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 implemented in the JetBrains MPS language workbench. In addition, a simple language from the image manipulation domain will be implemented in MPS to investigate its practicalness. The workflow and results will be compared with an implementation of the same language in Papyrus [1].

Keywords:
mbeddr, language workbench, JetBrains MPS, domain specific language, embedded systems, domain specific tooling, modeling, Eclipse Papyrus

[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 [2] 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

[2] `https://www.jetbrains.com/mps`

external tools like NuSMV [3], 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 [4].

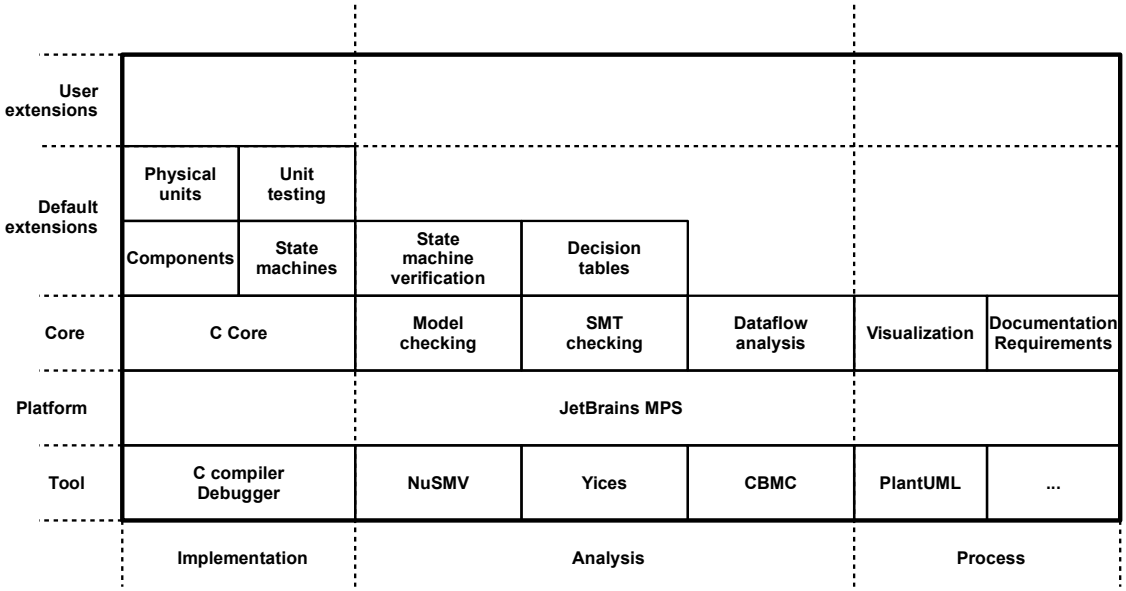

Figure 1: The mbeddr architecture stack. The underlying platform for every component is the JetBrains MPS language workbench.

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

[3] `http://nusmv.fbk.eu`
[4] `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.

