FAKULTÄT FÜR INFORMATIK
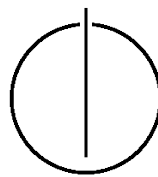
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov

FAKULTÄT FÜR INFORMATIK
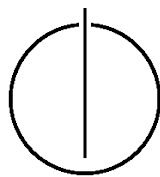
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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, [6]. Being a general purpose programming language, C++ does not provide, however, any special support for embedded systems programmers.

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. There are two known approaches to change the language itself.

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++ like virtual base classes, exceptions, namespaces and templates. It allows for a higher degree of optimizations by compiler possible.

Embedded C++ was intended to ensure higher software quality through better understanding of the limited C++ by programmers, higher quality of compilers, through simplicity, better suitability for the embedded domain, through memory consumption considerations [1].

The approach taken in *Embedded C+++*, however, has been criticized by the C++ community, specifically for the inability of the limited language to take advantage of the C++ *Standard Template Library (STL)*, which requires the C++ language features, absent in *Embedded C++*, [7]. As a response for it *Extended Embedded C++* has appeared, which includes many of the omitted by *Embedded C++* language features and a memory-aware version of *STL*, [8].

The second approach to modify a language in order to get it more suitable for embedded development consists of extending the language with constructions specific to the domain. Such approach is taken, for example, in the *mbeddr*, to improve on the C programming language, [13].

Specific extensions may represent some often met idioms in the domain, for example, behavior of a device under control is often described by a state table or a state machine diagram in the device specification. Such notions can be incorporated into language, providing a higher abstraction level, compared to the core language constructs.

Moreover, higher level extensions induce some higher level semantics, which could be checked for correctness on the programming stage. For example, a given decision table could be checked for completeness of choices and their consistency, [3]. Such checks can improve quality of the software under development.

Extensions to C language developed in *mbeddr* include state machines and decision tables, together with analyses for them.

A special language engineering environment, *JetBrains MPS*, is used to support modular and incremental language development in *mbeddr*. The language under development is split in a special class-like items, called *concepts*, representing the *Abstract Syntax Tree (AST)* node types. As an example of a *concept* an expression can be taken. It is possible to describe

in *JetBrains MPS* different expression kinds, similar to object oriented class hierarchy, allowing the objects to reference each other, and enabling polymorphism, in a way when any descendant can be used instead of its ancestor, e.g. binary minus expression can be used wherever an expression (any expression) is required. After various expression types were described to the language engineering environment as *concepts*, the environment provides a chance to instantiate concrete expressions and edit them, acting as an editor for the language created.

Over the inheritance mechanisms, it is possible to extend languages, providing new concepts as descendants of the existing ones. For example, expression concept can be extended to support new sort of expressions, like decision tables. Thus language modularity is achieved and incremental development is made to be possible.

Modularity is achieved as well, when one language is enabled to interact with another one. For example, expressions, described independently as a language, can be reused in any language, which has a need in expressions, like language with statements of a programming language, because statements include expressions naturally.

Having in mind the opportunities, the language modularity in *JetBrains MPS* brings, it makes sense to recreate a general purpose programming language in *JetBrains MPS*. Building the general purpose programming language brings a basis to develop domain specific extensions to the well-known general purpose language. The editor for the general purpose language comes almost "for free", as a side product.

Later, from the code in the implemented general purpose language a text code can be generated for further processing, compiling, deployment. The language extensions, ofcourse, are not known to the existing tools which process the language. But they usually can be reduced to the base general purpose programming language statements, presenting a regular syntax to the further tool chains as an outcome. Thus the general purpose programming language is getting enhanced, remaining compatible with all the existing tools to process it further.

Additionally to the language modification itself, an *Integrated Development Environment (IDE)* can be improved to support the domain specific development.

Various analyses[1] 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. The *IDE* can be enhanced with various automations, like support for code generation and refactorings.

As the new *IDE* works internally with *AST* described through the node types, or *concepts*, in order to perform code analysis, generation, or transformation, there is no need to invoke parsers for the code, which is advantageous.

A mixture of the two approaches is used in this work in an attempt to achieve a modular C++ language, a suitable base for the further language engineering, including the specialization of C++ for embedded development, and even more general, for later extension, to specialize the common base C++ language to any domain of choice. A special *IDE* is created together with a new C++ language flavor, which supports the C++ programmer.

During the creation of the C++ programming language in the way described, the language modularity in general is analyzed, and caveats of it are described together with the ways to avoid them.

---

[1]analyses not only for extensions, but for the base language itself

The newly created *IDE* features analyses. The question of their computational complexity for such analyses is raised in general, together with the practical outcomes of it.

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*. While 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*.

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* as a foundation. Not only reuse of the basic C is achieved, but also the embedded extensions from the *mbeddr* are immediately supported by the newly built C++.

