 interfaces, 167 components and 640 runnables. Here is a component `ADCDriver` that provides the `IADC` interface:

```
component ADCDriver {
  provides IADC adc
  int16 adc_read(uint8 addr) <= op adc.read {
    int16 val = // low level code to read from addr
    return val;
} }
```

A client component can now declare a required port that uses the `IADC` interface. Implementation code in runnables can call operations on this required port:

```
component CurrentMeasurer {
  requires IADC currentADC
  internal void measureCurrent() {
    int16 current = currentADC.read(CURR_SENSOR_ADDR);
    // do something with the measured current value
} }
```

Components must be instantiated to be used, and their required ports connected to interface-compatible provided ports of other instances. This can be edited graphically (inline in a "C editor"), as shown in the bottom pane of Figure 2.

mbeddr also supports composite components, enabling hierarchical decomposition of systems (they contain their own set of instances). Of the 167 components, 27 are composite components. The code that instantiates, parametrizes and connects ports of components instances comprises 1,222 LOC (for deployment and multiple test setups).

The majority of SMT behavior resides in components: of the 16,840 total statements, 40% live in component runnables, 22% are in regular C functions (discussed above) and 34% are in test cases; the remaining 4% reside in state machines and a few other places. On average, each runnable consists of 11.5 LOC. The cyclomatic complexity of each runnable is low; the average is 1.98.

**State Machines** State machines encode state-based behavior, and they live inside implementation modules, alongside C code or components. Textual, graphical and tabular syntax is available for any given state machine via multiple projections. SMT is not primarily a state-based system, so the use of state machines is limited to two examples. One implements the communication protocol and message parsing, a typical use case for state machines. The other one drives the display: since the display has limited real estate, its contents change based on various parameters, events and system states. The state machine tracks these changes and updates the display. Here is a very much simplified example of the state machine used for message parsing:

```
statemachine FrameParser initial = idle {
  var uint8 idx = 0
  in event dataReceived(uint8 data)
  state idle {
    entry { idx = 0; }
    on dataReceived [data == LEADING_BYTE] -> wakeup
  }
  state wakeup {
    on dataReceived [data == START_FLAG]
       -> receivingFrame { idx++; }
  }
  state receivingFrame { .. }
}
```

State machines can be used as types in C. For example, the code below shows a local variable of type `FrameParser`. Built-in operators are available to interact with them:

```
// create and initialize state machine
FrameParser parser;
parser.init;
// trigger dataReceived event for each byte
for (int i=0; i<data_size; i++) {
  parser.trigger(dataReceived|data[i]);
}
```

**Physical Units** C types and literals can be annotated with physical units. New units can be declared based on existing units and conversion rules between different units can be defined. The type system then performs unit computations and checks. Figure 3 shows an example.

In SMT, which measures and samples real-world quantities and uses other quantities for calibration, units provide an additional level of checks that cannot be provided by just data types. Based on the 7 SI units available by default, SMT has 122 unit declarations and 181 conversion rules (units with different magnitudes such as `km` or `mm` count as different units in mbeddr). 593 types are annotated with a unit (in local or global variables, constants or arguments) and 1,294 numeric literals in the code have a unit associated with them.

**Testing** mbeddr has first-class support for assertions, unit tests, and test suites. Below is an example that contains test cases for the `FrameParser` state machine plus a `test` expression (which represents test suites):

```
testcase testFrameParser1 {
  FrameParser p;
  assert(0) p.isInState(idle);
  // invalid byte; stay in idle
  parser.trigger(dataReceived|42);
  assert(0) p.isInState(idle);
  // LEADING_BYTE, go to awakening
  parser.trigger(dataReceived|LEADING_BYTE);
  assert(0) p.isInState(awakening);
}

testcase testFrameParser2 { ... }
testcase testFrameParser3 { ... }

int32 main(int32 argc, char* argv) {
  return test[testFrameParser1,
              testFrameParser2,
              testFrameParser3];
}
```

```
unit V :=      for voltage
unit A :=      for Amps
unit Ω := V·A^-1 for resistance

uint16/Ω/ resistance(uint16/V/ u, uint16/A/[] i, uint8 ilen) {
  uint16/A/ avg_i = (sum_{p=0}^{ilen} i[p]) / ilen;
  return avg_i / u;   Error: type uint16 /V^(-1) · A/ is not a subtype of uint16 /Ω/
} resistance (function)
```

**Figure 3.** Example of physical units in SMT. Assigning a value with unit $\frac{A}{V}$ to the return type with unit $\Omega$ results in an error in the IDE. Note also the use of mathematical syntax.

mbeddr also supports constructs for efficiently writing tests for some of the other extensions. The most important one are mock components, which use special syntax for specifying expected behavior as part of a test case. The mocks can then be validated in a test case. Below is an example of a mock component for a protocol handler that specifies operation sequencing, assertions over parameters and also remembers the `handle` argument so it can be closed later:

```
mock component USCIReceiveHandlerMock {
  provides ISerialReceiveHandler handler
  Handle* hnd;
  sequence {
    step 0: handler.open { } do { hnd = handle; }
    step 0: handler.dataReceived {
        assert 0: parameter data: data == 1 }
    step 1: handler.dataReceived {
        assert 1: parameter data: data == 2 }
    step 2: handler.dataReceived { .. }
    step 3: handler.dataReceived { .. }
    step 4: handler.finsihed { } do { close(hnd); }
} }
```

SMT has 107 test cases in 35 test suites, with over 2,400 `assert` statements. 56 of the 167 components are specific to tests. Of those, 8 are mocks and 9 are stubs. As discussed in Section 5.4, two of the three custom extensions were developed to simplify testing.

**Variability** Feature models are an established formalism for expressing product line variability [3]. A feature model consists of a tree of features with constraints between them. Constraints include `mandatory` (feature must be in any valid system), `optional` (feature may not be in a system), `or` (one or more from a group of features may be in a system) and `xor` (exactly one of a group of features must be in a system). A feature may have attributes, and additional cross-tree constraints may be specified. The code below is one of the feature models from SMT, expressed in mbeddr's textual notation for feature models. It handles the variability associated with different LCD displays and configurations.

```
feature model SMTFeatures
  root opt
    Data_LEDs opt
      DataReadLED
      DataWriteLED [DigitalIOPortPin pin]
    DISPLAY xor
      DISPLAY_V10
      DISPLAY_V22
    WRITABLE_FLASH_MEMORIES
```

The features (and hence, the variability expressed) in a feature model can be connected to implementation code through presence conditions. A presence condition is a Boolean condition over the features from a feature model attached to a part of a program; only if the condition evaluates to true for the selected product configuration will the corresponding code fragment be included in the program. Presence conditions are roughly similar to C's `#ifdef`, but they are more structured, because they operate on MPS' syntax tree: they cannot lead to syntactically invalid code. Figure 4 shows an example of a presence condition used on component ports.

In SMT, mbeddr's variability support was used to implement 4 different feature models (metrology, platform, display variability, LED variability) with 18 features in total. 10 different configurations were defined for deployment and test setups. 117 presence conditions are used throughout the code. SMT also used the built-in consistency analysis which ensures that no variant contains dangling references: it checks that for every reference in the code (e.g., a reference to a variable), the referenced node (e.g., the variable) is part of (at least) all configurations that contain the reference.

```
exported composite component MetrologyPlatformLayer {
  provides IWatchdogTimer watchdogTimer
  ? {DataReadLED && WRITABLE_FLASH_MEMORIES}
  ?provides IDigitalOutputPin pin1
  ? {DataWriteLED}
  ?provides IDigitalOutputPin pin2
```

**Figure 4.** A part of a composite component where two of its provided ports have presence conditions (the gray area marked with question marks). The ports are only part of the system if the respective features are selected.