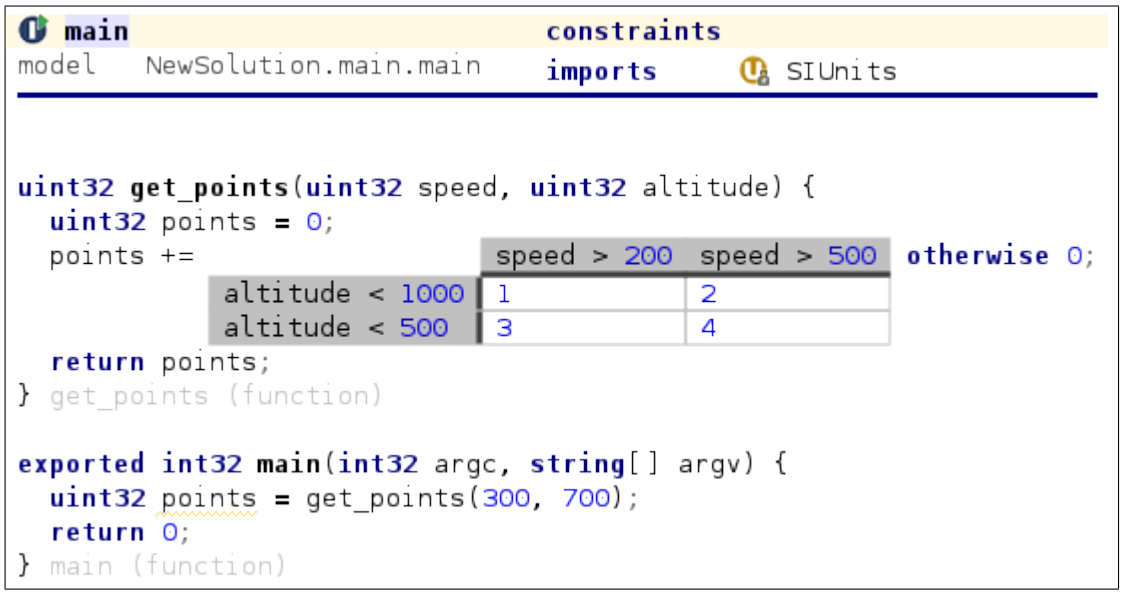

Figure 2: Concrete syntax of a decision table embedded in standard C code. See code listing 2 in the appendix section for the generated code.

1.3. Language workbenches and MPS

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 that allows for the definition of new languages while making heavy use of projectional editing. It simultaneously serves as an IDE for this language. Textual, symbolic, tabular and graphical notations are natively supported.

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. Chapter 2 shows how to implement a simple domain-specific language in MPS.

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 [5] Packet utiliza-

[5]European Cooperation for Space Standardization

tion standard, a standard defining the telemetry and teledata packets sent and received by a satellite. The authors identify great potential to increase 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 a case study to demonstrate the process of developing a language in MPS on a small scale. Section 3 elaborates on the experiences made with MPS and compares both the workflow of implementing the language and the quality of the results with the same language implemented in Eclipse Papyrus, a tool for graphical modelling of UML2 applications with extended code generation capabilities. Section 4 gives a conclusion and a short overview of possible future work.

2. Case study

2.1. Overview

The process of designing a domain-specific language in MPS will be shown and evaluated by means of a simple example. As as case study, an 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 image manipulations. Examples for such manipulations are scaling, colorspace conversion or various filters like sharpen, blur or brightness adjustment. Listing 1 shows a textual representation of a model in that language.

```
1 load input_image.jpg
2
3 grayscale
4 blur method gauss strength 2
5 resize factor 0.7
6
7 save output_image.png
```

Listing 1: Sample textual definition of a processing pipeline

2.2. Implementation

MPS uses a set of DSLs itself for defining different aspects of the language to implement. This includes the *structure, editor, constraints, data flow, type system* and the *generator*. Once the language is designed and implemented, MPS can function as an IDE for this language, offering code completion, intentions, refactoring and debugging facilities. In the following, the process of implementing the image processing pipeline language is described.

Abstract syntax

Typically, the abstract syntax of the language is implemented first. In MPS, this is called the *structure*. The structure consists of a set of *concepts*, which represent types of nodes in the AST and define properties, children and references. As such, concepts can be seen as classes in the object-oriented programming world.

In the image processing language, every command has a concept. Figure 4 shows such a concept definition for the "blur" command. It extends the

"Command" concept, which is an abstract concept that is extended by every command. The "properties"-field serves as a specification for the arguments of the command, which in this case are the blurring method and the strength of the effect.

