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, [1]. 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 [3].

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++*, [4]. 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*, [5].

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, [6].

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, [7]. 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*. The C++ implemented in *JetBrains MPS* and discussed in this work I call the *Projectional C++*.

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, [7], 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.

Finally, it is fair to say, that some practical foundations for the *Projectional C++* have been already grounded, before the start of this work by the *mbeddr* team. Some of them, were kept (reference type, templates partially) and just described and analyzed here, some of

them were considerably reworked (classes, inheritance, encapsulation and polymorhism). Couching from the *mbeddr* team made me changing some of the implementation aspects, as from I planned originally (namespaces, operator overloading partially). And some parts were built, of-course, new from scratch and were not (at least yet) anyhow influenced by *mbeddr* team at all (analyzes, construction and copying, some more). The evaluation of the experience, gained by me during the practical implementation part, which results into the extensibility analysis, research on complexity of cheks and the run-time for them, and general projectional guidelines represent my own theoretical focus and comittment.

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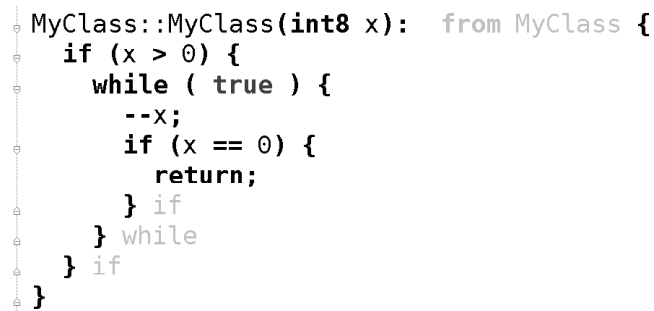
Secondly, there is no need to format the code on the level of indentation and look, since it is only needed for textual code.

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, [8]. 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, which I conside to be of practical value in the Section 5.2.

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. This creates a great opportunity for language extensibility. In order to extend a language at some point, just a passing base *concept* hast to be determined and inheritted from, by a new *concept* which is meant to provide the language extension. The new concept is immediately able to be used in place of the base concept. As an example, one could think of extending statements of the language. The only new statement is needed, a *concept* has to be created, which represents the new statement, and enabling the new statement concsist of just inheritting it from the base statement concept, which exists in the language extended.

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 [9], [10]. The language modularity, in a context of *JetBrains MPS* is described in [7], [11].

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

2.2.3 Text Language Importers

When recreating an existing text language in projection, it is natural to support some libraries with it. The libraries, or a code base in another form, is present however not in the projectional form, but in the native for the language text code. Thus, for the user of the language, to be able to use the code base, it has to be *imported* to the projectional editor.

The import process consists of parsing the text code getting an *AST* and converting the *AST* in an *AST*, as can be described by the language in projection.

The task of import contains usually several principal challenges. At first, the native text language can have more constructions, than the version described in the projection. This happens, due to the intentions to omit some of them in projection[3], absence of them due to the simplicity of projectional implementation of the language, no need for them in projection, because the constructions are only specific to the textual nature of the code[4]. Second, the projectional language can contain more information about the nodes, than is present in the textual language, thus the information has to be generated or manually added later.

Additionally the technical work to create an importer can be considered significant, especially for complex languages, like the C++ programming language.

---

[3]dangerous constructions, like `reinterpret_cast`

[4]for example, preprocessor directives

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, [12].

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, *JetBrains MPS* provides to support it. Each *concept* is described by several views on it.
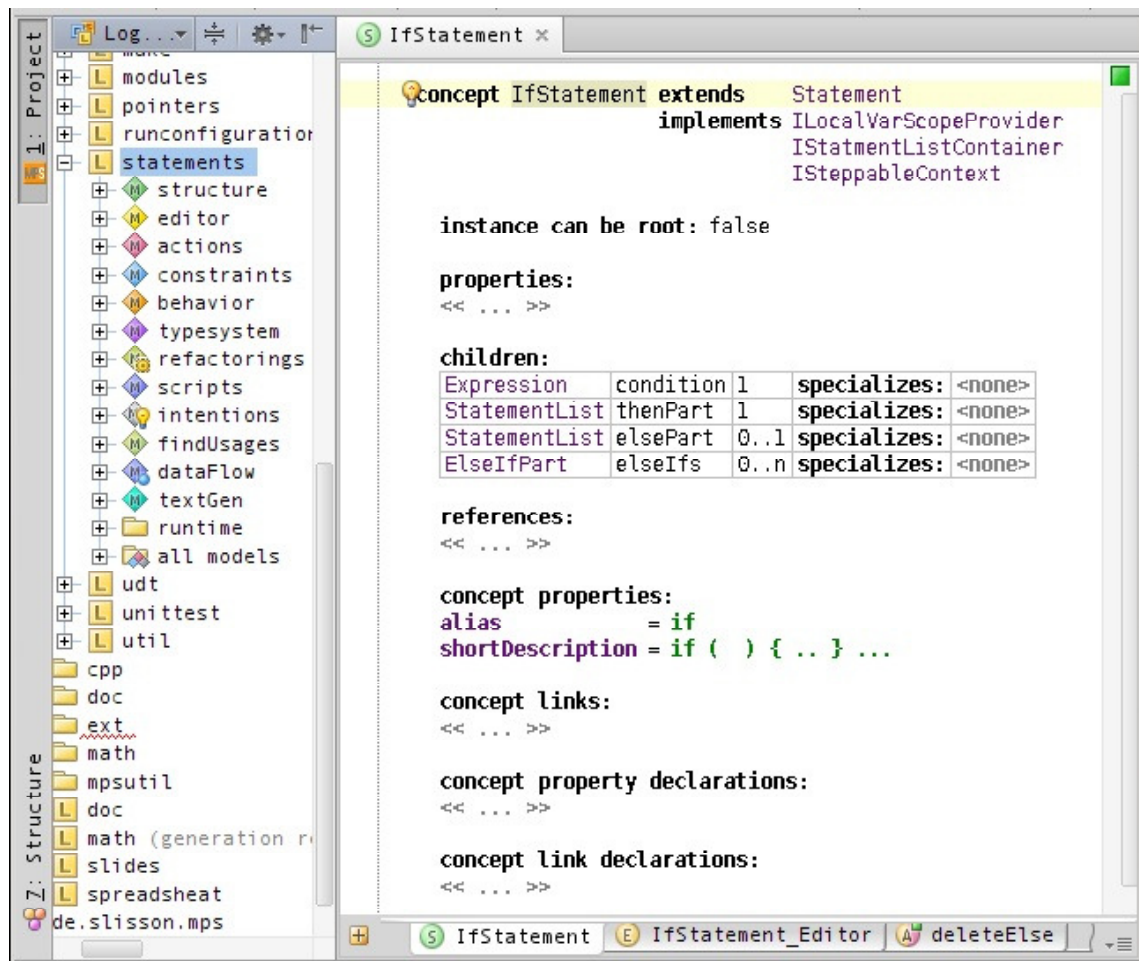
Figure 3.1: *JetBrains MPS* User Interface, `if` Statement *Concept*

