FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

C++ Unterstützung für MBEDDR

Author:	Zaur Molotnikov
Supervisor:	Dr. Bernhard Schätz
Advisor:	Dr. Daniel Ratiu
Date:	September 16, 2013

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 16. September 2013 Zaur Molotnikov

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

Contents

1 Introduction

In embedded programming the C++ programming language is widely spread, [5]. Being a general purpose programming language, C++ does not provide, however, any special support for an embedded systems programmer.

By changing the language itself, together with a tool set for it, it is possible to get a better environment for a dedicated domain, for example, specifically for embedded programming.

The first possible approach is dropping some language features, to get the language, which is simpler. As an example, a subset of C++, called *Embedded C++* can be brought, [2]. The approach taken in Embedded C++ is omitting very many core features of C++ off, allows for a higher degree of optimizations by compiler possible. Embedded C++ was intended to allow higher software quality through better understanding of the limited C++ by programmers, higher quality of compilers, better suitability for the embedded domain, [1]. This approach, however, has been criticized by the C++ community, specifically for the inability of the limited language to take advantage of the C++ standard library, which requires the C++ language features, absent in Embedded C++, [6].

The second approach to modify a language to get it more suitable for the embedded development, consists of extending the language with constructions specific to the domain. Such approach is taken, for example, in the *MBEDDR Project*, to improve on the C programming language, [7]. Extensions to C language developed in *MBEDDR Project* include state machines and decision tables.

A special language engineering environment is used to support modular and incremental language development in *MBEDDR Project*, *JetBrains MPS*. The language under development is split in a special class-like items, called concepts. As an example of a concept expression can be taken. Over the inheritance mechanisms, it is possible to extend languages, providing new concepts as children of the existing ones. For example, expression concept can be extended to support new sort of expressions, e.g. decision tables.

Building a general purpose programming language in a language engineering environment brings a basis to develop domain specific extensions to a well-known general purpose language.

Additionally to the language modification, the *Integrated Development Environment (IDE)* can be improved to support domain specific development. Various analyses can be built in into the code editor in order to detect inconsistencies, or, simply, “dangerous” constructs, and inform the programmer. Certain code formatting, or standard requirements could be enforced as well, and many more.

A mixture of the two approaches could be used in an attempt to achieve a “better” C++ for embedded development. A special *IDE* can be created together with a new C++ language flavor, which supports the embedded C++ programmer. This is the problem to be solved by this Master Thesis.

The new language together with the new *IDE* can later serve as a basis for extending the

C++ programming language with domain specific constructs for embedded programming. Creation of these extensions lies out of scope for this Master Thesis, and is left for further research.

The approach taken in this work goes further into exploring the language modularity on the basis of *JetBrains MPS*Ẇhile building the C++ programming language itself with the goal of embedded domain specific extensions in mind, the C++ itself is being built itself as an extension to the C programming language, provided by the *MBEDDR Project*.

Although C++ is a separate from C language, the high degree of similarity allows to make use of the C programming language, implemented by the *MBEDDR Project* as a foundation. Not only reuse of the basic C is achieved, but also the embedded extensions from the *MBEDDR Project* are immediately supported by the newly built C++.

This work explores further the support, provided by *JetBrains MPS* for the modular language construction, [3], and reviews it from the architectural point of view.

The C++ programming language is provided with a number of automations and analyses for it. The automations include code generation and structuring. They are implemented as a programming on the *Abstract Syntax Tree (AST)* in a Java-like programming language. The analyses and automations intend to improve quality, security and better understanding of the code.

In the *JetBrains MPS Application Programming Interface (API)* it is not explicitly defined, when the analyses and checks take place, how much of the computational resource they can take advantage of. This may affect the overall *IDE* performance, as the analyses complexity may be high. The question of analyses run-time and complexity is raised and discussed in this work.

2 Foundations

Before describing the technologies on which the current work is based, as well as the work itself, it makes sense to describe more general foundations and principles, around which the technology is built.

In the Section 2.1 I describe two approaches to create an *IDE* for a certain language, and mainly the projectional approach, which originates from the area of building new *Domain Specific Languages (DSLs)*.

In the Section 2.2 I describe the modular approach toward alnguage engineering and extending, intensively used with the projectional approach to construct languages.