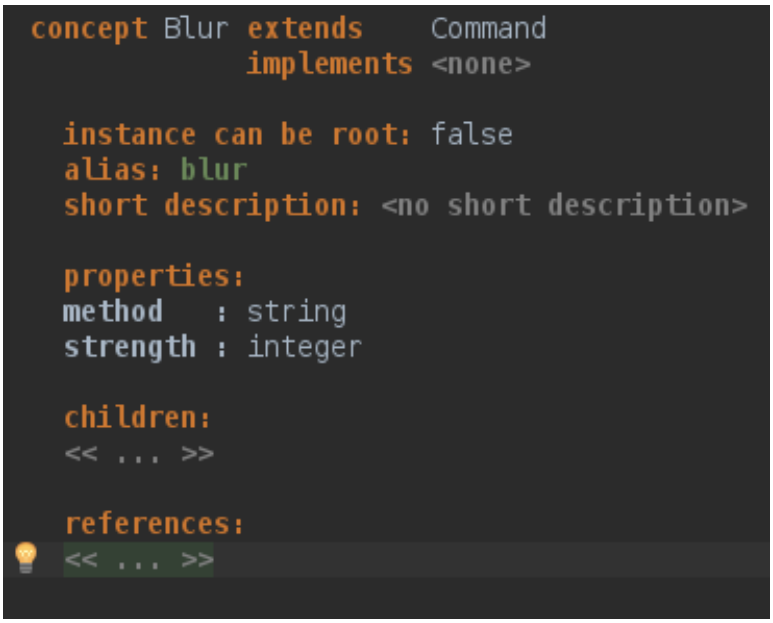

Figure 4: Concept definition for the blur command.

Beside the definition of a concept for every command, the pipeline itself has to be defined. Figure 5 shows the abstract syntax for the pipeline. The most important part is the "children"-field: Every pipeline starts with exactly one load command, followed by a list of commands and ends with a save command. Note that the concepts for "load" and "save" do not implement the abstract "Command" concept so they are not included in the list of commands.

Concrete syntax

Once the abstract syntax is defined, the next step is the design of the visual representation of every concept (more specific: a representation of every AST node). This is done by adding one or more editors to every concept. A concept can have multiple editors to let the programmer choose between different notations that best fits the task. For example, mbeddr

```
concept Pipeline extends    BaseConcept
                implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  load     : LoadImage[1]
  commands : Command[0..n]
  save     : SaveImage[1]

  references:
  << ... >>
```

Figure 5: Concept for the complete pipeline. The children define which concepts are valid in the pipeline.

makes use of this feature by defining different representations for the state chart construct (textual, tabular and graphical, as shown in section 1).

The language for defining editors is rather straight-forward. For the image processing language, only textual representations were implemented. The font and the syntax highlighting is also defined in the editor. Note that it is not strictly necessary to define editors; MPS comes with a default textual representation for every concept that will be used when no editor is defined.

Enhancing editing experience

Although the language is now completely defined, it is possible to assist the programmer with modern IDE features and further constrain the language. One constraint is already implicitly given by the structural definition of the pipeline (see Figure 5): Every pipeline begins with a load command, followed optionally by some processing commands and ends with a save command. Further constraints mainly concern valid values for the properties. For example, it is currently possible to give the "method"-property arbitrary strings whereas only "gaussian" and "box" should be valid. This can be

defined in the *constraints*-aspect of MPS. The image processing language makes only use of the property constraints, which are Java functions that have access to the property and evaluate to a boolean. In the same manner, constraints for the numeric values of most image effects are implemented.

The functions that define the constraints are invoked each time when evaluating the allowed position for a node in the AST. This means that an invalid argument is immediately shown in the IDE editor of the language.

Generator

Typically, a domain-specific language is not executed but transformed into another language. The generation process in MPS consists of two phases: First, a template-based model-to-model transformation engine reduces the model into a model of the target language. The second phase uses text generators to convert the reduced model into regular program text. Two main parts are needed to define the generator: A *mapping configuration* which specifies which concepts are processed with which templates and the templates itself. Templates are written in the target language and enriched with macros that tie the template to the input concept. Figure 6 shows the template for the load command. Every load command gets reduced to fragment of Java code that uses the ImageIO class to load an image file. The file name argument is annotated with a macro that copies the name property from the load image concept.