The ultimate goal during the reuse of *mbeddr* as a base for C++ is keeping *mbeddr* not modified towards the C++ programming language only, but instead, making, when needed, *mbeddr* more extensible in general, so that both resulting C++ and the base for it, *mbeddr*, can develop further being disjoint to a high degree. This independence of *mbeddr* on C++ extension ensures, that the *mbeddr project* can develop further without looking back on C++, making the C++ support an independent task.

The task of such a one-side-aware only extension is a challenge for the whole language modularity concept, provided by *JetBrains MPS* and used in *mbeddr*. 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 designed to compensate on some caveats of C++, or lack of support for some aspects, like coding style. The analyses are intended to increase the understanding of the constructed code by a novice to C++ programmer, or to provide a quick information to an experienced C++ professional. Both analyses and automations are provided to achieve an improvement in quality, security and understanding of the pure C++ code. The automations and analyses are mostly implemented as a programming on the *AST* in a Java-like programming language.

As analyses and automations grow in complexity and quantity, the question of their computational complexity arises. In the *JetBrains MPS Application Programming Interface (API)* it is not explicitly defined, when the analyses provided for the language take place, how much of the computational resource they can take advantage of. This however 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 in general and in particular, taking one of the C++ specific analyses and researching on its complexity. Extensions to the *JetBrains MPS API* are proposed to get a better control over the analyses run-time.

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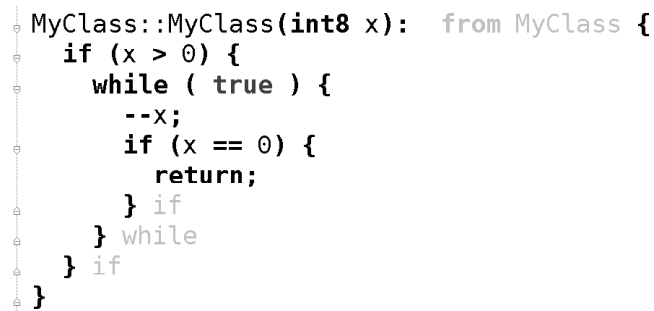
Secondly, there is no need to format the code on the level of indentation and look, since it is only needed for textual code.

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

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 **??**.

2.2 Modular Language Engineering

2.2.1 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].

The nodes of the *AST* are described through giving their types. The node type in projectional editing is called a *concept*. *Concepts* are very similar to classes in object oriented programming languages. They feature inheritance, they can implement interfaces, they can have internal data, similar to member fields, and they can feature behavior, similar to member functions. The difference with classes, however, is that the member fields are not usually encapsulated.

The edges of the *AST* are not described on their own, but instead as a properties of nodes. A node can have children, or can reference other nodes. An example of child relationship, can be a condition expression of the `if` statement. An example of reference relationship can be a local variable usage, referencing the declaration of the local variable. Child and reference relationships can have different cardinality, with minimal border from 0 or 1, to the maximal border of 1 or N, where N stands for just "many", or several.

The cardinality itself, is not usually enough, to restrict as desirable node relationships. Special constraints can be added and checked for each relationship, which describe precisely, or provide procedure to check, the validity of a relationship being established. The

---

[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 a possible task, which leads to the increasing parser complexity, known problem in particular in the C++ area

projectional editor must inform the user, every time, the constraint was not satisfied, so that the user has a chance to correct the code, to match a valid *AST* description.

For the user to be able to manipulate the *AST* for each node *concept* an editor has to be created. The editor defines, how a node of a given *concept* should be represented to the user, which editing operations, and how, the user may perform on the node.

The minimal set of data was described above, which has to be defined for a language, to enable the projectional editing for it.

Additionally, constraints may be refined, involving some usual for typed languages type restrictions and checks. Generators can be added to transform *ASTs* given in a language. Text generators can be defined to generate a text code from an *AST*.

Behavior can be defined for a concept, to provide some method-like functionality to it. Additionally, some user-invokable functions can be described, to perform manipulations with the *AST*.

The process of defining a modular language in the *JetBrains MPS* environment is described additionally with practical details in the Section 3.1.

2.2.2 Language Modularity

As *concepts* feature inheritance, it is possible[2] to use a child concept at a place where a parent concept could be used.

The inheritance works over the language borders, allowing to create the child concept in a language L2, separate from the language L1, where the original parent concept has been described. Thus the language L2 can be seen as a modular extension to the language L1.

The modular *DSL* creation is discussed in [5], [9]. The language modularity, in a context of *JetBrains MPS* is described in [3], [12].

The main focus of this work is a construction of the C++ programming language on top of the *mbeddr* C implementation. Thus the modularity in the language engineering plays a key role in the work.

While extending the C language of the *mbeddr* with C++ specific *concepts*, all the aspects of the language description (see previous subsection) have to be extended. The newly introduced *concepts* for nodes of the *AST*, typical for C++, must inherit from some *concepts* of the original *mbeddr project* C language. Not only new nodes and edges are introduced, but also constraints and other language description aspects have to be made incorporating the newly introduced concepts. The practical side of the language modularity and extensibility is discussed throughout this work.