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 demonstrates 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*.
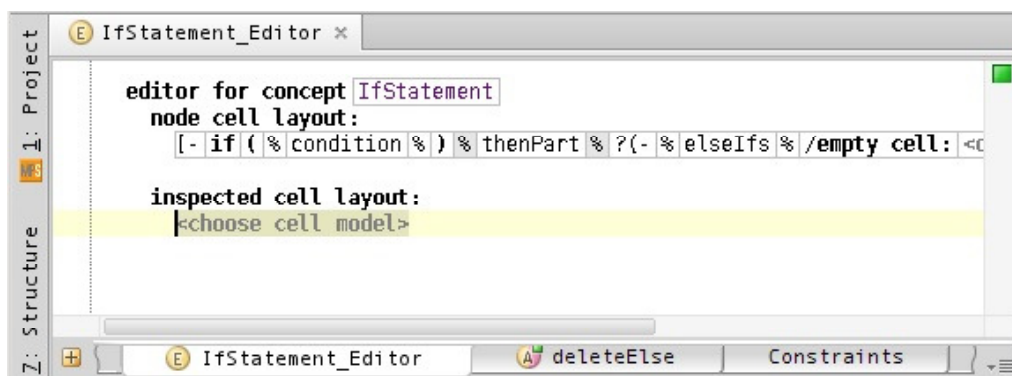
Figure 3.2: Editor View for the **IfStatement** *Concept*

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.
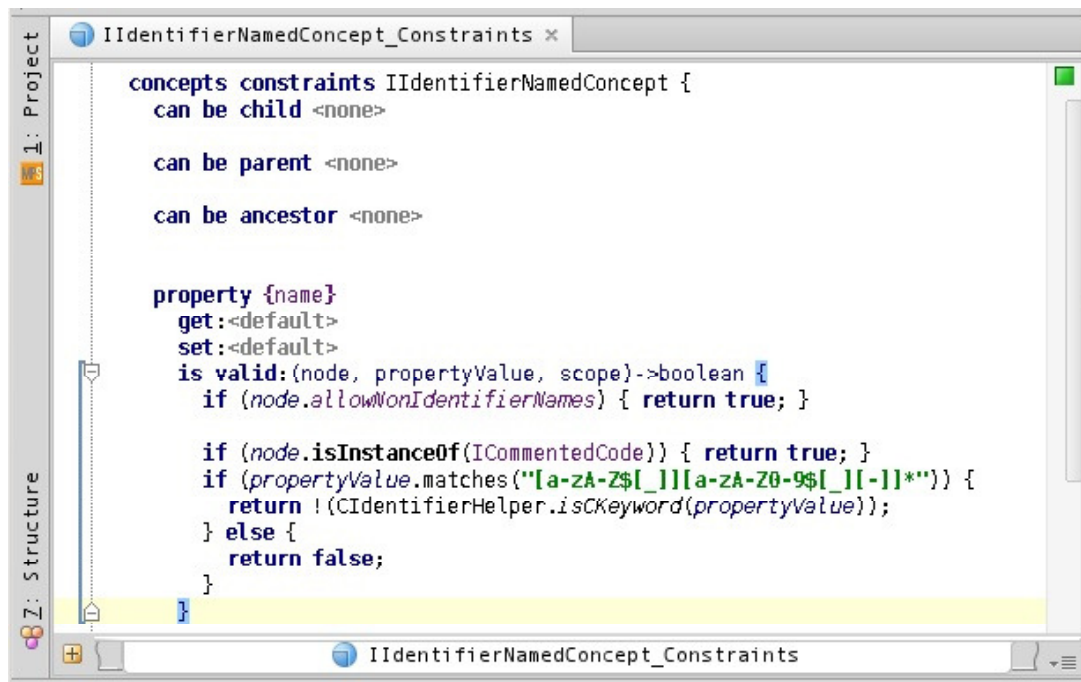
Figure 3.4: Editor View for the **IfStatement** *Concept*