For the case study, Java was chosen as target language. MPS itself is implemented in Java and every language definition concept shown above builds around a custom dialect of Java, called the *BaseLanguage*. Although MPS supports the transformation to arbitrary target languages, these first have to be modeled in MPS itself. For example, mbeddr has implemented the complete C language in MPS and then build the extensions atop of it.

2.3. Code example

Figure 7 shows an example pipeline that was written in MPS after the language has been defined. After applying the code generator, the code shown in listing 3 in the appendix is produced.

```
template reduce_LoadImage
input    LoadImage

parameters
<< ... >>

content node:
{
  BufferedImage image = null;
  <TF [{
      try {
        image = ImageIO.read(new File("${filename}"));
      } catch (IOException e) {
        e.printStackTrace();
      }
    }] TF>
}
```

Figure 6: Generator template for the load image command.

```
Pipeline MyPipeline
  load image.png
  adjust brightness value 1.2
  grayscale
  blur method gauss strength 2
  save image.jpg
```

Figure 7: An example pipeline.

3. Discussion and comparison

3.1. Experience with MPS

Outside the research domain, language workbenches have the reputation to be complicated and not mature enough to be used in production (Erdweg et al. (2015)). One main concern in the past was the projectional editing concept which is deeply integrated into MPS. Programmers are accustomed to edit code freely as text and not directly manipulate the abstract syntax tree of the language. In the recent years however, projecitonal editing has become more user friendly and is nowadays considered just a minor obstacle (See Voelter et al. (2014) for an overview of projectional editing and Berger et al. (2016) for an empirical study about the editing efficiency). This was confirmed while implementing the image processing language. While the behavior of the editor was unfamiliar in the beginning, it was found to be on par with textual editing where the concrete syntax is changed directly.

MPS itself has proven to be a stable and mature environment for developing domain-specific languages. It is extensively documented with tutorials and screencasts that gradually guide the user through the different aspects of MPS. However, some screencasts were recorded for older versions of MPS and therefore slightly outdated. Also, although quick successes were possible in the beginning, more advanced topics like data flow or the type system aspect are often neglected in tutorials or only poorly described.

Initially, it was planned to transform the DSL to C/C++ code (following the mbeddr project on a small scale). After some initial research it became clear that this was impracticable and beyond the scope of this paper, as it would have required to first implement the target language in MPS. Since this was already done by the mbeddr project for generating C code, copying the language definition was considered but discarded because the mbeddr core language is tightly integrated in the whole framework. However, apart from mbeddr, MPS was successfully used for generating code for different target languages, among them C++ and R [6] [7].

3.2. Comparison with other tools

Beside MPS, numerous other language workbenches exist. They can be informally divided into textual, graphical and projectional. Textual work-

[6] `http://campagnelab.org/software/metaR/`

[7] http://aveco.com/

benches include Xtext [8] or Spoofax [9]. An example for a graphical workbench is MetaEdit+ [10]. The most recent development are projectional workbenches like MPS. For a comprehensive comparison between different language workbenches, refer to Erdweg et al. (2015). Here, a comparison with Eclipse Papyrus, a UML2 tool for domain-specific language modeling is done. The DSL described in section 2 was also implemented in Papyrus. The following metrics were compared:

- Tool complexity: How easy is it to install, learn and use the tool?
- Tool scope: What is the target audience of the tool? What are its capabilities?
- Usability of DSL: How usable is the DSL? How good does it reflect the domain?
- Extensibility: How can the DSL be extended with additional constructs? What steps are involved?

In the following section, theses questions will be subsequently answered.