### 5.4 Custom Extensions

mbeddr encourages user-defined, project-specific extensions to grow the language towards a domain [46]. For SMT, three extensions have been developed; below we introduce the extensions and the specific rationales for developing them.

**Registers** The MSP430 processor has special-purpose registers: when a value is written to such a register, a hardware-implemented computation is automatically triggered based on the value supplied by the programmer. The result of the computation is then stored in the register. The reason for developing a custom extension is testability. In particular, running code that works with these registers on the PC for testing purposes leads to two problems: first, the header files that define the addresses of the registers are not valid for the PC's processor. Second, there are no special-purpose registers on the PC, so no automatic computations are triggered. SMT solves this problem with a language extension that supports the definition of registers as first-class entities and allows read/write access from C code (see code below). The extension also supports specifying an expression that performs the computation. When the code is translated for the real device, the real registers are accessed based on the addresses defined in the processor header files. In the emulated case used in testing, generated `structs` are used to hold the register data; the expressions are inserted into the code that updates the struct, simulating the hardware-based computation.

```
exported register8 ADC10CTL0 compute as val * 1000

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

**Interrupts** As explained in Section 4.3, SMT is driven by interrupts. To integrate the component-based architecture used in SMT with interrupts, it is necessary to be able to trigger component runnables via an interrupt. Similar to registers, the primary driver was testability: interrupts must be emulated for testing on the PC. A language extension allows the declaration of interrupts. In addition, the extension provides runnable triggers that connect the execution of the runnable to the occurence of an interrupt. The example below declares two interrupts, and the runnable `interruptHandler` is marked as triggered by an interrupt:

```
module USCIProcessor {
  exported interrupt USCI_A1
  exported interrupt RTC

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

Note that this code does not specify *which* interrupt triggers the runnable, because, for reasons of deployment flexibility, this is done as part of component instantiation, as shown below. Instantiation also checks that each interrupt-triggered runnable is bound to at least one interrupt. In addition, for testing purposes on the PC, there are language constructs that simulate the occurrence of an interrupt: test cases then simulate triggering of interrupts based on a test-specified schedule, and assert that the system reacts correctly.

```
instances usciSubsystem {
  instance RTCImpl rtc;
  bind RTC -> rtc.interruptHandler
  connect ... // ports
}
```

**Messages** External communication of the SMT device takes place via DLMS/COSEM messages. The low level protocol definition involves arrays pointing into other arrays, linked lists, multi-byte identifiers, fields that contain the size or number of other fields as well as other fine-grained, low-level details. Below is an example (DLMS/COSEM is even more complex, but the example below illustrates the challenges). SMT contains hundreds of message definitions.

```
// a field representing a timestamp for 10:20:00
uint8[6] f_time = {0x00A,  // field type identifier
                   UNIT_TIME24, // unit used: time
                   3, // 3 payload bytes follow
                   10, 20, 00 // the time itself
                  };

// a field representing a measured value
uint8[4] f_value = {0x04D,  // field type identifier
                    UNIT_QDOT, // unit used: mass flow
                    1, // 1 payload byte follows
                    &dataField // addr of variable
                   };

// a message that uses the two fields
uint8[5] message = {0xAEE, // message type idenfier
                    ID, // unique running message ID
                    2, // two fields folllowing
                    f_time, // embed the time field
                    f_value// embed the value field
                   };
```

It is tedious and error prone to set up these structures manually, so the primary driver for this extension is robust-

ness and mainainability. The extension supports a higher-level syntax for defining messages, plus a code generator that generates the low-level details. Each message has a name (`CurrentMeasuredValue`), a unique numeric identifier (42) and a number of fields:

```
message CurrentMeasuredValue:42 {
  int32     timestamp; // time of measurement
  uint16/A/ value;     // measured value in Amps
  uint16    accuracy;  // accuracy in 1/100 %
}
message ... { ... }
...
```

In terms of interactions, the SMT is passive; it is queried by external systems. Consequently, there are no `send message` statements in the SMT code that supply values for the message fields. Similarly, the set of messages supported by SMT depends on the configuration. The configuration, in turn, is determined by the set of deployed components. This is why we associate the code that provides data for the messages with components. We have extended the component language to support `message data` specifiers in addition to the existing component contents (such as runnables and fields). They are generated to callback functions. The example below illustrates that message values can either be constants (`100`), pointers to variables (`&lastValue`) or function references (`:currentTime`), mbeddr's cleaned up version of function pointers.

```
atomic component CoreMeasurer {
  field uint16/A/ lastValue = 0;
  message data 42 {:currentTime, &lastValue, 100};
  void measure() {
    lastValue = // perform actual measurement
} }
```

## 6. mbeddr Evaluation

From Table 2 we see that the extensions are used extensively to address the challenges in SMT. In this section we investigate this in more detail by evaluating the research questions introduced earlier relative to mbeddr's use in SMT.

### 6.1 RQ-Complexity

**Improved Structure using Components** Components have been used extensively, as illustrated by Table 2 and Section 5.3, Components. All the small boxes in Figure 1 have been implemented as components. This helps break down the system into smaller units, which in turn helps understanding and reasoning over each component in isolation. Interfaces provide a contract between the provider and consumer of the service specified by the interface. Composite components support a hierarchical breakdown and incremental composition. This helps with understandability and enables a structured approach to variability, which in turn facilitates a platform-based architecture.

**Platform and Variability** A platform-based approach relies on reusable modules from which similar (but not identical) systems are developed. Custom code is typically composed with the reusable modules – implemented as components in SMT. Circa 80% of the code has been factored into the platform, and the team expects future projects to reuse (parts of) the platform. Referring to Figure 1, the platform code includes all of HAL, COMM and UTIL as well parts of APPLICATION and METROLOGY.

However, since the product-line that is built on the platform consists of similar, but not identical products, it must be possible to express variability. In SMT, coarse-grained variability is realized by combining components in different ways. This is enabled by the components' support for polymorphism (different implementations of an interface) and multiple instantiations of a component (similar to objects in an object-oriented language). According to the developers, this came close to the vision of "Lego$^{TM}$-like software assembly". For example, the metrology and communication stacks can be used with different processors by combining them with different HAL components.

For finer-grained variability, components support parameters. These are declared as part of the definition of a component and are supplied with values when the component instance is defined. This mechanism has been used extensively in SMT: 50 components have a total of 84 parameters set to 324 different values. Examples are the conversion factors between raw measurements and the physical values.

Presence conditions (Section 5.3, Variability) are used sparingly; only 117 presence conditions are used for fine-grained variability. This is in contrast to most embedded software, including the ESC, which is typically laced with `#ifdef`s, leading to a variety of problems in terms of analyzability and IDE support [27, 32].

**Additional Type Checks with Units** The algorithms adapted from the ESC contained several errors that would have been found if physical units, were available in the type system. An example is the "calibration" of a temperature `T` through `T = T * T + offset;`. Obviously, this is wrong because, assuming the temperature was measured in $K$ (Kelvin), the right hand side unit would be $K^2$, which cannot be assigned to $K$ on the left side. mbeddr's support for physical units (Section 5.3, Physical Units) detected this and other similar errors (the variable `T` would be declared as `double/K/ T;`). Note that `typdef`'ed C primitive types are not enough because the C type system cannot calculate with units (as in $K^2 \equiv K * K$ or $\Omega \equiv \frac{V}{A}$).

**Custom Extensions** The extension for messages introduced in Section 5.4, Messages, prevents low-level mistakes in these arrays and hence removes accidental complexity. The extensions for registers and interrupts (Section 5.4, Registers and Interrupts) also reduce complexity since they encapsulate the variability necessary for switching between the variants for the target device and for testing on the PC; no explicit `#ifdef`-like variability is necessary in the code.