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.
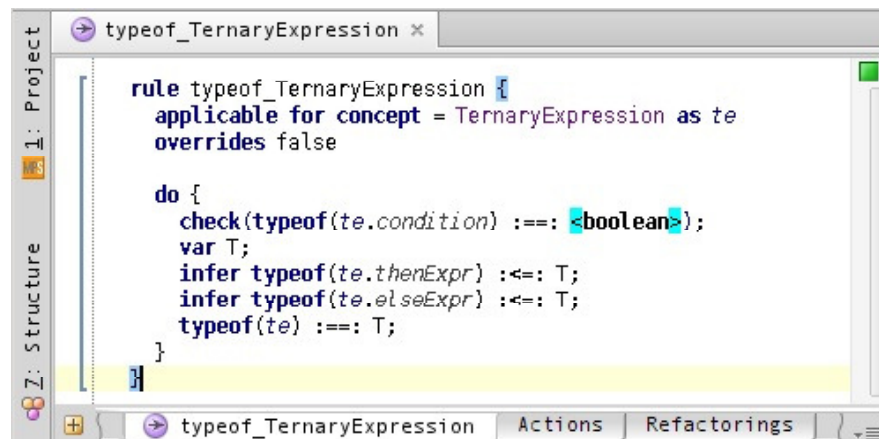
Figure 3.5: Type System View for the **TernaryExpression** *Concept*

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*.
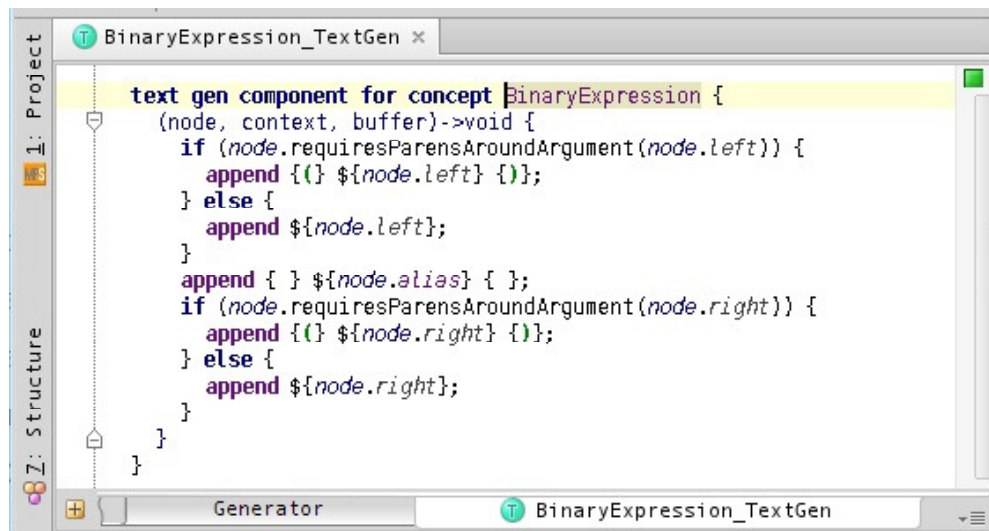
Figure 3.6: TextGen View for the **BinaryExpression** *Concept*

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*.
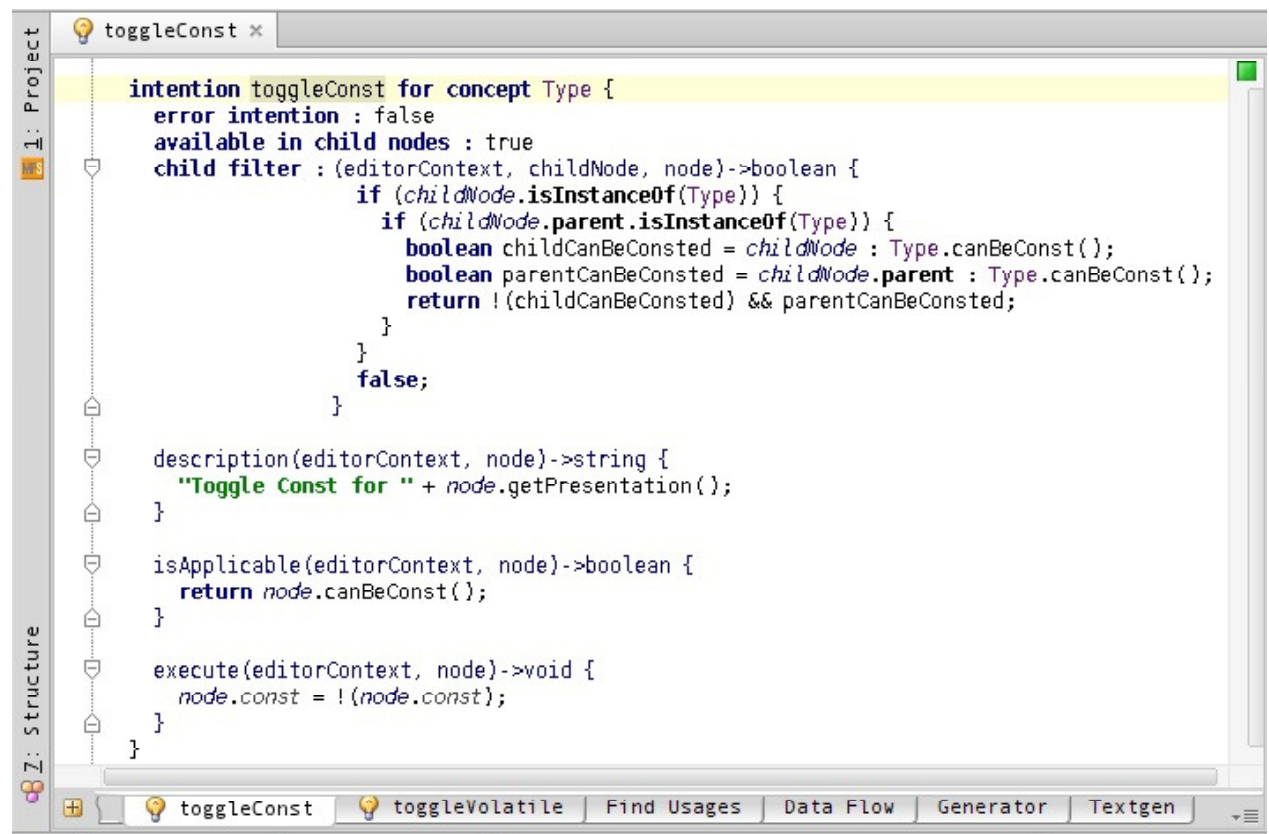
Figure 3.7: Toggle Const Property for a **Type** *Intention*