Tool complexity

JetBrains MPS is implemented in Java and therefore available for every system that runs the JVM. There exists a official documentation as well as a set of screencasts and tutorials to guide beginners through the process of developing a language. Once initial struggles with the projectional editor are hurdled, the tool is just as easy to use as every other JetBrains product.

Papyrus is build around the Eclipse platform and therefore readily available. An official documentation also exists but is rather limited to UML modeling and not the creation of custom DSLs. Much information around this topic has to be gathered from forums and third parties. Compared to MPS, the learning curve was perceived as more steep in the beginning.

Tool scope

[8] `http://www.eclipse.org/Xtext/`
[9] `http://www.metaborg.org/en/latest/`
[10] `http://www.metacase.com/`

With MPS, it is possible to create DSL for every domain. Although the main focus is on textual representation of the language, MPS also allows for tabular and graphical editing. The mbeddr project is a good example for that. MPS is closely tied to Java and uses a Java dialect to define the DSL constructs. As a result, only Java is natively supported as a target language for code generation. Generating code for different target languages is possible but involves much more effort since this language itself has to be implemented first.

Papyrus itself is not explicitly a tool for designing DSLs, however it supports extension mechanisms that make it possible to add new concepts to existing UML constructs. In order to define a DSL, a UML profile has to be created which defines the abstract and concrete syntax of the language. Like MPS, the generator is template-based: fragments of the code are written by hand and annotated with macros to copy information from the source model.

Usability of DSL

In most cases, the concrete syntax of DSLs in MPS is textual, although graphical elements are also supported. However, these seem to be more complex to implement and are therefore only sparsely described. Figure 7 shows a model in the DSL implemented in MPS. Several constraints were added to the DSL. First, each model starts with a load and ends with a save command. Also, parameters to several functions were restricted to valid values only. Most commands support auto complete. Finally, MPS simultaneously serves as an IDE for the DSL which makes distribution to end users easy.

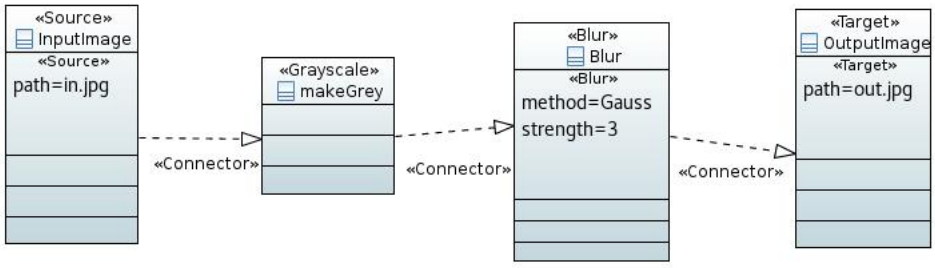

Figure 8: An model of the image processing DSL in Eclipse Papyrus.

The DSL in Papyrus is represented in a graphical way. See figure 8 for an example. The implemented constraints were similar to the ones in MPS.

However, in order to make the DSL user friendly, a plugin has to be developed first which then can be installed in Eclipse.

Extensibility

Adding a new command to the DSL is straight-forward in MPS: A new concept has to be added, an editor defined and a new template for the generator written.
In Papyrus, the process of extending the DSL is similar: First, a new stereotype which extends an UML element has to be defined. Then, the template for code generation has to be edited so that code is generated for the new element.

In summary, it was feasible to realize the DSL in a reasonable amount of time with both tools. While MPS emphasizes projectional editing as well as textual representations for the concrete syntax, Papyrus, due to its proximity to the UML, is well suited for languages that are expressible with object-oriented constructs.

4. Conclusion and future work

With mbeddr, a concrete example of a large-scale domain-specific language has been presented. mbeddr has been found a powerful solution for solving the inherent problems that occur when developing software for embedded systems. The success of the mbeddr project suggests that language workbenches are mature enough be used in production. To further investigate the practicability of language workbenches in real-world applications, a language from the image manipulation domain has been implemented in JetBrains MPS. That involved defining the abstract syntax, the concrete syntax and a generator to transform models in the language to executable Java code. The process as well as the results have been compared to an implementation of the same language in the Eclipse Papyrus project.

