mbeddr

Lucas Heer

Abstract

Embedded systems are widely programmed in the C programming language. While the major domain of C is efficient low-level code, it has shortcomings when it comes to safety, testing, maintainability and high-level constructs, all of them being desirable features in embedded systems development. With the recent technological advances in domain-specific modeling associated with domain-specific languages and supporting language workbenches like JetBrains MPS or Xtext, it has become easier to create new or extend existing languages. This paper gives an overview of mbeddr, an extensible language and IDE for embedded software development based on the C programming language. In addition, mbeddr will be extended with a simple language from the image manipulation domain. The workflow and the results will be compared with an implementation of the same language in Papyrus [1].

Keywords:
mbeddr, language workbench, domain specific language, embedded systems, domain specific tooling, modeling

[1] `https://eclipse.org/papyrus`

Email address: `lucas.heer@student.uantwerpen.be` ()

1. Introduction

1.1. Embedded software

Since mbeddr's main goal is to support developers of embedded software, it is first necessary to identify the main challenges and problems in the traditional embedded software development process.
an embedded system is a computer system which is embedded in a bigger technical or electrical device and serves a special purpose, such as controlling actuators or measuring sensors. As a consequence, the following challenges arise when developing software for an embedded system:

- **Safety:** A failure of an embedded system can have drastic consequences, ranging from damage to life-threatening situations.
- **Performance:** The software for embedded systems often runs on small microcontrollers with tight memory and performance constraints. Furthermore, embedded system must frequently fulfill realtime requirements. Therefore, embedded systems are traditionally developed with a low-level programming language like C, which offers direct access to the registers and memory of the underlying platform. High-level abstractions are desirable but often come with a substantial overhead, both memory- and performance-wise.
- **Maintainability:** Once an embedded system has deployed or came into the market, it is often hard to change its software. Therefore, a "get it right at the first time" development approach is important. This can be achieved through rigorous testing and formal verification of the code. Also, maintainability of the code itself can be improved through a clean design and the use of high-level constructs.
- **Time to market:** Depending on the domain, reducing the time needed for developing an embedded system may be an important goal. Especially technical devices designed for a broad audience of customers are quickly followed by either a new product or a newer generation.

Some of these goals contradict with each others. For example, a fast time to market reduces the time for testing and verification, leading to errors in the code. Safety and maintainability can be improved with high-level constructs, but these may introduce overhead which can be problematic for the performance aspect.

Embedded software is tightly associated with its domain and highly diverse, ranging from customer products like digital cameras or refrigerators to complex and distributed systems used in the automotive or aerospace. Domain-specific languages have shown to greatly contribute to the ease and productivity of the development of embedded systems (Manfred et al. (2012)). According to Ebert and Jones (2009), more than 80 percent of all companies that develop embedded systems use C as their main programming language. This seems natural since C allows for low-level access to the underlying executing platform and can be compiled to efficient binary code. On the other hand, it lacks support for high-level constructs, thus making the code hard to understand, debug and maintain. In fact, many errors arise from rather simple careless mistakes like bound errors, memory management and pointer misuse or uninitialized variables that can be easily checked for using static code analysis. See Vasik and Dudka (2011) for an overview of the most common mistakes in C source code and how static code analysis can help preventing these.
Due to C's low-level nature, it is complicated to find errors in the logic of the program itself. For example, C lacks built-in support for high-level constructs like state machines that are commonly used in embedded systems. The *mbeddr* project tries to solve these problems by changing and extending C with modern language engineering methods.

1.2. mbeddr

Language engineering offers methods to solve the aforementioned problems. The mbeddr project is an approach to address shortcomings of the C programming language with a strong focus on the embedded system domain. mbeddr is a set of integrated and extensible languages, allowing seamless integration of high-level constructs into standard C as well as custom extensions. These high-level constructs are translated to standard C code, which then is compiled with a normal compiler. The whole project is build on top of the JetBrains MPS language workbench and provides an IDE. Figure 1 shows the complete mbeddr architecture. MPS is used as a platform to implement both the C core and default extensions to the language, like state machines or physical units. On top of that, it is possible to define own user extensions. mbeddr is able to formally verify portions of its high-level constructs with external tools like NuSMV [2], a symbolic model checker used to proof certain properties of state machines. mbeddr also supports parts of the software engineering process. It allows for the textual definition of requirements and code documentation as well as adding trace links from code to requirements. Modules and their dependencies can be visualized with PlantUML [3].