---

[2]to some extent, the extensibility is described separately in one of the following chapters

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

JetBrains MPS stands for Jetbrains Meta Programming System. In this *IDE*-like software it is possible to develop *DSLs*. The approach taken in the *JetBrains MPS* is rather unique, and it is considered to be advantageous in many ways, [11].

In general, the way the language is describe in *JetBrains MPS* coresponds to the way, described in the Section 2.2. Here I will describe the process in practical details, as it is crucial for understanding the practical part of this Master Thesis.

In this section I will go through a definition of one *concept*, describing the facilities, *Jet-Brains MPS* provides to support it. Each *concept* is described by several views on it.
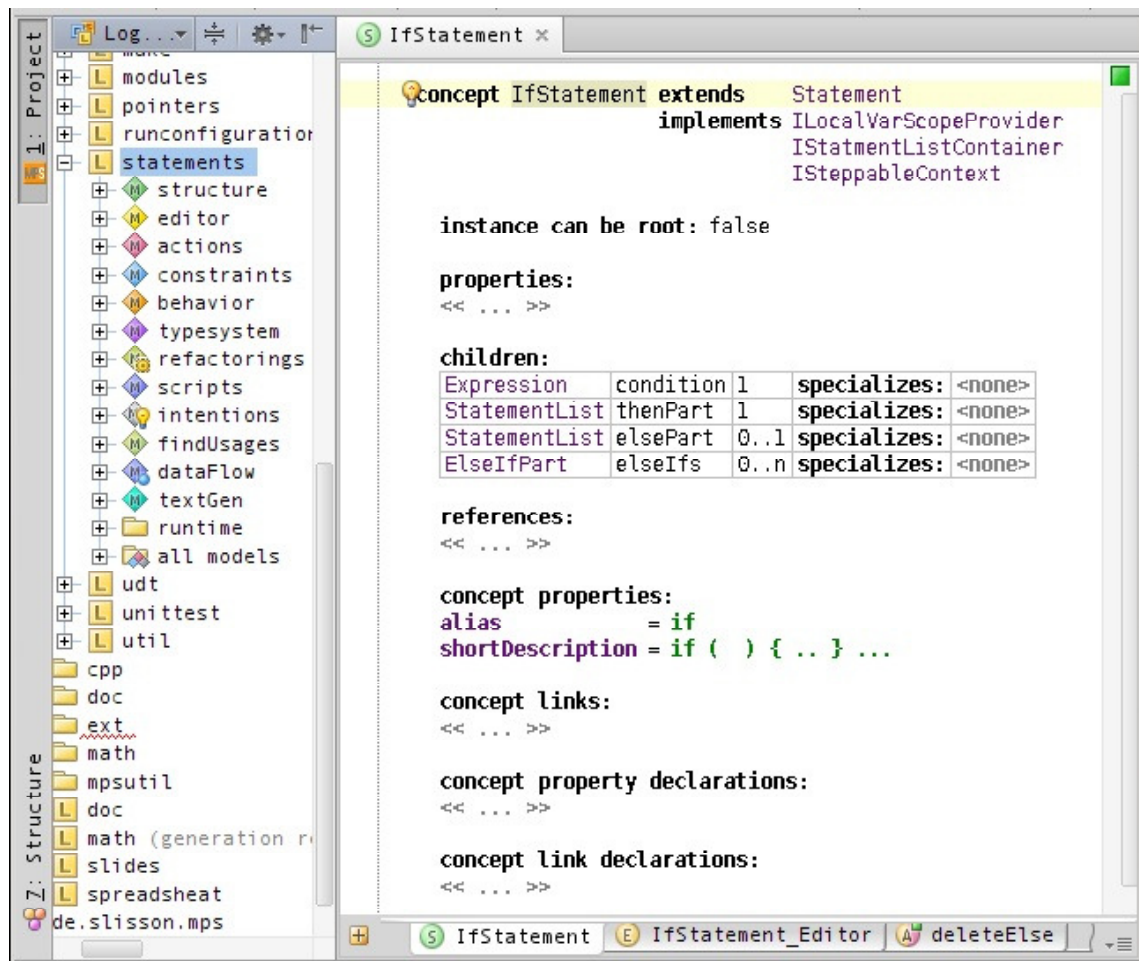
Figure 3.1: *JetBrains MPS* User Interface, `if` Statement *Concept*

Among such views there are *concept* declaration (or language structure) view, editor view, behavior view, constraints view, type system view, generator view and few more views. As a language consists mostly of the included in it concepts, the whole language is presented by the mentioned views as well, where each view on the language contains all views of the kind on each language *concept*. Left part of the Figure 3.1 demonstates the views on the *statements* language.

The representation of a new language created in concepts and views to them, present a seamless approach to creating a new language with a projectional editor for it.