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.

There is also a special kind of *intentions*, called "error intensions". They are not anyhow fundamentally different from the usual *intentions*, except their special purpose to fix an error when one occur. Error *intentions* are visualized using a red bulb icon in *JetBrains MPS*.

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 [13], [12].

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.

```
Hierarchy for MinusExpression
  BaseConcept (jetbrains.mps.lang.core.structure)
    Expression (com.mbeddr.core.expressions.structure)
      BinaryExpression (com.mbeddr.core.expressions.structure)
        BinaryArithmeticExpression (com.mbeddr.core.expressions.structure)
          MinusExpression (com.mbeddr.core.expressions.structure)
```
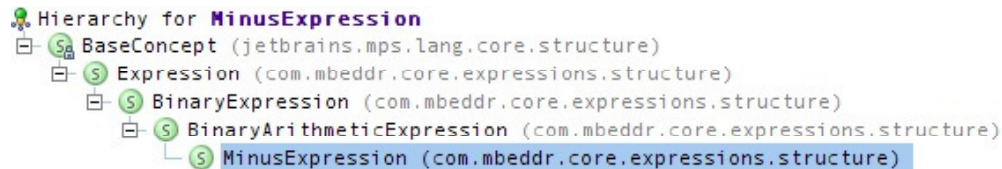
Figure 3.8: *Concept* Hierarchy Example

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.

```
Hierarchy for IntType
  BaseConcept (jetbrains.mps.lang.core.structure)
    Type (com.mbeddr.core.expressions.structure)
      PrimitiveType (com.mbeddr.core.expressions.structure)
        PrimitiveBasicIntegralType (com.mbeddr.core.expressions.structure)
          IntType (com.mbeddr.core.expressions.structure)
```
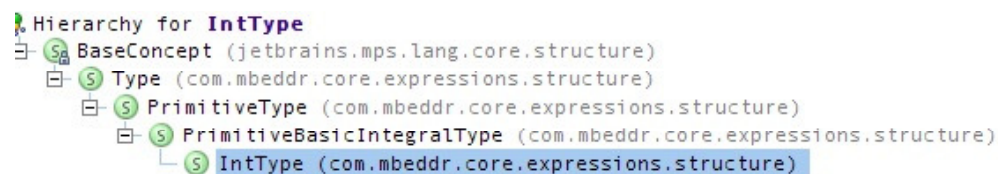
Figure 3.9: *mbeddr* Type Hierarchy Example

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 Together

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*.

3.2.3 Modules in mbeddr

In C (and in C++ as well) there is no clear concept of a module. The *mbeddr project* improves on it, defining modules, [12]. A C module is a *concept*, from which the header and the .c files are generated in mbeddr. Flagging an object (function, variable, structure, etc.) in a module as `exported` causes the declaration of the object to appear in the header, and thus the object starts to be accessible by other modules.

An issue with the C programming language is that there is one and only global namespace. The *mbeddr project* improves on it by introducing so-called name mangling. All names of the module contents are prefixed with the module name, when generated to the C text code. Thus the object with the same name but from different modules do not cause a name clash.

The implementation of modules in *mbeddr* can be found in the *modules* language. Functions are described there as well, for the **Function** concept example, see the Figure 3.10, not marked with colors part.

Modules are further included into *JetBrains MPS models*, which correspond to one single *JetBrains MPS* file unit. Each model should have a node of **TypeSizeConfiguration** and **BuildConfiguration** *concepts*. In the former, the sizes for types must be given, in the latter, all directives needed to compile the generated from the *model* text.

3.2.4 Pointers and Arrays in mbeddr

In the *pointers* language array access and pointer dereferencing expressions are defined as *concepts*. Similar syntax to them have overloaded operators with classes. Thus this language could also be extended by the *Projectional C++*, if some reuse is possible there as well. This is discussed in detail in the Section 4.3.