**Missing Extensions** The SMT team has identified the need for additional mbeddr extensions. This need is itself a con-

firmation of mbeddr's approach since it demonstrates that developers realize that language extensions can be used to solve real problems. What follows is a "wish list" expressed by the SMT developers.

Extensions for queues, stacks and ring buffers with very limited overhead and OO-style syntax (`stack.pop`) would be useful. Support for Q-formatted fixed point numbers would help for platforms without a floating-point unit. The SMT team also suggested extensions for testing and debugging, and some of them have been prototyped in SMT. These include extensions for signal analysis (plotting of signal sequences as graphs), tracing of signal sequences in the target system (instrumented code, UART communication, visualization/evaluation on host PC) as well as performance profiling (with port pin toggling, oscilloscope on host PC).

The SMT developers suggested additional checks, some done by the type system and others performed at runtime by optionally generated code: (1) detection/avoidance of unnecessary initialization of RAM variables to prevent watchdog resets during startup, (2) detection of missing `volatile` usages, (3) detection of automatic promotion of signed to unsigned integers, and (4) detection of word accesses to byte-aligned (odd) memory addresses.

Finally, a language for specifying initialization parameters and constraints between the parameters was suggested to streamline component configuration. Currently, primitive C types or `structs` are used for this purpose.

None of these extensions are SMT-specific, so they will be developed as part of mbeddr's evolution. Some of them (such as the ring buffers and stacks) have already been implemented at the time of this writing.

**Unused mbeddr Extensions** mbeddr ships with more extensions than those used in SMT. The two most important ones are requirements and requirements tracing [56] as well as formal verification [34]. The requirements and requirements tracing was not used because it was not requested by the customer. The verification support could have been used to statically check contracts of interfaces. It was not used because of the team's unfamiliarity with formal verification and the realization that unit testing was enough to ensure quality. Both of these extensions have been used successfully in other projects, though. In particular, static verification is an important ingredient of the controls engineering tool currently developed by Siemens PLM Software.

**Notation and Readability** Once the right abstractions are in place, these can be rendered with intuitive notations by MPS' projectional editor. SMT has made use of mathematical notations such as sum, square root or fraction bar symbols in some places. Also, component instances and connections are rendered graphically (similar to UML composite structure diagrams, see Figure 2): SMT has 72 instance configurations with on average 5 instances and 8 connectors (the top size quartile has 10 instances and 19 connectors) to set up the test scenarios. These structures are much more accessible with a graphical notation. Generally, by using easily recognizable first-class language constructs for domain-relevant concepts (such as registers, message definitions or state machines), readability is improved.

We summarize as follows regarding RQ-Complexity:

> The developers naturally think in terms of extensions, and suggested additional ones during the project.
>
> mbeddr components help structure the overall architecture and enable reuse and configurability.
>
> mbeddr extensions facilitate strong static checking, improve readability and help avoid low-level mistakes.

### 6.2 RQ-Testing

**Components and Testing** Componentization simplifies testing because well-defined behavioral units are available that can be unit-tested individually. The 107 test cases, 2,400 assertions and 56 test-specific components (including the 9 stubs and 8 mocks) lead to a test coverage (line coverage) of 80% for the critical metrology subsystem and ca. 40% for the less challenging application parts.[10] The state-of-the-practice in industry (outside of safety-critical domains) achieves much lower unit test coverage and relies heavily on hardware-in-the-loop test, often for whole systems, and often executed manually [43].

**Automated, Hardware-Independent Testing** Hardware-independent testing refers to the ability to run unit tests for functional requirements on the developer's PC and on a continuous integration (CI) server. In SMT, this was facilitated by isolating the hardware-dependent functionality using components and interfaces. Efficiently testing SMT requires hardware-independent testing for two reasons. First, the SMT hardware only became available after the software development had started. Second, it enables continuous build, test and integration on a CI server. The latter is well established in software engineering and is known to increase quality and reduce integration time [15]. However, in embedded software, it is still rarely used (as are agile processes in general [43]). In SMT, all the hardware-independent parts were continuously built, tested and integrated with the Teamcity CI server.[11]

The focus on testing paid off: only 13% of the total effort were spent on integration. For embedded software, this is very low: Sztipanovits [47] puts the number at 40% to 50%, and Broy calls integration a "major challenge" and a "nightmare" [5].

A common alternative to hardware-independent testing is the use of a simulator. Simulators, if available, are supplied by the hardware vendor and faithfully simulate the behavior

[10] The team stopped adding unit tests once the overall acceptance test suite (during integration and commissioning, covering the key functional and non-functional requirements) passed.

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

of the hardware in software, executing the hardware-specific binaries. While this is useful for some tests, it is not a replacement for automated unit testing on a CI server: simulators are hard to integrate into a CI build process because they are typically not designed to be used in an automated, non-interactive way.

**Testing Hardware-Dependent Parts** Some tests that involve hardware specifics were performed on the PC nonetheless, exploiting the improved testability provided by the register and interrupt extensions: both can be generated in a way that emulates special-purpose registers and the occurrence of interrupts. This contributed to the low testing effort of 16%, because formerly manually executed, hardware-specific tests could now be executed as part of the automated test suite.

Some aspects (e.g., timing) had to be tested manually on the target device once it was available. In one case a broken checksum on the serial interface led to a failed test. Oscilloscope-based low-level debugging revealed that the last byte was not sent because of a timing issue: the `TransmitEnable` interrupt, which triggers sending of each byte, was disabled before the last byte was sent. Generally, component-based incremental integration helped identify (hardware-dependent) components that created problems.

**Incremental Integration and Commissioning** Commissioning refers to the steps necessary between finishing the implementation on the PC and getting it to run on the target device. Despite extensive hardware-independent testing, this is challenging: any number of problems can occur in the overall system as a consequence of timing and resource allocation interactions as well as hardware configuration.

Tracking down such problems is simplified by commissioning the system in steps. Initially, only a minimal system is deployed, one that performs only a single task and requires only limited resources. The system is then grown incrementally to the full scope. Occuring problems must be related to the parts added during the last step. The causes for problems can also be narrowed down by iteratively deploying *different* subsets of the full system (as opposed to linearly growing the full system). Components helped with this process because it is easy to compose subsets of the total set of componenrs for commissioning. As an example, consider inter-processor communication: it transports both measurement data and calibration data. In the fully deployed system, these two kinds of data are multiplexed over the single connection. Commissioning happened in four steps: calibration data only, measurement data only, then both together, and finally both together with the real metrology (which changes the timing).

The commissioning of the minimal SMT system was painless and only one problem was found: an explicit cast between 16 bit and 32 bit integers was missing for the 16 bit target. This is in stark contrast to the team's previous experience, where integration often meant that developers debug their way forward, bug by bug.

**Certification** The SMT device must be certified by government agencies regarding its accuracy. This is done by subjecting the finished product to calibrated signals, measuring the achieved accuracy. Certification has not been done yet. However, the certification-relevant signal scenarios are known and they are continuously exercised with software-in-the-loop tests. This builds confidence in the readiness for certification, which in turn avoids premature (and expensive) certification attempts. By making these tests part of the regression test suite, one can also avoid problems with certification as the software evolves. Testing is done by using stub and mock components in place of some of the drivers. In particular, the ADC driver is replaced by a signal simulator that generates changing values over time. Assertions are used to signal inaccuracies.

Regarding RQ-Testing we summarize the SMT experience as follows:

> mbeddr components are instrumental in improving testability through clear interfaces and small units, leading to 80% test coverage for core components.
>
> The custom extensions and the components facilitate hardware-independent testing, continuous integration and automated dry runs of the certification process.
>
> The modularization facilitated by components helps track down problems during commissioning.