The *mbeddr project* is a software, separate from *JetBrains MPS* but based on it, representing an extensible C language implementation with extensions. I will use *concepts* from the *mbeddr project*' throughout this section as examples to demonstrate MPS. Different C language parts are going to be decomposed into *concepts* and these concepts are going to be defined using *JetBrains MPS*. The reader should not confuse though the *mbeddr project* and *JetBrains MPS* itself: the former is a software, developed in the latter and is used to demonstrate the latter.

3.1.1 Concept Declaration

Concepts, as it is described in the Section 2.2 represent a class-like types for nodes of an *AST*. This terminology is kept in *JetBrains MPS* and MPS concept has the same meaning as *concept* term used in the Section 2.2. I use the term "concept" both in general, to describe an *AST* node type, and in particular referring to an MPS concept.

The Figure 3.1 on the right part demonstates a declaration of `IfStatement` concept from the *mbeddr statements* language. It corresponds to the `if` statement of the C language.

At first, the *concept* is named, and a *base concept* is defined. The `IfStatment` *concept* inherits from the `Statement` *concept*. This allows the `IfStatement` to be used at any place at which the `Statement` could be used, and inherits like in object oriented programming all data and behavior of the `Statement`.

Next, it is defined, which interfaces the *concept* is going to implement. For example, by implementing `ISteppableContext`, the `IfStatement` supports the *mbeddr* debugger when stepping in the body of the `IfStatement`.

The "instance can be root" property defines, if it is meaningful, to create a *concept* without a parent *concept* for it. In the case of the `IfStatment` it does not make sense, as the statement should belong to some block. The `true` value can be used, e.g. for modules, as they do not have any outer concepts, and can be seen as a document in *JetBrains MPS*.

The "properties" part defines if the described *concept* instances should have some primitive type data fields (string, boolean, int). An example of a property could be a `name` property of a variable declaration. The `IfStatement` *concept* does not specify any properties, neither does it inherit any from the `Statement` concept.

The "children" section describe which nodes can be children on the *AST* of the `IfStatement`. Each child is assigned with a role and cardinality. For example, the `IfStatement` should have exactly one child of *concept* `Expression`, it has a role "condition".

The "references" section describes in the similar way as in the "children" section, which nodes could be referenced by the node of a given *concept*. Referencing can be used, to bind a given node, to a node, located somewhere else on the *AST*. As an example, a variable

usage in expression shall reference the variable declaration, to express precisely, which variable is used.

Finally, some attributes of a *concept* follow, which do not have a primary importance for the discussion here. The "alias" is used to name a *concept* in a short way, to allow for quick instantiation in the editor. The "short description" is shown to hint a programmer on the alias meaning. A *concept* can be made abstract in the "concept properties" section. Abstract *concepts* are purely used in inheritance to create other non-abstract concepts with a common parent.

JetBrains MPS separately allows to define so-called *interface concepts*. Interface concepts are *concepts* which can not be instantiated, but which serve as a base for inheritance and implementing and interfaces as Java classes do. A *concept* can have only one base concept, but can inherit from/implement many interface concepts.

3.1.2 Editor View

The editor view, Figure 3.2, is designed to give a look for a node of a *concept*, and a way to input it. This is where the projection of an *AST* node is defined. As editors are mostly defined to look like text, a program in the C++ programming languageimplemented in MPS looks almost like a regular C++ code.

In the editor view one defines an editor for a *concept*. *JetBrains MPS* introduces here a bit confusing terminology, calling one editor for a given *concept* concept an "editor concept". Thus an error message "An editor concept not found for a concept X" would mean that no editor has been defined for the *concept* X in the editor view for it. In this work I call the content in the editor view for a *concept* X an "editor for the *concept* X". The same terminology applies to all the following views related to a single *concept*.
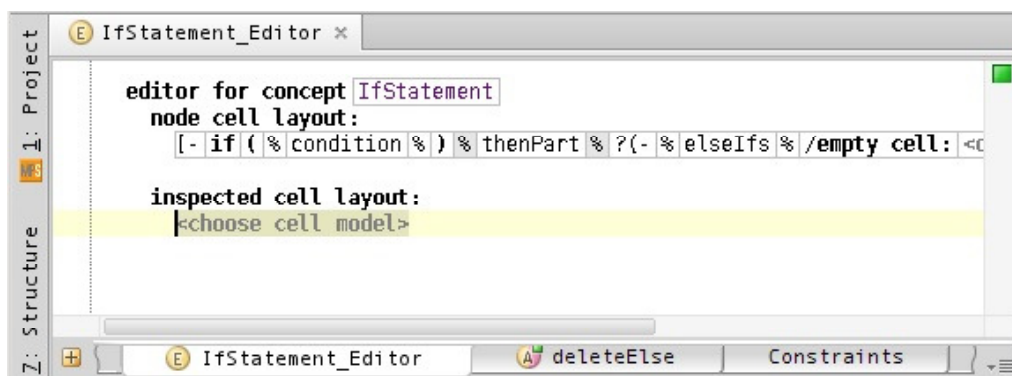

Figure 3.2: Editor View for the `IfStatement` *Concept*

The editor for a *concept* defines the visual representation for a node of the concept, using special syntax. For example, Figure 3.2 defines and editor for the `IfStatement`. At first the "constant" non-changeable by a programmer text is given, which is "if" and "(". Then the child `condition` is referenced, so that after the "if(" the user will be able to input an expression for the condition of the `if` statement. The editor can be configured to show or hide some parts of a node, depending on some condition, e.g. hiding `else` part of the `if`

statement, if it is not defined anyhow. Editor are also responsible for all the interaction, a user experiences when editing a code in *JetBrains MPS*.