---

[1]not marked with color is an instance of the *Function* concept, which comes from the *modules* language

3.3 The C++ Language

This work is basically an implementation of the C++ programming language in *JetBrains MPS*. No matter, the whole work is about the C++ implementation, it is infeasible to describe the language in detail itself here. Thus I give a references to literature about the language in this chapter only. Certain aspects of C++ are raised during the description of the C++ implementation in *JetBrains MPS* itself in the Chapter 4.

Various literature is available, to get acquainted with the language closer. The most complete guide to the language, covering *STL* as well is [14]. The more easy-to-read and more suitable for beginners book on C++ is [15]. A newer book, oriented towards intermediate level C++ programmers and updated to the recent C++11 standard, [16], and describing *STL* as well, [17].

The collection of techniques to get more effective C++ programs is [18]. Templates are explained in detail and researched in, [19]. The book dedicated to ad hoc polymorphism and advanced template meta programming can be also of interest as an approach to language engineering in a certain sense, [20].

The C++ programming language is a mature language, with long traditions, and high flexibility, [21] can serve as an example. Although the language is also famous for being complex. It will not be possible to simplify language, removing features from it, which will restrict the language use, most importantly the *STL* support. Thus in this work I try to research, how the editor can be more supportive for the user of the existing complex language, to eliminate usual mistakes made while programming, as well as provide help in structuring the code.

4 Projectional C++ Implementation

4.1 C and C++

Initially C++ appeared to be an extesnion to the C language, called "C with Classes" [22]. Till now the high degree of commonality can be found between the two languages. Mainly, the most of the C code is going to be valid C++ code.

Some differences exist however, I introduce them here, together with the way they are adopted in the *Projectional C++*.

4.1.1 Reference Type and Boolean Type

Among primitive types and operations there are two major differences relevant to practice. They are discussed in the following subsections.

Reference Type

The reference type is basically a new construction in C++ which represent the old notion of pointer in C with a different syntax. Infact behind the reference type stands a pointer to the base type, and it is dereferenced when needed.

The syntax for a variable of type T can be used with the variable of type reference to T, or &T as it is designated in C++. Thus the programmer benefits from the shorter syntax in comparison to the pointer dereferencing.

It comes especially handy when passing arguments to methods by reference, which avoids creating a copy of an object to pass by value, and still preserves the short syntax without pointer dereferencing.

Listing 4.1: Reference Type in a Copy Constructor

```
class MyClass {
    MyClass( const MyClass& original );
};
```

The value of the reference type in C++ is bigger however, than just a new syntax for pointers. A constant reference to a class object has to be used in a constructor to give it a special meaning of the copy constructor for the class, as the Listing 4.1 demonstrates. This changes the whole semantics of the class, making it copyable. It is discussed in detail in 4.2.1.

The reference type itself is implemented rather straightforward and similar to the pointer type of *mbeddr*. It iherits the **Type** *concept* of *mbeddr* as described in the 3.2.1, and has a base type as a child.

Boolean Type

The C++ programming language has a special `bool` type to represent the two logical values. There is no such type in C.

No matter, the `bool` type is not present in C, for the convenience of the user the *mbeddr* C implementation introduced a type `bool`, which is translated to `int8_t`, together with `true` and `false` values, converting to `1` and `0` respectively. This implementation can arguably be considered better than the original C++ `bool` type present in the generated code, because the C++ programming language standard does not define explicitly the size of the `bool` type [16]. This can be suitable for the embedded domain better, conforming to the *mbeddr* spirit.

In the context of embedded systems, which *mbeddr* targets, it is often very important to know precisely, with which type the user is operating, as the limited resources are important to consider. Substituting the `bool` type with the `int8_t` type ensures that the size of the `bool` variables is known. Also, it can be changed as needed in the **TypeSizeConfiguration** in each model created with *mbeddr* separately, see 3.2.3.

The C++ standard explicitly allows the `bool` type to participate in integer promotions. This ensures further the compatibility of the custom-written text code, which may use actual `bool` type, with the code generated with substitution of `bool` to `int8_t`, as the user `bool` will be promoted.

Among limitations of this approach one can take as an example `std::vectoor<bool>`. Since the word "bool" itself is never generated, it is not possible to use the specialization of the template, which ensures storage optimization, through, though, higher processing load when extracting a value from such a `vector`.

The `bool` type from *mbeddr* is kept in the *Projectional C++*. This decision is arguable, and can, of-course, be changed in the future.

4.1.2 Modules and C++

The C++ programming language is an improvement over the C language by itself. There is, however, no concept of a module, which can bring certain advantages and disadvantages. As a disadvantage can be named a potential disorder in placing implementations for classes and functions, as the language itself does not enforce a clear one place for them. Potential name clashes are also disadvantageous. As an advantage, conditional compilation can be used in C++ to include different files with different implementations, which is often used to create cross-platform code. Additionally, classes can be used analogously to modules.

The name clash problem is solved in C++ by introducing the namespaces. The programmer defines a namespace and then places there all the exported code. The nodes with equal names from different namespaces do not clash.

C++ however does not elaborate on the disadvantage of having declarations and implementation not organized in one place, and being fully disjoint. *mbeddr* compensates on it, 3.2.3.