### 6.3 RQ-Overhead

**Memory Consumption** In terms of memory consumption, the mbeddr-generated binary is small enough for the target device: the fully configured system for the metrology processor uses 16,736 bytes of flash ROM and 4,321 bytes of RAM. For the application processor, these figures are 10,978 and 2,917 bytes, respectively (both cases refer to non-optimized debug code, which means the production binary will be smaller). 512 KB of flash and 32 KB of RAM are available, so future growth is possible. Note that no dynamic memory allocation (`malloc`, `free`) is used as the program runs. All memory is acquired statically at runtime or resides on the stack, as is common in embedded systems.

The componentization is also useful in the context of memory consumption: since the system is split into (relatively) small components with clear dependencies, and since the mbeddr component generator does not generate C code for components that are not instantiated as part of a given executable, one can avoid deploying unnecessary functionality and avoid consuming resources that are not necessary for a given product variant. On the flip side, the components also produce some overhead: the binary uses 2,804 bytes of ROM and 2,647 bytes of RAM to hold the data structures and pointers that represent component instances and their connections. While not a problem for SMT, Appendix B discusses how this overhead can be reduced.

| Development Tasks | Effort | % Total |
|---|---|---|
| Implementation | 200 PD | 66% |
| Reimplementation | 145 PD | 48% |
| Additional Functionality | 55 PD | 18% |
| Tests, Simulators | 48 PD | 16% |
| Integration & Commissioning | 38 PD | 13% |
| Custom Language Extensions | 14 PD | 5% |

**Table 3.** Breakdown of the SMT development effort.

**Performance** We group language extensions relative to overhead and identify three categories. In the following paragraphs we relate the language constructs used in mbeddr from Table 2 to these categories.

The first category has no runtime footprint, because the respective extensions are removed before code generation. In SMT, physical units (only relevant for the type system) and the product line variability (presence conditions are evaluated statically, similarly to `#ifdefs`) belong to this category.

The second category has a footprint similar to manually written, idiomatic C code. Most of the mbeddr and custom extensions belong to this category: state machines are reduced to a `switch`-based implementation; registers become direct memory access based on `#defines`; interrupts are reduced to interrupt handler functions; protocol message definitions are reduced to their native, array access-based form; all C constructs are transformed to C text without change.

Category three requires more sophisticated code structures to be generated and leads to some performance overhead. In SMT, components are the only extension that falls into this category: interface polymorphism is handled via function pointers, which introduces a performance penalty because of the indirection. Appendix B discusses existing and future optimizations to reduce this overhead. Since the optimizations trade flexibility for efficiency, and because the optimizations were not necessary to run the SMT software on the intended hardware, the team decided not to use them.

Runtime performance was paramount for SMT to achieve the 4,096 Hz sample rate, which directly influences the measurement accuracy. The implementation achieved the required sample rate, which is testament to the limited runtime overhead. In terms of overhead, we summarize:

> The memory requirements of SMT are low enough for it to run on the intended hardware, with room for growth.
>
> Componentization enables deployment of only the functionality necessary for a variant, conserving resources.
>
> The performance overhead is low enough to achieve the required 4,096 Hz sample rate on the given hardware.

### 6.4 RQ-Effort

Developing and integrating the SMT software consumed 300 person days (PDs) overall. Table 3 breaks the 300 PDs down into the different development tasks.

66% of the total effort (200 PDs) were used for the actual implementation of SMT. 48% (145 PDs) were required for a maintainable and extensible implementation of the ESC functionality in mbeddr. Since no C code importer was available, even the algorithmic code that should be reused from the ESC had to be retyped into mbeddr. An importer is available now; however, the team estimates that only ca. 10 PDs could have been saved if it had been available earlier, because, as mentioned before, the vast majority of the ESC had to be redone completely.

18% of the overall effort (55 PDs) were necessary to implement additional functionality required by the specification that goes beyond the ESC. This includes

- the ability to run on two processors,
- the communication between the two processors,
- increased flexibility of the communication infrastructure (multiplexing between calibration data and MQTT, two communication technologies RS485 and IrDa),
- an I2C Bus driver,
- an EEPROM controller,
- a subset of the required DLMS/COSEM messages
- as well as additional application functionality such as historical data recording and reset functionality.

Since, at this point, the core system was already in mbeddr and structured into components, integrating this additional functionality was straightforward; we consider 55 PDs for this additional functionality a low figure. Of these 55 PDs, ca. 20 were required for the MQTT implementation. Once this was done, the distribution over two processors was a matter of a few hours.

16% of the total effort (48 PDs) were spent on unit and integration tests and the required test harness, specifically, mock components and signal generator components.

13% of the overall effort (38 PDs) were spent on integration and commissioning onto the target hardware and on validation of the non-functional properties (performance, resource consumption). As mentioned before, this figure is very low for embedded software. The team attributes this to the componentization and the custom register and interrupt extensions, which made extensive, automated testing feasible and facilitated incremental commissioning.

It took 5% of the effort (or 14 PDs) to build the custom extensions. Considering the benefits of the extensions for testing and maintainability, the SMT team considers this effort well spent. Also, the number is small enough to make custom extensions a realistic option for real projects.

Based on mbeddr's promises and some initial experience with mbeddr, the team originally estimated 250 to 290 PD of total effort for an mbeddr-based implementation of SMT.[12] The resulting effort of 300 PD is only slightly over this estimate, so no effort-increasing suprises came up during development and mbeddr delivered as promised: the resulting

[12] The team does have experience with estimating software project efforts.

software has a better structure and higher test coverage than what could have been achieved with plain C.

Concerning RQ-Effort we conclude:

> The effort for the additional functionality, integration and commissioning is lower than what is common in embedded software.
>
> The effort for building the extensions is low enough for it to be absorbed in a real project.
>
> Overall, using mbeddr does not lead to significant effort overrun, while resulting in better-structured software.

## 7. Discussion

In the preceding sections we have seen how C language extensions as provided by mbeddr affect the complexity, testability, overhead, and effort involved in the development of a commercial smart meter device. In this section we put our results in a broader perspective.

### 7.1 Threats to Internal Validity

From the perspective of internal validity, the key question is whether our findings are trustworthy.

**Bias** One factor that affects this question is the bias because of the involvement of the authors in this case study itself. The first and third authors are the lead creators of mbeddr, and the fourth author led the SMT project in industry with evident commercial interests. To counter this bias, we focused on aspects that can be objectively measured (size, concept counts, effort, overhead), not just for this case study, but also in other (future) projects. Furthermore, the second author has no connection to mbeddr or the companies involved in the case study, and was brought in primarily for his experience in conducting qualitative research.

**Team Expertise** To clarify the potential impact of the team on the case study outcomes, we describe the team's background and expertise: The team was led by a senior developer with 15+ years of experience in software engineering, object orientation and application development with Java and C++. He had a solid background in embedded hardware and embedded software development in C. The team had no experience in the SMT domain. When the project started, the SMT team had no signficant experience with the mbeddr languages or tools, but understood the abstractions behind the extensions (components, state machines, product lines); little education was necessary. However, the SMT team had access to the mbeddr developers for training and assistance.

**Example Smart Meter Code** Another factor potentially affecting the outcomes is the ESC. On the one hand, it served as a means for the team to understand the specifics of the SMT functionality (thereby reducing effort). On the other hand, as mentioned in Section 6.4, the team did not have experience with smart meters and the team estimates that 30% to 40% of the 200 PDs of implementation effort was due to this lack of expertise. We estimate that the net effects of the ESC on overall effort are roughly neutral.

In terms of the architecture, structure, testing or performance considerations, the mbeddr SMT implementation is new software: non-availability of the ESC would not have had a significant influence.

### 7.2 Conclusion Validity

Conclusion validity raises the question whether there is an *explanation* for our findings, which are positive overall, and favor the adoption the use of language extensions.