3.1.3 Behavior View

The behavior view, can be used to define certain behavioral methods for a concept. A concept is represented there similar to a Java class, and it is possible to define the methods in a Java-like language. *Concept* inheritance is taken into account like in Java.

A *concept* constructor can be defined there to initialize by default a newly created node of a *concept*.

Figure 3.3: Behavior View for the `LocalVarRef` *Concept*

The Figure 3.3 shows the behavior of the `LocalVarRef` *concept*. This *concept* represents in the *mbeddr project* C language an expression, referencing a local variable. The local variable declaration is stored as a reference `var` in the *concept* nodes.

Two convenience methods are defined for the `LocalVarRef` *concept*, to get an easy access to the local variable properties.

3.1.4 Constraints View

The constraints view can be used, to limit in a desirable way *concept* property values and relationships, a node of the *concept* can have to other nodes.
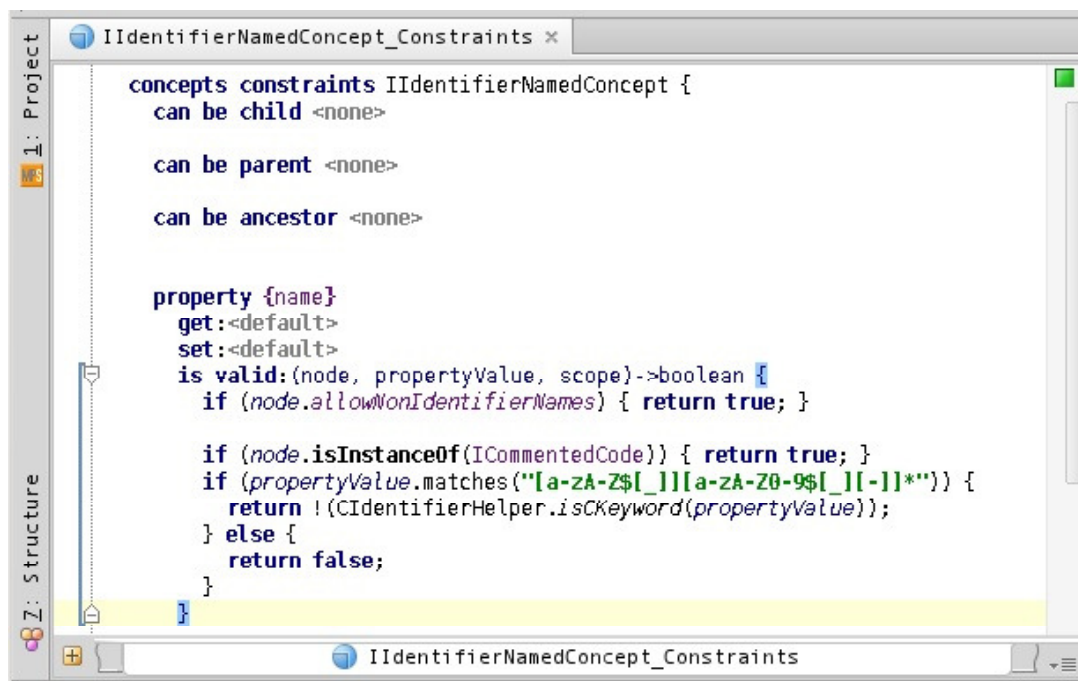
Figure 3.4: Editor View for the `IfStatement` *Concept*

The Figure 3.4 shows constraints defined for a property `name` of the `IIdentifierNamedConcept` *interface concept*. All *concepts* which have a name, which must confirm to the identifier naming restrictions, can implement the `IIdentifierNamedConcept`, to immediately get the desired characteristic.

The `name` property is programmatically restricted by the use of Java-like code snippet, Figure 3.4. In a similar way relationships to children and referents can be restricted.

It is possible to create a pure Java class withing an *JetBrains MPS* language, and use it in almost any *concept* view in the *JetBrains MPS*. In the `IIdentifierNamedConcept` *interface concept* constraints the `CIdentifierHelper` class was used to check the name property on collision with the C language keywords.