2.1 Building DSLs and IDEs

This section compares the traditional approach to build textual editors for the program code with the projectional approach, bringing up motivation for the least.

2.1.1 Traditional Approach

Traditionally programming languages are used in a textual form in text files, forming programs. However the textual nature is not typical for the structure of programs themselves, being rather a low-level code representation, especially when talking abut syntax, which is only necessary for parsers to produce correct results, and not for the program intended semantics.

Parsers are used to construct so-called *AST*s from the textual program representation. *AST*s are structures in memory, usually graph-alike, reminding a control flow graph, where nodes are different statements and edges are the ways control passes from one statement to the next one.

For the developer, using an editor, the degree to which the editor can support the development process is important. For this, the editor has to recognize the programming language constructions and provide possible assistance. Among such assistance can be code formatting, syntax validation, source code transformations (including refactoring support), code analyses and verification, source code generation and others. Many of these operation rely indeed on the higher than text level notions related to program such as a method, a variable, a statement. A good editor has to be aware of these higher level program structures, to provide meaningful automations for the operations mentioned above.

Nowadays, most of the editors work with text, and, to provide assistance to a programmer, integrate with a parser/compiler front-end for the programming language. Such way to extract the program structure during editing is not perfect for several reasons.

First of all, the program being edited as text is not syntactically correct at every moment, being incomplete, for example. Under such circumstances the parsing front-end can not be successfully invoked and returns error messages which are either not related to the program, when the code is completed, or false-positive warning and errors.

Secondly, after a minor editing of the code, usually the whole text file has to be processed again. Such compiler calls are usually computationally expensive, they slow down, sometimes significantly, the performance of the developer machine. Various techniques exist to speed it up, including partial and pre- compilation, but the problem is still relevant to a large extent.

Moreover, the textual nature of the code complicates certain operations additionally. As an example, we can take a refactoring to rename a method. Every usage of the method, being renamed, has to be found and changed. To implement it correctly an editor must take into account various possible name collisions, as well as presume a compilable state of the program prior to the start of the refactoring.

Not to mention the parsing problem itself. Parsing a program in a complex language like C++ is a difficult problem, it involves the need to resolve correctly scoping and typing, templates and related issues, work with pre-processor directives incorporated in the code. In this regard different compilers treat C++ in a different way, creating dialects, which may represent obstacles for the code to be purely cross-platform.

Listing 2.1: Closing several blocks

```
class MyClass {
  void doSomething() {
    while(true) {
      try {
        // ...
        }
      catch(Exception e) {
      }
    }
  }
};
```

The textual representation of program code, involves the need in formatting and preserving syntax. These both tasks, indeed, have nothing to do with the functionality program, and additionally load the developer, reducing productivity. As an example, here I can mention the need to close several blocks ending at the same point correctly, indenting the closing brace symmetrically to opening one. The Listing 2.1 demonstrates it in the last few lines.

2.1.2 Projectional Approach

Another approach which can be taken in the organization of an editor for a programming language is called *projectional approach*. Projectional editors do not work with a low-level textual representation of a program, but rather with a higher level concept, *AST*s. This approach is especially useful and used when constructing new *DSL*s.

Working with *AST*s directly has several advantages over the conventional textual code editing.

Firstly, all syntax errors are no longer possible, as there is no syntax.

Secondly, there is no need to format the code on the level of indentation and look, since it is only needed for textual code.

Thirdly, all features, which in textual approach require parsing, can be implemented without a parser involved, because *AST* is always known to the editor.

Additionally, as the compilers still expect a code in a textual form, code generation is used to convert the *AST* into the text code for the further use. The code generation step can be customized to provide support for a variety of compilers, when the compilers differ.

Projectional editors have to display the *AST* to the developer, in order for him/her to work with it. Such visualization of an *AST* is called "projection", giving a name to the editor class.