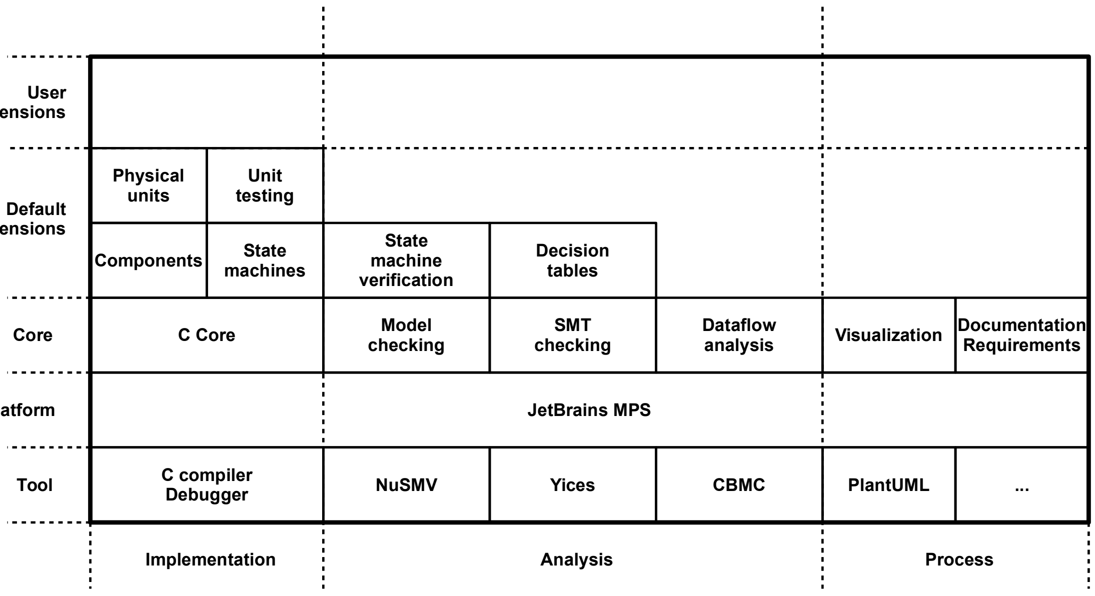


Figure 1: The mbeddr architecture stack. The underlying platform for every component is the JetBrains MPS language workbench.

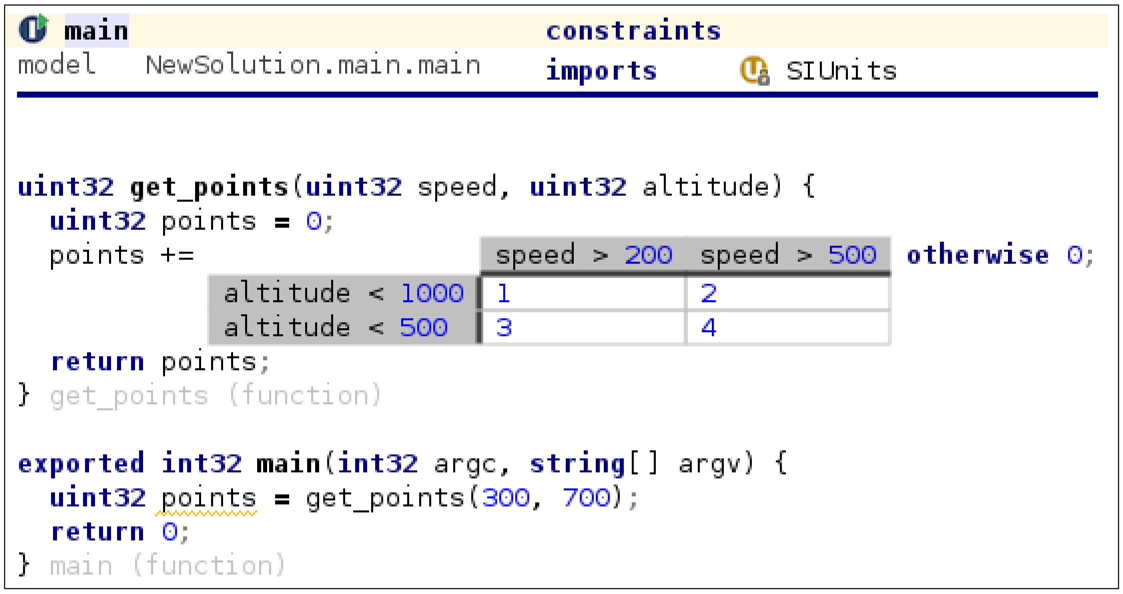
Following is an incomplete overview of changes and extensions that mbeddr offers to the programmer compared to standard C.