3.1.5 Type System View

JetBrains MPS has a special support for creation of typed languages. Types are mainly used in expressions. An expressions may count on a certain sub-expression to have a given type, or a type, compatible with it. Whenever the expectation does not meet the reality, a warning or error can be displayed to a programmer.
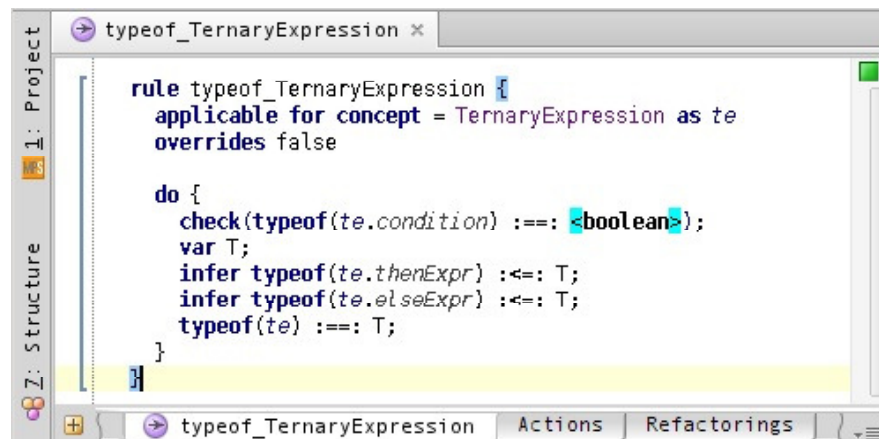
Figure 3.5: Type System View for the `TernaryExpression` *Concept*

A *concept* representing nodes which can have type in *mbeddr* inherits from `ITyped` interface concept. In the type system view certain rules must be defined with the use of a special language, which define the type or type comparison rules for a *concept*. Moreover, the type system language can be used to infer a type for a given node, Figure 3.5.

The Figure 3.5, demonstrates the use of the *JetBrains MPS* type system language to infer a type of the `TernaryExpression` *Concept* node. The syntax is on the one hand self-explanatory when reading, but on the other hand could be rather confusing when crafting.

3.1.6 TextGen View

To make use of the code in the projectional editor, further tools must be invoked on it, e.g. parser, compiler, etc. Normally they work with a textual representation of the same code. In order to obtain the textual code from the *AST* in the projectional editor generators are invoked.

Generators can be of two kinds. The kind of generator is dedicated to transform an *AST* in one *DSL* into another *AST* represented in a, usually, lower-level language. The second kind of generators is dedicated to transform an *AST* into text. Such generators are called "*TextGens*" in *JetBrains MPS*.
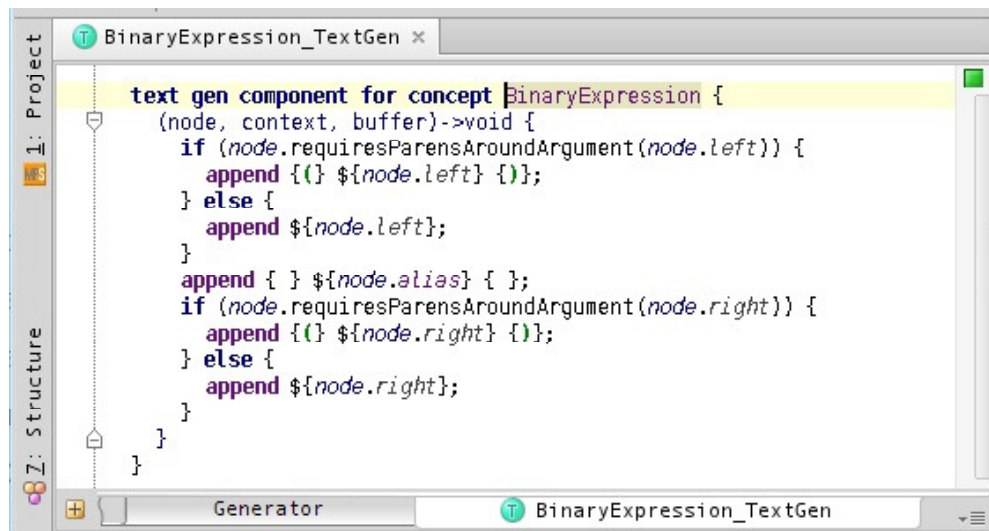
Figure 3.6: TextGen View for the `BinaryExpression` *Concept*

The Figure 3.6 demonstrates how a node of a `BinaryExpression` *Concept* is converted to text. It is noteworthy to say, that when rendering to text the `left` and `right` sub-expressions, the corresponding *TextGens* are invoked, making the text generation recursive.

3.1.7 Generator View

The generator view is one of the most complex *JetBrains MPS* views. It servers the transformation purposes of one *JetBrains MPS* language into another one. This view does not have a strong connection to the present work, as the generation of the projectional C++ programming language is mostly performed by *TextGens*.