The model of code is stored as an *AST* in the projectional editor. As in the Model-View-Controller pattern the view for the model can be implemented separately, [4]. Thus the code may be presented in a number of different ways to the user. For example, the *AST* can be visualized as a graph, similar to control flow graph. This visualization, however, is not always advantageous being sometimes not compact and complicated to overview.

```
MyClass::MyClass(int8 x):  from MyClass {
  if (x > 0) {
    while ( true ) {
      --x;
      if (x == 0) {
        return;
      } if
    } while
  } if
}
```

Figure 2.1: Example projection of an *AST*, "source code" view

modularlang One of the well-accepted way to visualize *AST* is by visualizing its textual representation, as if it would be written as a text code in the programming language, see Figure 2.1. There can be in principle many such textual visualizations, supporting different ways the code looks. Normally in the traditional approach this has to be achieved by reformatting, and thus changing, the source code. This is performed for the code to look similar across the developed software, and standards or coding guidelines are written to enforce the way to format the text code. Compare to the projectional approach, where such formatting guidelines are not needed, when arguing about the low-level code formatting, like indentation.

The textual projection of the *AST* looks similar to the text code. However the projectional nature of it has certain outcomes, which may be unusual for a programmer, who is used to editing the code as text.

The statements in the projectional editor are only selected as whole. There is now way to just select the "while" word for cut or copy, without selecting the condition and the block belonging to the statement. This behavior represents the position of the condition and `while`-body in the *AST* as children of the `while` statement. The statement can be selected all together only, including all of its children. Alternatively, one could select just an expression in the condition part.

Every block delimiters are just a part of the block visualization. They are organized in a proper way automatically, and there is no way to delete or confuse them, as well as to type them initially. Each closing brace can marked with the parent statement name (through implementing such behavior in the *AST* visualization), enhancing navigation through the displayed code.

As one can see, the textual projection of the *AST* looks almost the same, as a text code in a conventional textual editor. This can cause some confusion for the developer at first, as attempts to edit this textual visualization as a real text will sometimes fail.

Eventually, however, advantages of such visualization overwhelm the disadvantages. Among the benefits of the textual projection over text code are quicker code construction after short learning, better way to select code fragments, since not individual characters or lines, but rather *AST* nodes or groups of nodes are selected, plus, all the advantages, the projectional editing brings by itself, as discussed above.

I discuss additionally the projectional approach and some of its basic principles in the Part **??**.

2.1.3 Describing a Language in Projection

When building a projectional editor for a language, the language must be given as a certain description of the *AST*. As *AST* represent a graph, the nodes and edges types, as well as their possible relationships must be described[1].

2.2 Modular Language Engineering

[1]Compare this with the textual approach, where a grammar for the language must be built, which is generally speaking complex, and some times even not possible task

3 Technologies in Use

The C++ programming language developed through out this Master Thesis is based on two technologies, which are introduced in this chapter. The first technology is the *JetBrains MPS* language engineering environment, which provides the core foundations and means for incremental language construction. The second technology is the *MBEDDR Project*.

3.1 Jetbains MPS

3.2 MBEDDR Project

4 Projectional C++ Implementation

5 Evaluation

6 Conclusion

Appendix

Glossary

API Application Programming Interface. 2

AST Abstract Syntax Tree. 2–6

concept is a class-like item, representing a modularity base of a language, e.g. expression, statement, method, etc.. 1

DSL Domain Specific Language. 3, 4

Embedded C++ is a language subset of the C++ programming language, intended to support embedded software development. 1

IDE Integrated Development Environment. 1–3

JetBrains MPS is a language engineering environment allowing to construct incrementally defined domain specific languages. 1, 2, 7

MBEDDR Project is a JetBrains MPS based language workbench, representing C language and domain specific extensions for the embedded software development. 1, 2, 7

projectional approach is an approach to create an editor for a language, when the editor is aware of the AST for the code, and shows the code to the user, projecting the AST itself, and allowing to edit the AST directly. 4

Bibliography

[1] Embedded C++. Objectives behind limitting c++, http://www.caravan.net/ec2plus/objectives/ppt/ec2ppt03.html.

[2] Embedded C++. Official website, http://www.caravan.net/ec2plus/.

[3] Zaur Molotnikov Daniel Ratiu, Markus Voelter and Bernhard Schaetz. Implementing modular domain specific languages and analyses. In *Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation*, 2012.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[5] VDC Research. Survey on embedded programming languages, http://blog.vdcresearch.com/embedded_sw/2010/09/what-languages-do-you-use-to-develop-software.html.

[6] Bjarne Stroustrup. Quote on embedded c++, http://www.stroustrup.com/bs_faq.html#ec++.

[7] Markus Voelter, Daniel Ratiu, Bernhard Schätz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *SPLASH*, pages 121–140, 2012.