Cleaned up C C99 serves as a basis. In order to make C safer and more maintainable, header files and the preprocessor were removed from the language and a modern module system was added.
Decision tables mbeddr provides a graphical table embedded directly in the code that is translated to nested if-statements (see Figure 2). This leads

[2]`http://nusmv.fbk.eu`
[3]`http://plantuml.com`

to safer code, since the tabular representation helps to spot mistakes much faster than the bare textual counterpart.
State charts The language was extended with a notion of state charts. They can be defined in a textual or tabular way and visualized. State charts are translated to a switch-based implementation and can be formally verified for e.g. dead states or non-determinism, both potentially harmful and undesired properties in embedded systems.
Requirement tracing mbeddr comes with an own language to write requirements using normal text. Every requirement can be annotated with a link to the corresponding source code that implements the requirement.
Unit testing A new language extension for unit testing was designed to address the problem of testing in embedded software development.

Voelter et al. (2012) and Voelter et al. (2013) give a throughout overview of mbeddr, both from the language engineering and the embedded software development point of view.



```
main                                      constraints
model    NewSolution.main.main            imports        SIUnits

uint32 get_points(uint32 speed, uint32 altitude) {
  uint32 points = 0;
  points +=                  speed > 200  speed > 500  otherwise 0;
            altitude < 1000  1            2
            altitude < 500   3            4
  return points;
} get_points (function)

exported int32 main(int32 argc, string[] argv) {
  uint32 points = get_points(300, 700);
  return 0;
} main (function)
```

Figure 2: Concrete syntax of a decision table embedded in standard C code. See code listing 2 in the appendix section for the generated code.

1.3. Language workbenches and the mbeddr IDE

The term language workbench was first used by Fowler (2005). Accordingly, a language workbench is a tool in which it is possible to freely define new languages which are fully integrated with each other. A characteristic feature of a language workbench is that it uses a projectional editor (as opposed to a textual or parser-based editor) to manipulate a domain-specific language. Figure 3 shows the difference between these two concepts: In a textual editor, the user edits and perceives the concrete syntax in a text buffer. This buffer is then checked and transformed into the abstract syntax tree (AST). Projectional editors do not use parsers. Instead, the user directly modifies the AST while still perceiving the concrete syntax. For example, projectional editors allow the integration of graphical or tabular notations along with texutal notations. mbeddr makes use of this feature to provide a tabular representation for large if-else constructs in form of a decision table (see Figure 2). Voelter et al. (2014) give an overview over projectional editing and investigate its practical usability.

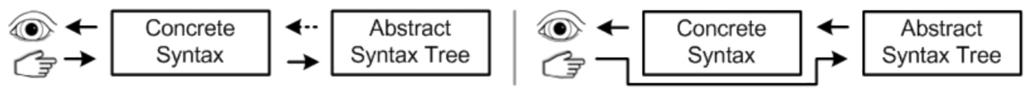


Figure 3: Difference between a textual (left) and projectional editor (right)

mbeddr is implemented using the JetBrains MPS (Meta-Programming System) language workbench. MPS is an open-source language workbench allowing the definition new languages and their IDEs while making heavy use of projectional editing. See Pech et al. (2013) for a short overview of MPS as well as an example on how to extend Java using this language workbench.

1.4. Related work

mbeddr has gained some popularity among embedded system development. In Wortmann and Beet (2016), mbeddr was used to create a domain-specific extension to the C programming language specific to the needs of satellite flight software. The extension is aware of the ECSS [4] Packet utilization standard, a standard defining the telemetry and teledata packets sent and received by a satellite. The authors identify great potential to increase

[4]European Cooperation for Space Standardization

both developer productivity and quality of the resulting software.
In order to evaluate the practical use of mbeddr, Voelter et al. (2015) have conducted an industrial case study on developing software for a smart meter. While making heavy use of mbeddr's high-level constructs, they show that it is still possible to generate efficient code with low overhead that runs on a time- and memory constraint microprocessor. They also identify a sound improvement in terms of mastering complexity and maintainability.
Vinogradov et al. (2015) shows how mbeddr can help writing code for railway domain applications. In essence, a subsystem of a legacy framework for railway applications was re-engineered. Several advantages to the traditional software writing process as well as some limitations of mbeddr were identified, among them some restrictions when it comes to copy-pasting source code into the projectional editor of mbeddr.