**Design of mbeddr** The mbeddr C extensions have been specifically *designed* to achieve the benefits reported in this case study. So the design rationale of mbeddr forms the theoretical explanation of the case study outcomes. For an extensive description of this design rationale we refer to [51].

**Cognitive Dimensions of Notations** The extensions improve C according to Green's cognitive dimensions of notations [18], a set of established language evaluation criteria. Five dimensions are specifically improved by the extensions. Incrementally adding extensions to C directly realizes the *Abstraction Gradient*: the abstraction level is increased incrementally. The user is not forced to encode everything in either a (too) low-level or a (too) high-level language. A suitable extension can be used (or developed) for each particular case. Adding domain-specific abstractions and notations increases the *Closeness of Mapping* between the code and the domain. The additional abstractions and notations are also a way of adjusting the *Diffuseness/Terseness* tradeoff of a language (or a specific program). Generally, a more terse program is better, since it exhibits lower complexity [17], assuming the language constructs used to achieve the terseness are known to all involved parties. Using the extensions reduces *Error-Proneness* because programmers can ignore low-level details irrelevant for the problem at hand. Finally, *Progressive Evaluation* is improved by IDE support and good error messages; both are emphasized in mbeddr and generally more helpful than in a regular C IDE (partly because the preprocessor makes this hard in regular C).

**Concepts vs. Language** A rival explanation of the success we measured might be that mbeddr's *concepts* (such as proper modularization) are responsible, but that the mbeddr *language extensions* are not needed. The SMT team is skeptical: a lot of discipline would be required without the extensions, and the team appreciated the seamless integration between the extensions and C itself as well as the tool support. In fact, one of the developers has developed component-oriented software in plain C before, and expresses that having these abstractions as first-class language constructs makes a huge difference in terms of productivity: "mbeddr simplifies doing it the right way. Without language and tool support, you are constantly tempted to do it the wrong way, and you are on your own not to do it wrong."

**Language vs. Tool** One may also ask whether it is the language extensions themselves or the *IDE support for the extensions* that led to the success. However, since MPS always provides IDE support for a language and its extensions, we are not able to evaluate the case where users have access to the language extensions but not to the IDE support. In addition, the extensions and the IDE support are synergistic in the sense that because of the abstractions provided by the extensions the IDE is able to provide meaningful support. Stated differently: a major reason for defining language extensions is to enable better IDE support. In this sense, a strict distinction between the language itself and the IDE support for the language is not meaningful.

### 7.3 Additional Embedded Software Challenges

When describing our case study setup (Section 3), we explained how the four aspects studied (complexity, testability, overhead, and effort) relate to our overall goal of assessing the usefulness of C language extensions for developing embedded systems. From a construct validity point of view, there are additional aspects (constructs) that we could have studied, which we briefly cover below. However, they are secondary in importance and also harder to measure than the criteria covered by the research questions; this is why we cover them here in the discussion section instead.

**Debugging** Embedded software development requires debugging to understand some of the hardware-specific behavior. Extensive testing can reduce, but not avoid this need. We distinguish extension level debugging (in mbeddr) and low-level debugging (of the generated code).

mbeddr's debugger supports step and watch on the level of the extensions [36]. While it is possible to exchange debugger backends to support on-device debugging, extension-level debugging is mostly used for hardware-independent test cases running on the PC, using a gdb backend.

Low-level debugging of the generated code uses target-platform specific debuggers (such as the one from the IAR Embedded Workbench). In SMT the latter was also necessary to find bugs in the mbeddr generators, some of which were not yet completely finished at the time (this happened on average once per month). To make low-level debugging feasible, the generated code must be readable (see below).

Both are here to stay: developers will always want to debug application logic on extension-level since extensions are built to simplify the expression of application logic. They will also want to understand the low-level representation to understand timing and resource consumption (some embedded software developers look at assembly code even today).

**Quality of the Generated Code** Quality refers to resource consumption and performance as well as readability and good coding practices. We cover overhead and performance in Section 6.3, so we focus on the other two criteria here.

Readability is essential for debugging as well as for tracking down problems in the mbeddr generators. mbeddr has always taken readability of generated code into account: names are propagated from the model, generated names are meaningful and the code is correctly indented. The code looks generally as if it were handwritten, except where the high-level extensions require the use of ugly, low-level idioms (such as pointer indirections for supporting component polymorphism, see Appendix A). We are currently investigating the use of (generated) macros to make this kind of code more readable.

mbeddr supports namespaces, and generated names are prefixed by their namespace. Since this has led to some very long names, an option has been added to mbeddr that only adds prefixes if the non-prefixed name is not globally unique. This has improved readability of the code significantly.

In embedded software, "good coding practices" is usually synonymous with compliance to MISRA-C, an industry standard that defines rules for improving the reliability of C code and avoiding errors [33]. We found through manual analysis that ca. 25% of all MISRA rules are automatically followed as a consequence of mbeddr's language design, and only a few rules are violated by the code generator (e.g., the 31 character limit for identifiers or the avoidance of function pointers). The status of the remaining rules depends on the code written by the developer in mbeddr. A future mbeddr release will ship with a checker to detect MISRA violations of code written in mbeddr.

The quality of the generated code is also important in safety-critical contexts. Standards such as ISO 61508, ISO 26262 or DO178 have strict guidelines on process, tools and code quality. Since mbeddr is not considered a qualified tool (i.e., one whose correctness has been formally proven or shown over time), the generated C code will be considered as the relevant artifact relative to the standard, requiring high code quality. A detailed discussion of tool qualification and safety-critical systems is beyond the scope of this paper.

**Maintainability** Van Deursen and Klint [49] conclude that *a DSL designed for a well-chosen domain and implemented with adequate tools may drastically reduce the costs [..] for maintaining [applications].*. We have no long-term experience on SMT maintainability, but we can make observations that confirm the conclusion by Van Deursen and Klint [49].

The implementation of SMT proceeded in two phases. First, the ESC functionality was rebuilt with mbeddr, and then additional functionalities were added. This second phase can be considered an evolution of SMT. The low effort (55 PD) illustrates the extensibility of the component-based architecture, especially considering that the system had to be distributed to run on two processors during these 55 PDs.

Recently, new sensor hardware has become available to perform the core SMT measurement. It provides better accuracy and makes SMT easier to certify. The team has decided to develop a variant of SMT that uses this new sensor. A preliminary investigation has found that support for this sensor can be provided through alternative implementations

for a small set of interfaces. The variant itself can be created by integrating (connecting) instances of the new and the existing components. This confirms the expectation that new product variants can be handled easily.

Part of the effort in software maintenance is the developers' (re-)understanding the code after a potentially long time. mbeddr's emphasis on readability (through good abstractions and notations) suggests that (re-)comprehension of the system is simplified compared to a C implementation.

Another maintenance concern is the migration of existing code as languages change in non-backward compatible ways (an active research area, as exemplified by [20]). Since mbeddr evolved significantly during SMT development, this occurred repeatedly. Until recently, no migration support has been available in MPS, requiring manual (or manually scripted) migration work; sometimes the evolution of mbeddr and SMT had to be coordinated explicitly, which was feasible because of the slack in the project timeline. Partly as a result of the experience with SMT, MPS 3.2 has added support for systematically dealing with language versions and code migration, largely solving these problems.

### 7.4 External Validity

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

**Beyond SMT** mbeddr is best suited for systems where low-level and high-level code must be mixed, where different abstractions must be used together, and where efficiency is an important (but not the only) concern. While SMT is the most significant system of this kind built with mbeddr so far, mbeddr has been used for several smaller systems (see Chapter 5 of [51]) and for a number of additional industry projects. The findings discussed in this paper apply to these projects as well, to various degrees.

Systems that are primarily control algorithms, are better developed with data-flow oriented modeling tools such as Simulink (we are currently exploring how data-flow abstractions can be added to mbeddr in the context of the controls engineering tool developed by Siemens PLM Software).