After comparing the two approaches of *mbeddr* and of the C++ programming language to improve on C, the hybrid approach has been picked. Firstly, the C++ implementation modules have been introduced. And secondly, the namespaces can be defined and used.

In *Projectional C++* modules are program modular elements, where a programmer declares and implements C++ classes, and the .h and .cpp files are automatically generated from them. This eliminates the need to create and control two files for each logical program element, like a class, and keep them synchronized with each other. For example, changing an interface of a class would require at first opening the header file, and introducing the changes in the declaration, and then rewriting the corresponding part of the .cpp file in strong correspondence with the header. This work decrease in order to change an interface can improve productivity.

Namespaces are introduced in the *Projectional C++* to avoid inter-modular name clashes, they work similar to the namespaces in C++.

As auto-completion works very effective in the projectional editor, the need in the `using` directive does not emerge the same strong as in plain text editors. Long names of namespaces are appended to the end of method defintions for classes, to be exact about the namespace, but the declarations are kept looking short, with only the last namespace shown, Figure 4.1.

```
namespace MyNamespace {
  namespace MyInnerNamespace {
    class A /copyable and assignable/ {
      public:
        void DoSomething()
    }
    void A::DoSomething() from MyNamespace::MyInnerNamespace::A {

    }
  }
}
```

Figure 4.1: Example of Nested Namespaces

In general, the generator is responsible in generating correct namespace resolution, and the projectional C++ programmer can benefit from short and clear namespace presentations, which are also quite handy to input.

4.1.3 Memory Allocation

In the C programming language memory (de-)allocation on the heap is happening via calls to the dedicated library functions. C++ instead introduces separate language construction for memory allocation and de-allocation.

The `new` and `delete` together with their array versions are simply an expression and a statement correspondingly, which perform the required operations.

Creating them is relatively easily possible by inheriting from the **Expression** and **Statement** *concepts*, and introducing the corresponding typing for the `new` expression, as described in 3.2.1.

When a segment of memory is allocated as an array, it has to be de-allocated as an array either. As it is stated in the C++ standard ([16]) the behavior of simple de-allocation of an

memory segment, allocated as an array, results into undefined behavior. A simple check is introduced in order to control this, see the Section 4.5.6.

4.2 C++ Object-Oriented Programming

The C++ programming language is a multi-paradigm programming language. The ability to support the object-oriented programming, is incorporated via classes support.

A class represents a new type in the C++ programming language. Each class may have data in the form of fields, and behavior in the form of methods. Two types of methods are special for C++ - constructors and destructors, they have special meaning and syntax.

Encapsulation is enabled via governing access permissions to fields and methods of a class. The access control is governed with the creation of `public`, `protected` and `private` class sections.

Inheritance is implemented in C++ via allowing for each class to have one, or even many base classes. Inheritance from a base class is performed under a certain access control modifier. There is no pure notion of an interface, but rather abstract classes are introduced.

Polymorphism is implemented via pointer-to-class type compatibility over inheritance-connected classes.

The implementation of these C++ features in a projectional editor environment is discussed in the following sections.

4.2.1 Class Declaration and Copying

Visibility Sections

Instead of declaring visibility type for individual class members (C#, Java), visibility sections are created in C++.

The sections can be opened with a string `private:`, `protected:` or `public:` within a class declaration, and closed when another section is opened or when the class declaration ends. This allows the user to open and close the same section multiple times and declare sections without any particular order. This can be errorprone and confusive.

Various coding guidelines ([23], [24]) exist to enforce some restrictions on the visibility sections.

In particular, the sections are allowed to be opened only once. This ensures, the reader of the code will see the interface of the class (public section) in one place, "contents" of the class (private section) in one place, and opportunities to access members in an inheriting class (protected section) in one place, without the need to search through the whole class declaration.

Another typical requirement in coding standards, is the order of the sections. Usually the public section is required to be first, for the class users to see immediately the (public) interface, the class provides.

```
class WebPage {
  public:
    explicit WebPage() (constructor)
    WebPage(const WebPage& original) (copy constructor)
    boolean loadFromFile(string path)
    string getHtml()
  private:
    int32 visits
  protected:
    string html
}
```

Figure 4.2: Sample class type declaration

The Figure 4.2, shows an example class declaration implemented in the projectional editor. The **Class** *concept* has the visibility sections as children. Each section is given a separate role (see 3.1.1) and can appear 0 or 1 times. The editor for the **Class** *concept* orders the visibility sections so, that the public section always comes first, if present, followed by the private and protected sections.

The creation of a section is made with the use of *intentions*. The user uses *Alt+Enter* combination on the class declaration to create visibility sections. It should be more practical and fast for the user, compared to typing the keyword, colon and indenting the result.

A question arises on how to support another way to represent a class, so that it will reflect requirements from a different coding standard. And as a way to resolve it a definition of another editor for a class concept can be offered, Unfortunately, the current version of *JetBrains MPS* 2.5 does not support a definition of multiple editors for the same concepts. This limitation however is addressed in the newer 3.0 version, which was not tested completely with *mbeddr* on the time this work was made.

Constructors

Constructors are special methods of a class, used to construct the class instances.

Constructors have special syntax and no return type, being similar to class methods. Additional value, however, the constructors gain, when participating in type transformations. Namely, when a constructor of a class `B` exist from a type `T`, instances of the type `T` can be used whenever the `B` class instance is required. The constructor will be *implicitly* called and a temporary object of class `B` is going to be created as a mediator.