3.1.8 Intentions

JetBrains MPS intentions are special procedures which can be used for automatic manipulations on the *AST* with a node of a given *concept*.
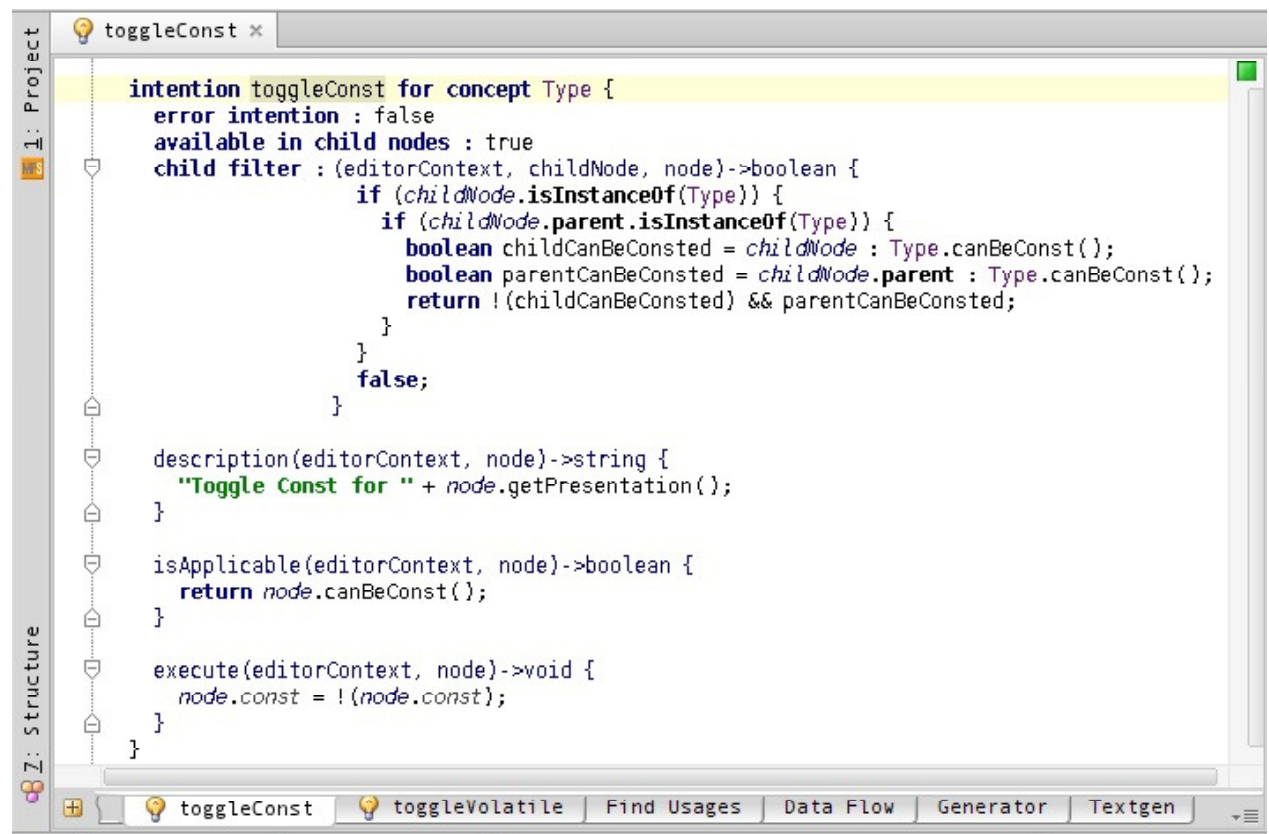
Figure 3.7: Toggle Const Property for a `Type` *Intention*

The Figure 3.7 demonstrates an intention, used to modify the `const` property of a given type.

For an *intention* to be defined, one has to name it, specify a *concept*, to which the *intention* is applicable, provide a textual description for it, and finally specify the desired effect.

Intentions are accessible in the projectional editor from a context menu, when focused on a target node. They represent a useful mechanism to support a programmer with various automations, including automatic code generation.

3.1.9 Other MPS Instruments

JetBrains MPS provides other instruments to enhance languages.

Actions are used to automate node deletions or editing. For example, deleting an array indexing expression, could be made to provide a substitution, an array expression itself. Such behavior may seem more natural to a programmer, used to text editing.

JetBrains MPS provides a special support for refactorrings. Special code snippets in the Java-like language can automate routine operations. As an example, factoring out a local variable from an expression can be taken.

3.2 MBEDDR Project

The *mbeddr project* is a software built with the use of *JetBrains MPS*.

The *mbeddr project* represent mainly an implementation of the C programming language in the *JetBrains MPS* environment. Having embedded systems and software for them as the main focus, *mbeddr* provides certain language extensions to empower the programmer in the mentioned domain [10], [11].

Being a different language the C++ programming language shares a lot of commonality with the C programming language. As *JetBrains MPS* allows, to some extent, see 2.2.2, incremental language construction, the *mbeddr project* represents a suitable basis for the C++ programming language implementation in *JetBrains MPS* .