**Beyond the Team** To be successful with mbeddr, a team should have solid software engineering skills (abstraction, modularization, reuse and automated testing) in addition to proficiency in embedded software and C. The SMT developers had these skills. Unfortunately, these skills are not ubiquitous in the embedded software workforce, which traditionally emphasizes locally optimized, efficient code over (big picture) software engineering. If developers have these skills, the mbeddr-specific training is a matter of a few days based on our experience. Feedback from other organisations using mbeddr tells us that the user guide, examples and the occasional question in the forum are sufficient for learning mbeddr: no training by the mbeddr team is necessary.

**Beyond mbeddr's Extensions** The C extensions evaluated in this paper have proven useful because, as Section 6 evaluates, they solve real problems in embedded software development. Other extensions have been built in other projects that, anecdotally, exhibit similar benefits. The Missing Extensions discussed in Section 6.1 suggest there are additional extensions that could prove useful as well. Based on this experience, we conclude that language extensions are useful if (1) they address a real problem in embedded software, (2) their implementation does not introduce significant runtime overhead, and (3) they enable improved type checking, verification or IDE support compared to native C abstractions or macros. In addition, the effort for building the extensions must not be prohibitively high, but this is ensured by relying on a language workbench as the foundation.

**Beyond mbeddr's MPS Implementation** The language engineering necessary for building mbeddr and enabling the extensibility is beyond the scope of this paper (see Section 7.5 and Chapter 10 of [51]). We have chosen MPS because of its robust support for modular language extension and flexible notations. To the degree that these features are available in other language workbenches, we expect similar results when building something like mbeddr and SMT. While MPS' support for non-textual notations was important for SMT, modular language extensibility was even more critical. Modular language extension is also available in Rascal [25] and Spoofax [23], and in fact, both communities are currently implementing (subsets of) mbeddr in order to compare the language workbenches.

### 7.5 Reliability (Repeatability)

The case study reports on the development of a real-world embedded system: SMT was *not* specifically set up as a case study. This has advantages and drawbacks. The advantages include a realistic system, realistic constraints, real deadlines and experienced developers with industry-level skills. The drawback is the unavailability of the source code as well as limited availability of data. In McGrath's terms [31], this case study emphasizes realism (a real industry project) over repeatability (availability of all sources).

### 7.6 Practical Challenges and Drawbacks

We draw generally positive conclusions for mbeddr from SMT. However, there are also challenges and drawbacks when using mbeddr. Some of them are related to organizational change and introducing news ideas into organizations [39] and we do not discuss them here. Some of them are more directly related to mbeddr; we discuss those below.

**Limited Generator Optimizations** The generators that create C code from mbeddr's extensions are not as optimized as those of some established tools (such as Simulink or Stateflow). Since our existing extensions are fundamentally imperative in nature, this is not a problem. As demonstrated by SMT, the generated code runs on reasonably small hardware. In addition, because of the tight integration with C, users can always write efficient, low-level C code if in

particular places the generated code is not efficient enough (this was not necessary in SMT). For extensions that implement a different programming paradigm (such as matrix math or data flow models), optimizations will become more important, requiring more effort in generator development.

**Longer Build Times** Use of mbeddr and its extensions lengthens build times, because of the required code generation. Compared to just compiling and linking C, mbeddr's build times are typically 2-3 times longer. In SMT, we have modularized the software to ensure that incremental build of a model (as the developer writes code and executes tests) runs in less than 10 seconds. A full rebuild of SMT can take up to 4 minutes (done primarily during nightly builds).

**Tool Lock-in** mbeddr relies on MPS for editing, diff-merge and C code generation; MPS does not rely on any modeling standards (beyond a MOF-like meta meta model). While an export as generated text or on AST level (e.g., to EMF) is feasible, the benefits of the approach in terms of notation and language modularity can only be reaped when using the mbeddr/MPS tools. The drawback of tool lock-in is mitigated to some degree by the fact that mbeddr and MPS are both open source software.

**Version Control Integration** mbeddr stores programs in files (as an XML-serialized AST) which can be managed with any file-based version control system (such as subversion or git, the latter being used in SMT). However, diff-merge can only be done meaningfully in MPS, which uses the projectional editor also when showing diffs (known as a rendered diff). This means that text-based diff tools or web-based code-review tools such as gerrit cannot be used.

**Learning Curve** Using MPS' projectional editor requires some getting used to and may lead to some initial frustration. As the study in [57] shows, the required time varies between a few hours and a few days. In addition, users have to learn the extensions provided by mbeddr. This includes learning the concrete syntax, but also the concepts and the semantics; as discussed in Section 7.1, Team Expertise, the degree to which this is an issue depends on education of the developers. It was not a problem in SMT because of the team members' skills and their access to the mbeddr developers.

**Language Engineering Skills** mbeddr can be used out-of-the-box, exploiting the existing extensions. As this paper shows, these extensions provide significant benefits in themselves. However, to fully exploit mbeddr and extend it with domain-specific extensions (see the Missing Extensions discussed in Section 6.1), an organization may want to acquire language engineering skills. These skills are not naturally present in many organizations that develop embedded software, and may even be hard to hire.

**A Language Extension Ecosystem** An ecosystem requires independent third parties to be able to develop language extensions, without changing the base languages. For example, all the missing extensions identified in Section 6.1 could be built by third parties as modular language extensions. While the extensions in SMT have been developed together with the mbeddr team, non-trivial extensions have been developed without help of the mbeddr team in other projects and companies (for example, by Siemens PLM). Openly available extensions include those developed by students: parallel programming,[13] and extensions for tasks and scheduling.[14]

For an extension ecosystem to be healthy, clear quality criteria are needed. We propose the criteria expressed by the research questions in this paper as a starting point for such quality criteria. An extension ecosystem need not be global (with a central and public "extension store") to be useful. An extension library can also be maintained within an organisation or even specifically for a (large) project.

### 7.7 Research Implications

This paper provides an in depth case study of the use of mbeddr to implement a smart metering system, focusing on complexity, testability, performance, and effort. To corroborate and challenge our findings, additional studies are needed, both for mbeddr-based systems as well as for other extension-based approaches in embedded software. Furthermore, to better understand long term implications on, e.g., maintainabilility, longitudinal studies should be set up. For SMT, we will monitor and report on the continued evolution of SMT itself as part of our future work

## 8. Related Work

**Industry Studies** Model-driven engineering (MDE) uses models (as opposed to code) as the main development artifacts. In embedded software, MDE often specifically refers to dataflow models as implemented in Simulink[15] or Ascet SD[16]. An empirical study by Whittle et al. [59] finds that most commercial tools only support models at an abstraction level very close to the code. This is reinforced by Kamma et al. [22]. Whittle et al. also state that users were only successful with MDE if they customized existing tools or built their own; otherwise the complexity of the overall process was too high. Kuhn et al. [26] also performed an industry study on MDE in embedded software. Similar to Whittle's study, it expresses a need for problem-specific expressibility, i.e. the ability to define "little languages" for particular abstractions in the domain. The paper also emphasizes the lack of real abstraction. mbeddr addresses these issues by supporting different abstractions, some more code-oriented (for the low-level aspects of a system like the units in SMT) and some more abstract (where reasonable abstractions can be found, e.g., state machines). Since mbeddr relies on a language workbench, it is straightforward for users to add domain-specific extensions (such as the registers, interrupts and messages in SMT). Whittle also points out that users of-

[13] http://mbeddr.com/2014/09/29/bastianThesis.html

[14] http://mbeddr.com/2014/09/26/janoschThesis.html

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

[16] http://etas.com/en/products/ascet_software_products.php

ten prefer to not exclusively use graphical notations. This is in line with our own experiences, which is why mbeddr supports (mixing of) textual, symbolic, tabular and graphical notations [52]. Finally, Kuhn identifies model diffing and fine-grained traceability as major issues in existing tools. Both are addressed by mbeddr: it supports diff-merge for any language, and tracing works for program elements expressed in any language, at any abstraction level.