Thus constructors extend the type system of the C++ language, adding conversion rules to it. Since this type extension can not be easily observed, it is highly possible to get various *run-time* errors or unexpected behavior.

Listing 4.2: Example of an Implicit Constructor Error

```
#include<iostream>
using namespace std;

/*
```

```
 * API definition
 */

class Circle{
private:
  int r;
public:
  Circle(int radius){
    r = radius;
  }

  float getPerimeter(){
    return 2*3.14*r;
  }
};

void print(Circle c) {
  cout << "The perimeter is: " << c.getPerimeter();
}

/*
 * Use case by user of the API
 */

int main(){
  print(5);  //Prints "The perimeter is: 31.4"
  return 0;
}
```

The Listing 4.2 demonstrates a simplified use case where the function `print()` is invoked on `int` without any compiler error, and the resulting behavior is unexpected.

To avoid similar situations, it is possible to deprecate participation of a constructor in type conversions, adding a word `explicit` to the constructor declaration.

The described problematic motivated the following decision. When a new constructor is created with one argument, it is by default declared to be explicit, the user must intentionally change it to get the type conversion behavior. Such behavior is safer by default. This is an example improvement to classical C++ which makes the code understood better by novices and safer for them.

Copying

In the C++ programming language the programmer controls memory allocation fully on his/her own. This affects the way of copying fot the class instances. In C++ a programmer should define two methods for a class: a copy constructor and an assignment operator. These two methods work when assignment like `a = b` happens, when instances of a class

are passed by value to a function, when one object is initialized with a value of another object and so on.

C++ serves here sometimes dangerously generating default copy constructor and assignment operator, which by default represent a bitwise cloning of an object. This can lead to problems. For example, if a pointer value is getting copied bitwise in a second instance and is deallocated twice in destructors.

Listing 4.3: Need in a Custom Copy constructor

```cpp
class Resource{
  int* r;
public:
  Resource(){
    r = new int(0);
  }
  ~Resource(){
    delete r;
  }
};

int main(){
  Resource a;
  Resource b = a; // b.r is the same address as a.r
  return 0;
}
```

The Listing 4.3 demonstrates a program which crashes upon execution as destructors of `a` and `b` are deallocating memory with the same address, after default copy-constructor copies the address from `a` into `b`.

To avoid the described problem, the programmer has to either define a proper copy-constructor or forbid copying of the objects for the class. The same applies to the assignment operator. Many standards require the two functions to be implemented in sync, i.e. implementing the same semantic behavior, [25]. This can be performed in an elegant way by implementing the assignment operator first and reusing it in the copy-constructor.

When not providing the copying behavior it comes logical to disable also the assignment behavior. This is done by a trick of declaring the corresponding functions in the private ares, without implementing them (as they never get called). To visually explain such design and make it handier to implement, specialized known macros exist in various libraries, for example `DISALLOW_COPY_AND_ASSIGN` or `Q_DISABLE_COPY`, [24], [26]. An alternative approach is the use of `boost::noncopyable` from the boost library, [27].

The use of macros in C++ appears often in similar cases, in order to perform some language-engineering tasks to add the missing features to the language (copying or assignment deprecation in this case). Macros bear pure textual nature, and are processed by the pre-processor. Some negative effects may come out: need to preprocess reduces the speed of compilation and hides the resulting code from a programmer. Macros lead

to error prone programming, as no type checks are possible. And macros make code less analyzable by automatic analyzers.

```
class A /copyable and assignable/ {
  public:
    explicit A() (constructor)
    A& operator = (const A& originial ) (makes class assignable)
    A(const A& originial) (copy constructor)
    int16 getX()
  private:
    int16 x
}
```

Figure 4.3: Hinting about Copyable and Assignable Class Properties

The projectional editor allows for another solution, different from macros, or the use of a new library. In order to provide some support for the programmer regarding the copying issue, the *Projectional C++* hints on the class declaration its assignable and copyable properties, Figure 4.3, and generates by default the declarations of copy-constructor and assignment operator declarations, when the class is created.

The copy constructor and the assignment operator are recognized by the *Projectional C++*. Two *intentions* are provided on the **Class** *concept* to forbid or allow copying. The forbidding *intention* imitates the macros mentioned above, but displaying and explaining the implementation to the userFigure 4.4. The implementation of the intention consists of moving the declarations of two functions to the private section of the class. The allowing *intention* moves the declaration back to the public section, or creates them. The check for implementation provided for a method flags the two declarations appropriately, 4.5.4, to finish the process.

```
class A /neither copyable nor assignable/ {
  public:
    explicit A() (constructor)
    int16 getX()
  private:
    int16 x
    A& operator = (const A& o ) (makes class not assignable)
    A(const A& o) (makes class not copyable)
}
```

Figure 4.4: Class Made not Copyable by the *Forbid Copying* Intention

The *JetBrains MPS* supports the *Projectional C++* implementation by providing read-only model accesses, special parts of the editor concept, 3.1.2, by which the hinting is implemented. The *intentions* allowed manipulations on the **Class** *concept*, which made it possible to automate the allowing or forbidding of copying/assignment, making the implementation clear to the user, without the use of macros or libraries.