The use of *mbeddr* however, could not be purely incremental, and required some changes to the *mbeddr* itself. The changes were introduced however, in a way to make *mbeddr* simply more extensible in general, and not by adopting it to the current work needs.

In this section I describe the *mbeddr project*, as the current work is based on it.

No matter the *mbeddr project* has (one) C language with extensions as an outcome, internally, as a *JetBrains MPS* software it is represented as several *JetBrains MPS* languages. In *JetBrains MPS* a language corresponds to a module.

All *mbeddr* languages are named starting from *com.mbeddr.* name part. In this document I usually ommit it, keeping the last word of the name only. E.g. *com.mbeddr.expressions* is called simply *expressions* here.

3.2.1 *mbeddr* Expressions Language

The *expressions* language contains definitions for all expressions, possible in the *mbeddr* C language. As in object oriented programming languages *concepts* of the `expressions` language form inheritance hierarchies. *JetBrains MPS* is capable of showing a given *concept* in a hierarchy.

Figure 3.8: *Concept* Hierarchy Example

The Figure 3.8 shows a hierarchy for the `MinusExpression` *concept*. In a similar way all expressions of the C programming language are implemented in the *expressions* language.

Whenever there is a need in the C++ programming language to extend the C programming language with a new expression kind, like object member reference, *new* expression and so on, a point of inheritance has to be found in the *mbeddr expressions* language to base a new *concept* on it.

Additionally, the *expressions* language defines C language types.

Figure 3.9: *mbeddr* Type Hierarchy Example

All *concepts* corresponding to C types are based (directly or indirectly) on the `Type` *concept*. For example, the hierarchy of `IntType` *concept* is demonstrated in Figure 3.9.

In order to add a type to *mbeddr* C language, one should inherit from the `Type` *concept* as well. Such inheritance automatically allows the new type to appear at all places, where a type in general can be found in the C language or its extensions.

3.2.2 *mbeddr* Statements Language

mbeddr statements language contains definitions for C language statements. The `Statement` *concept* serves as the base for inheritance, and represent by itself an empty line, or no-statement.

In order to create a new statement, like `delete` statement in C++, the inheritance should start from the `Statement` *concept*.

```
int16 abs(int16 x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  } if
} abs (function)
```

Figure 3.10: Example of Multiple Languages Used

The *statements* language actively uses the *expressions* language, Figure 3.10. In the *mbeddr* code snippet the nodes coming from *statements* language are marked green, and the nodes, coming from *expressions* language are marked yellow[1]. As the example shows, `if` statement and `return` statement are coming from the *statements* language, but inside they contain as children expressions. This is an example of language modularity in *JetBrains MPS*, used by *mbeddr*.

---

[1]not marked with color is a *Function* concept, which comes from *modules* language

4 Projectional C++ Implementation

5 Evaluation

6 Conclusion

Appendix

Glossary

action is a JetBrains MPS term, which corresponds to a automations, specific to actions, occurring while editing. 20

API Application Programming Interface. 3

AST Abstract Syntax Tree. 1–3, 5–9, 13, 14, 17, 18, 31

base concept is a JetBrains MPS concept, which surves as an inheritance base or parent concept for a given concept, e.g. Statement concept for IfStatement concept. 13

concept is a class-like type, describing a node type in the AST when talking about projectional editing. 1, 2, 8, 9, 11–18, 20, 21, 31

DSL Domain Specific Language. 5, 6, 9, 11, 17

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

Extended Embedded C++ is a improvement of Embedded C++, bringing back the omitted language features, and a memory-aware STL version. 1

IDE Integrated Development Environment. 2, 3, 5, 11

intention is a special procedure in *JetBrains MPS* which can be used for automatic manipulations on the *AST* with a node of a given *concept*. 18–20

interface concept is a JetBrains MPS concept, which surves for inheriting a concept behavior interface, can not serve as a base concept. 14, 16

JetBrains MPS is a language engineering environment allowing to construct incrementally defined domain specific languages. 1–3, 9, 11–18, 20, 31

mbeddr same as mbeddr project. xi, 1, 3, 9, 13, 17, 20, 21

mbeddr project is a JetBrains MPS based language workbench, representing C language and domain specific extensions for the embedded software development. 3, 9, 11, 13, 15, 20

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. 6

STL Standard Template Library. 1

TextGen is an special kind of generator in JetBrains MPS, dedicated to produce a textual
representation of a node of a given concept. 17, 18

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

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

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

[8] IAR Systems. Extended embedded c++, http://www.testech-elect.com/iar/extended_embedded_c++.htm.

[9] Eric Van Wyk. Modular Domain-Specific Language Extensions. In *1st ECOOP Workshop on Domain-Specific Program Development (DSPD)*, 2006.

[10] 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.

[11] Markus Voelter. Embedded Software Development with Projectional Language Workbenches. In Dorina Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010. Proceedings*, Lecture Notes in Computer Science. Springer, 2010.

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

[13] 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.