**Model-Driven Engineering** mbeddr can be seen as a flavor of MDE in the sense that developers work at higher levels of abstraction, and generation translates these abstractions to C. However, there are two important differences compared to mainstream embedded software MDE tools (such as ASCET-SD, Matlab/Simulink or Stateflow[17]). First, mbeddr is fully open: additional extensions can be added as necessary (as has been done in SMT), and extensions can be combined because the semantics of all extensions is based on C. Second, users can seamlessly integrate low-level C code with the extensions; they are not forced into the abstractions prescribed by the tool (in SMT, components contain efficient, algorithmic C code). All of this is supported in the same IDE, avoiding tool integration hassles. Existing MDE tools do not have robust support for this.

**DSLs in Embedded Software** Studies such as Broy's [6] and Liggesmeyer's [30] show that DSLs substantially increase productivity, and DSLs are increasingly used for embedded software. Examples include Feldspar [1] for digital signal processing; Hume [19] for real-time embedded systems, as well as [16], where DSLs are used to address quality of service concerns in middleware for distributed real-time systems. These DSLs generate C code, but the DSL program is not syntactically integrated with C. mbeddr supports varying degrees of syntactic integration with C. This includes external DSLs that have no syntactic integration and just generate C code; DSLs which are syntactically separate, but reference C program elements; and DSLs which are syntactically embedded into C code, and are transformed to C. This last approach is used extensively in mbeddr and SMT.

Syntax extension of C is also not new, as exemplified by Palopoli et al. [35], Boussinot [4] and Ben-Asher et al. [2]. However, these are all *specific* extensions, created by invasively changing the C grammar, and they do not include IDE support. mbeddr is different because it provides an open framework and tool for defining *modular* extensions of C, as well as the IDE. Xoc [7] supports parser-based modular extensibility, but it is limited to textual notations and also does not address IDE extension. Our SMT study shows that developers actually use this extension possibility.

**Macros** In today's practice, macro libraries such as Protothreads [10], SynchronousC [58] and PRET-C [42] are are used extensively. Macros are resolved during compilation and hence incur no performance penalty, but they have drawbacks. Syntactic flexibility is limited because macros essentially look like function calls. Also, macro calls are not type checked. Since macros operate on text, they can lead to syntactically invalid C programs and subtle errors. Finally, many IDEs cannot deal well with macros in terms of navigation in the code. Analysis tools also often have problems. We refer to [32] for a detailed discussion on the good and bad aspects of macros. mbeddr's extensions have none of these problems, since they are native, first-class language constructs. The SMT developers did not indicate that they missed C's regular macro facilities.

**C++** Based on private conversations of the authors with developers in industry (from Bosch, BMW and Harmann Becker), complex embedded software that is not targeted to very small target platforms is increasingly developed with C++. Examples include entertainment and navigation systems in cars or flight management systems in aircraft. However, overall, C++ still plays a relatively limited role in embedded software, especially in systems that target relatively small hardware like SMT. According to Stroustrup[18], this is also true for Embedded C++ [19] a restricted version of C++.

While mbeddr does not yet integrate with C++, it shares some concepts. mbeddr's components support the main use case of C++ classes in embedded programming: a clear separation between interface and implementation, and the ability to have several implementations for the same interface. However, mbeddr supports this via translation to C, so no C++ compiler is necessary for the target device. Also, all component instances are allocated at program startup to avoid running out of memory as the program executes (this is good practice in embedded software and was no limitation for SMT; C++ reliance on the heap is considered a drawback in this context). Special care has been taken to avoid performance overhead. Another reason for using C++ is template meta programming [8] for compile-time "language extension". mbeddr supports language extensions natively, providing much better IDE support and avoiding the cryptic error messages known from template meta programming.

**Language Workbenches** In Section 7.4 we argue why we have chosen MPS for the mbeddr implementation and discuss how other contemporary language workbenches, in particular Rascal [25] and Spoofax [23], differ relative to a possible realization of mbeddr's C extensions. Since language engineering is beyond the scope of this paper, we keep the related work on language workbenches brief and refer to [13] for a comparison of contemporary language workbenches.

Early examples of language workbenches include the Synthesizer Generator [38] and the Meta Environment [24]. The latter is an editor for languages defined via SDF, a general parsing framework. Rascal and Spoofax provide Eclipse-based IDE support for SDF-based languages and,

[17] http://www.mathworks.com/products/stateflow/

[18] http://www.stroustrup.com/bs_faq.html#EC++

[19] http://www.caravan.net/ec2plus/

together with Xtext[20], are recent parser-based language workbenches. In contrast to MPS, their parser-based nature restricts language syntax to essentially linear text. Language extension and extension composition is supported in Spoofax and Rascal; Xtext does not support extension composition, and hence cannot be used to build mbeddr.

Like MPS, the Intentional Domain Workbench [45] uses a projectional editor. In terms of syntactic flexibility, it has demonstrated diagrams and tables mixed with text. In terms of language extension and extension composition the available information is limited, since it is a commercial product.

Renggli et al.'s Helvetia [37] supports language embedding and extension of Smalltalk using *homogeneous* extension, which means that the host language (Smalltalk) is also used for *defining* the extensions (these kinds of extensions are also known as embedded DSLs according to Hudak [21]). The authors argue that the approach is independent of the host language. While this is true in principle, their implementation strategy heavily relies on the unique aspects of the Smalltalk system which are not available for other languages, and in particular, not for C. 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.

## 9. Conclusions

This paper presents a case study that evaluates the use of mbeddr's extensible C for embedded software development. We describe the setup and context, the challenges as well an evaluation that concludes (compared to a plain C implementation, as explained at the beginning of Section 3):

- The extensions help mastering complexity and lead to software that is more testable, easier to integrate and commission and is more evolvable.
- Despite the abstractions introduced by mbeddr, the additional overhead is very low and acceptable in practice.
- The development effort is reduced, particularly regarding evolution and commissioning.

Our experience with this case study and other projects also reveals that introducing mbeddr into an organization may be difficult, despite these benefits, due to a lack of developer skills and the need to adapt the development process.

As part of future work we will track the evolution of SMT to evaluate long-time maintainability and growth. We will also investigate other mbeddr-based projects and compare the experiences in these projects to the findings in this case study; we will evaluate the same criteria introduced in this paper. We have also already started implementing some of the possible additions to mbeddr mentioned in Section 6.1 and the performance optimizations for components in Appendix B. In particular, we are working on adding data flow extensions as part of our work with Siemens PLM Software. Finally, we are considering updating the mbeddr user guide to not just explain how mbeddr works, but also teach the essential software engineering principles to prospective users to address the skill-related adoption barrier.

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


## Acknowledgements

We thank Prof. Dr. Sami S. Al-Wakeel of the King Saud University as well as Abdelghani El-Kacimi of itemis France for their advice and the permission to write about the smart meter projet. We also thank Daniel Ratiu, Sebastian Erdweg and Iris Groher for their feedback on the paper


## References


[1] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegard, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE 2010*, 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] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3), 2004.

[4] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, (4).

[5] M. Broy. Challenges in automotive software engineering. In *Proc. of the 28th Intl. Conference on Software engineering*, ICSE '06, New York, NY, USA, 2006. ACM.

[6] 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 Models*. ICI.

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

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

[9] W. Damm, R. Achatz, K. Beetz, H. Daembkes, K. Grimm, P. Liggesmeyer, et al. Nationale Roadmap Embedded Systems. In *Cyber-Physical Systems*. Springer, 2010.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06. ACM, 2006.