Future research can be done on the mbeddr project itself or on MPS as a language workbench. Although both have been proven to be valuable additions to their specific domain, industrial adoption seems to happen only slowly. Another aspect concerns the extensibility of mbeddr: Despite being mentioned in various sources, little can be found on concrete process of extending mbeddr with a custom language.

Appendix A. Code listings

Listing 2: Generated code from decision table

```
1  #include "main.h"
2
3  static int32_t main_get_points(float speed, float altitude);
4  static uint8_t main_blockexpr_get_points_6(float altitude, float speed);
5
6  static int32_t main_get_points(float speed, float altitude)
7  {
8      int32_t points = 0;
9      points += main_blockexpr_get_points_6(altitude, speed);
10     return points;
11 }
12
13 int32_t main(int32_t argc, char *(argv[]))
14 {
15     int32_t points = main_get_points(300, 700);
16     return 0;
17 }
18
19 static uint8_t main_blockexpr_get_points_6(float altitude, float speed)
20 {
21     if ( speed > 200 )
22     {
23         if ( altitude < 1000 )
24         {
25             return 1;
26         }
27         if ( altitude < 500 )
28         {
29             return 3;
30         }
31     }
32     if ( speed > 500 )
33     {
34         if ( altitude < 1000 )
35         {
```

```
36                  return 2;
37              }
38              if ( altitude < 500 )
39              {
40                  return 4;
41              }
42          }
43          return 0;
44      }
```

Listing 3: Generated code from the image pipeline

```
1   package ImageProc.sandbox;
2
3   /*Generated by MPS */
4
5   import java.awt.image.BufferedImage;
6   import java.io.IOException;
7   import javax.imageio.ImageIO;
8   import java.io.File;
9   import com.jhlabs.image.AbstractBufferedImageOp;
10  import com.jhlabs.image.ContrastFilter;
11  import com.jhlabs.image.GrayscaleFilter;
12  import com.jhlabs.image.GaussianFilter;
13
14  public class MyPipeline {
15
16    private static void saveImage(BufferedImage image, String filename)
17    throws IOException {
18      String fileExt = "";
19      int i = filename.lastIndexOf('.');
20      if (i > 0) {
21        fileExt = filename.substring(i + 1);
22      }
23      ImageIO.write(image, fileExt, new File(filename));
24    }
25
26    public static void main(String[] args) throws IOException {
27      BufferedImage image = null;
```

```
28      AbstractBufferedImageOp filter = null;
29
30      try {
31        image = ImageIO.read(new File("image.png"));
32      } catch (IOException e) {
33        e.printStackTrace();
34      }
35
36      filter = new ContrastFilter();
37      filter.setBrightness(1.2f);
38      image = filter.filter(image, null);
39      filter = new GrayscaleFilter();
40      image = filter.filter(image, null);
41      filter = new GaussianFilter(2);
42      image = filter.filter(image, null);
43
44      try {
45        saveImage(image, "image.jpg");
46      } catch (IOException e) {
47        e.printStackTrace();
48      }
49    }
50  }
```

Berger, T., Völter, M., Jensen, H. P., Dangprasert, T., Siegmund, J., 2016. Efficiency of projectional editing: A controlled experiment. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016. ACM, New York, NY, USA, pp. 763–774.
URL http://doi.acm.org/10.1145/2950290.2950315

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

Erdweg, S., van der Storm, T., Voelter, M., Tratt, L., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J., 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Computer Languages, Systems and Structures 44, Part A, 24 – 47, special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and {SLE} 2014).
URL //www.sciencedirect.com/science/article/pii/S1477842415000573

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.