1.5. Overview of the paper

Section 2 presents the extension that will be implemented in the mbeddr ecosystem. Section 3 gives details about the concrete implementation within MPS. Section 3 compares both the workflow of implementing the extension and the quality of the results with a solution implemented with Eclipse Papyrus, a tool for graphical modelling of UML2 applications with extended code generation capabilities. Section 5 gives a conclusion and a short overview of possible future work.

2. Case study

2.1. Overview

As as case study, a simple image processing pipeline was chosen. It consists of a set of atomics that are images, processing blocks and links. Images can be connected via links with the processing blocks. The processing blocks are a set of pre-defined manipulations to images. Examples for these are scaling, colorspace conversion or various filters like sobel, blur or sharpen.
mbeddr will be extended with a textual language for this domain. Several checks will be implemented, for a example, it should be invalid to overlay two images with different dimensions. In essence, the language will be a simple casual block diagram (CBD) without feedback loops and hierarchy. Listing 1 shows how a model in the language could look like.

```
Image in1;
Image In2;
Image out;

BlurBlock blur(strength=1.5);
OverlayBlock ovlay();

in1.connect(blur);
in2.connect(ovlay);
blur.connect(ovlay);
ovlay.connect(out);

out.run();
```

Listing 1: Sample textual definition of a processing pipeline

3. Implementation

4. Comparison

5. Conclusion and future work

Appendix A. Code listings

Listing 2: Generated code from decision table

```
#include "main.h"

static int32_t main_get_points(float speed, float altitude);
static uint8_t main_blockexpr_get_points_6(float altitude, float speed);

static int32_t main_get_points(float speed, float altitude)
{
    int32_t points = 0;
    points += main_blockexpr_get_points_6(altitude, speed);
    return points;
}

int32_t main(int32_t argc, char *(argv[]))
{
    int32_t points = main_get_points(300, 700);
    return 0;
}

static uint8_t main_blockexpr_get_points_6(float altitude, float speed)
{
    if ( speed > 200 )
    {
        if ( altitude < 1000 )
        {
            return 1;
        }
        if ( altitude < 500 )
        {
            return 3;
        }
    }
    if ( speed > 500 )
    {
        if ( altitude < 1000 )
        {
            return 2;
        }
        if ( altitude < 500 )
        {
            return 4;
        }
    }
    return 0;
}
```

Ebert, C., Jones, C., April 2009. Embedded software: Facts, figures, and future. Computer 42 (4), 42–52.

Fowler, M., 2005. Language workbenches: The killer-app for domain specific languages? `https://web.archive.org/web/20160710201655/http://martinfowler.com/articles/languageWorkbench.html`, accessed: 2016-12-07.

Manfred, B., S. Kirstan, H. K., Schtz, B., 2012. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? IGI Global, Ch. 13.

Pech, V., Shatalin, A., Voelter, M., 2013. Jetbrains mps as a tool for extending java. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. PPPJ '13. ACM, New York, NY, USA, pp. 165–168.
URL `http://doi.acm.org/10.1145/2500828.2500846`

Vasik, O., Dudka, K., 2011. Common errors in c/c++ code and static analysis.

Vinogradov, S., Ozhigin, A., Ratiu, D., Sept 2015. Modern model-based development approach for embedded systems practical experience. In: 2015 IEEE International Symposium on Systems Engineering (ISSE). pp. 56–59.

Voelter, M., Deursen, A. v., Kolb, B., Eberle, S., Oct. 2015. Using c language extensions for developing embedded software: A case study. SIGPLAN Not. 50 (10), 655–674.
URL `http://doi.acm.org/10.1145/2858965.2814276`

Voelter, M., Ratiu, D., Kolb, B., Schaetz, B., 2013. mbeddr: instantiating a language workbench in the embedded software domain. Automated Software Engineering 20 (3), 339–390.
URL `http://dx.doi.org/10.1007/s10515-013-0120-4`

Voelter, M., Ratiu, D., Schaetz, B., Kolb, B., 2012. Mbeddr: An extensible c-based programming language and ide for embedded systems. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity. SPLASH '12. ACM, New York, NY,

USA, pp. 121–140.
URL `http://doi.acm.org/10.1145/2384716.2384767`

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

Wortmann, A., Beet, M., 2016. Domain specific languages for efficient satellite control software development. In: Data systems in aerospace.