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

[12] C. Ebert and C. Jones. Embedded software: facts, figures, and future. *Computer*, 42(4), april 2009.

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

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

[15] M. Fowler and M. Foemmel. Continuous integration. *ThoughtWorks, http://martinfowler.com/articles/continuousIntegration.html*, 2006.

[16] 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. *Science of Computer Programming*, 73(1), 2008.

[17] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, W. Charles, et al. Cyclomatic complexity and LOC: empirical evidence of a stable linear relationship. *J. of Software Engineering and Applications*, 2(3), 2009.

[18] T. R. Green. Cognitive dimensions of notations. *People and computers V*, 1989.

[19] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. GPCE '03, 2003.

[20] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering*. Springer, 2011.

[21] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

[22] D. Kamma and K. Sasi. Effect of model-based software development on productivity of enhancement tasks - an industrial study. In *Proc. of the 21st Asia-Pacific Software Eng. Conference (APSEC) 2014*.

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

[24] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2 (2), 1993.

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

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

[27] D. Le, E. Walkingshaw, and M. Erwig. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing*.

[28] E. Lee. What's ahead for embedded software? *Computer*, 33(9), 2000.

[29] E. Lee. Cyber-Physical Systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008.

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

[31] E. McGrath. Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human-Computer Interaction: Toward the Year 2000 (2nd ed*. Citeseer, 1995.

[32] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Berlin/Heidelberg, 2015. Springer-Verlag.

[33] MISRA. Guidelines for the use of C in critical systems, 2004.

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

[35] L. Palopoli, P. Ancilotti, and G. C. Buttazzo. A C language extension for programming real-time applications. In *6th Int. Workshop on Real-Time Computing and Applications (RTCSA 99)*. IEEE CS, 1999.

[36] D. Pavletic, A. S. Raza, M. Voelter, B. Kolb, and T. Kehrer. Extensible debuggers for extensible languages. In *GI/ACM WS on Software Reengineering*, 2013.

[37] L. Renggli, T. Girba, and O. Nierstrasz. Embedding languages without breaking tools. In *ECOOP'10*, 2010.

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

[39] L. Rising and M. L. Manns. *Fearless change: patterns for introducing new ideas*. Pearson Education, 2004.

[40] P. Romaniuk. Introduction to multithreaded programming in embedded systems. *http://elesoftrom.com.pl/en/os/ multithreaded_programming.pdf*, 2013.

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

[42] A. G. S. Andalam, P. S. Roop. Predictable multithreading of embedded applications using PRET-C. In *Proc. of ACM-IEEE Int. Conference on Formal Methods and Models for Codesign (MEMOCODE),*, 2010.

[43] O. Salo and P. Abrahamsson. Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of XP and Scrum. *Software, IET*, 2(1), 2008.

[44] Saudi Electricity Company. Specifications for electronic revenue CT and CT-VT meter. https://www.se.com.sa/ar-sa/Business_Document/Specifications

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

[46] G. L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3), 1999.

[47] J. Sztipanovits. Embedded software: Opportunities and challenges. *http://archive.darpa.mil/DARPATech2000/Presentations /ito_pdf/2SztipanovitsEmbedSWBW.pdf*, 2000.

[48] J. Sztipanovits and G. Karsai. In T. Henzinger and C. Kirsch, editors, *Embedded Software*, volume 2211 of *LNCS*. Springer, 2001.

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

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

[51] M. Voelter. *Generic Tools, Specific Languages*. PhD thesis, Delft University of Technology, 2014.

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

[53] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible C-based programming language and ide for embedded systems. In *Proceedings of SPLASH Wavefront 2012*, 2012.

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

[55] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3), 2013.

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

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

[58] R. von Hanxleden. Syncharts in C - a proposal for light-weight, deterministic concurrency. In *Proceedings of the International Conference on Embedded Sofware (EMSOFT'09)*, 2009.

[59] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Proc. of the 16th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) 2013*. ACM, 2013.

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

## A. Runtime Overhead of Components

For every component, we generate a struct that holds the data associated with each component instance. Specifically, it contains a member for each component field (`divident__field` in the example below), a member for each required port (`store__port`) and a member that is typed to another struct that holds a function pointer for each operation on a required port (`store__ops`); this latter struct is specific to the interface associated with the particular port. Note that every instance of a given component may be connected to a *different* target component, as long as it provides a port with the required interface (interface polymorphism). This is why the "wiring data" must be stored for every instance, which is why it is held in members of the instance struct (`__port` and `__ops`).

```
// struct for the 'Interpolator' component
struct Interpolator__cdata {
  // component field 'divident'
  int8 divident__field;
  // required port 'store'
  void* store__port;
  // operations for the 'TrackpointStore'
  // interface on the 'store' required port
  TrackpointStore__idata_t* store__ops;   };

// struct for the 'TrackpointStore' interface.
struct TrackpointStore__idata {
  // operation void save(Trackpoint_t* tp)
  void (*save)(Trackpoint_t*,void*);
  // operation Trackpoint* get()
  Trackpoint_t* (*get)(void*);
  // operation Trackpoint* take()
  Trackpoint_t* (*take)(void*);
  // operation bool isEmpty()
  bool (*isEmpty)(void*);        };
```

The following code is generated from a component runnable that, in the first line of the implementation, invokes the `save` method on a `store` required port:

```
void Interpolator_processor_process
     (Trackpoint* p, void* ___inst) {
  Interpolator__cdata* ___ci =
      ((Interpolator__cdata*)(___inst));
  (*___ci->portops_store->save)(p, ___ci->port_store);
  ...
}
```

Every component runnable gets an additional argument that represents the data for the current instance, `___inst`. Its type is the struct generated for the component that owns the struct (`Interpolator__cdata` in the example above). For technical reasons it is passed as a `void` pointer, and then downcast to the correct concrete type in the first line of every runnable. The second line in the code above is the actual call; the call is performed via a function pointer `save` (the name of the called operation) in the `portops` struct for the `store` required port. While the accesses to the members in the struct has no overhead because the addresses can be calculated by the compiler, the call through the function pointer is less efficient than a direct function call. In addition, the additional `___inst` argument increases the required stack size, a scarce commodity on some embedded processors.

## B. Reducing the Overhead of Components

The overhead incurred by components is especially problematic because components are used extensively in order to create modular, testable and maintainable software. In this section we discuss existing and planned ways of reducing this overhead.

**Static Wiring** If in a given executable an interface is only provided by one component, and hence no runtime polymorphism is required, the components can be connected statically, and the indirection through function pointers is not necessary. This leads to better performance, and reduces the required stack size, because no function pointers have to be stored in the generated component struct. But it also limits flexibility: no polymorphism is supported. Below is the code from Appendix A with *static wiring* enabled. The generator knows statically that the `TrackpointStore` interface is only provided by the `InMemoryStore` component. As the code below shows, its `save` operation is called directly, no indirection through a function pointer is used.

```
void Interpolator_processor_process
     (Trackpoint* p, void* ___inst) {
  Interpolator__cdata* ___ci =
      ((Interpolator__cdata*)(___inst));
  InMemoryStore_store_save(p, ___ci->port_store);
  ...
}
```

**Limiting Stack Size** We are currently working on an additional optimization which allocates the `___inst` argument as a global variable if a component is only instantiated once, avoiding the increase in required stack size.

**Inlining Runnables** Extensive use of components leads to many calls between runnables (functions in the generated C code). So, while mbeddr incurs no *additional* overhead compared to function calls when static wiring is used, it is slower than putting everything into one runnable or function because of the function calls. We are currently working on the ability to inline runnables using a combination of C's `inline` keyword and a transformation that actually inlines a runnable call on mbeddr level. As a trade-off to the reduced performance cost, the code size will increase as a consequence of code duplication.