The whole work the programmer needs to perform to forbid copying and assignment contains of a call to an *intention*, one key-stroke. There is no need to include a header file with macros, and look up the documentation for them, using them in a right way afterward

(symmetric macro requires exactly one parameter - the class name, and it has to go in the private class section). The boost library, [27], providing functionality to disable copying is often considered to be to heavy-weight to include, when it goes about little tasks, like the one described in this section.

4.2.2 Encapsulation and Inheritance

Encapsulation and inheritance are considered here together, because from the language-engineering point of view, they just decide the access to class members. In other words, the *Projectional C++* implementation has to track encapsulation and inheritance related definitions and provide access to the class members accordingly.

Various Cases of Access Control

In the C++ programming language a number of ways to govern access control to class members exists. Before discussing the implementation of them in a projectional C++, I briefly review them with an example.

All members, a class has, are either declared in the class, or inherited from its base classes.

The members can be accessed in a number of different locations in the code, which differ by the level of access they have to the class members. Among these locations are the class methods, friend functions of the class, and external to the class code.

Each member can be declared with a certain visibility/access type, and the inheritance of the base class can happen with one of the three inheritance modifiers.

```
class A /copyable and assignable/ {
  public:
    int8 valAPublic
  private:
    int8 valAPrivate
  protected:
    int8 valAProtected
  friends:
    friend compare (boolean compare(const A& a1, const A& a2))
}

class B : public A /copyable and assignable/ {
  public:
    B(const B& originial) (copy constructor)
}
```

Figure 4.5: Declaration of two classes with a friend function

The Figure 4.5 shows a declaration of two classes. The class A has all three public, protected and private fields. A function compare() is declared to be a friend function of the class A.

This example demonstrate how showing only the central information, friend keyword and the friend function name, and just hinting on the details (the complete friend signature), can make the syntax more appealing. This is generalized later in the Section 5.2.

The visibility plays no role for the friend function declarations themselves. That is why a decision was made to create a special section for friend declarations, called `friends`. This section name is not generated anyhow in the resulting C++ text. This allows for all the friend functions to appear in one place, and be easily observable.

The class `B` in the Figure 4.5 is inheriting publicly from the class `A`, which means, that public members of `A` remain being public in `B`. The class `B` declares a copy constructor.

Such declaration can be utilized as shown in the figure Figure 4.6.

```
B::B(const B& originial)   from B {
  this->valAPublic = originial.valAPublic;
  this->valAProtected = originial.valAProtected;
  this->valAPrivate;
}

boolean compare(const A& a1, const A& a2) {
  return a1.valAPrivate >= a2.valAPrivate;
} compare (function)

void printOut(B b) {
  cout <<  b.valAPublic;
  cout <<  b.valAPrivate;
  cout <<  b.valAProtected;
} printOut (function)
```

Figure 4.6: Visibility resolution

In the copy-constructor the visibility resolution happens after `this` pointer and after the `original` object. Arrow expression and dot expression are used for this. The first and second lines are making use of public and protected fields of the base class `A`. The use of the private field is however not possible, since private fields of a class are only accessible to methods of the same class and friend functions.

It is even not possible to input the not-allowed member, as the projectional editor does not bind the text to anything, and it remains red. This means, no node in the *AST* is created and the code is in uncomplete erroneous state.

The `compare()` function is declared in advance (Figure 4.5) to be a friend function of the class `A`. Thus, it is not a problem for this function to access even private fields of `A`, for comparison purposes in this example case.

The function `printOut()` is not related anyhow to classes declared. Thus, it represent "external" for the class `B` code. Only public members are accessible, but not protected or private. The attempt to input them, simply fails, they are highlighted red, and are not bound to anything.

In this way the *Projectional C++* gives for the programmer only a chance to input correct from the encapsulation point of view constructions. As the members, accessible in each place of code, are provided by the *Projectional C++,* instead of typing the member name, the programmer usually will have a choice from a short drop-down list of options.

Expressions to Address Class Members

Members are usually accessed relatively to some object. The object can be designated as an expression of type class or a pointer to class, in particular, `this` expression. The resulting access represents nothing else, but an expression itself.

One of the greatest ideas in *JetBrains MPS* to allow extensibility is *concept* inheritance, 2.2.2, 3.1.1. Once a need to create a new concept arises, serving as a concept known before, the new concept has to inherit from the existing one, and this is almost all what has to be done. Thus inheriting the **OoDotOrArrowExpression** *concept* from the *Expression concept*, we get the ability to use the expression, designed for member access, wherever an expression in general can be used.

The abstract *concept* **OoDotOrArrowExpression** serves as a parent for **OoDotExpression** and **OoArrowExpression**. The commonality between the two, is that an object is accessed in the left part, and a member is selected in the right part, as well as the way to decide the access to members. The access is defined then, which left part is going to be possible in such expressions.

Within the class methods it is also possible in C++ to address class members as local variables. In the projectional implementation described, it is not possible. Instead, `this` expression has to be used. It makes typing a little bit slower, but allows to easily distinguish between members of the class and other variables or functions.

4.2.3 Polymorphism

There are several ways to achieve polymorphic behavior in C++. The purists of the language differentiate between the polymorphism based on virtual functions or based on templates. More general opinion can include in the notion of the polymorphism also functions overloading and operator overloading, also called *ad hoc* polymorphism, polymorphism for concrete one purpose. Occasionally various operations with `void*` type are also classified as a polymorphic programming.

In this section I am writing only about the class-related virtual functions polymorphism, and the way it is implemented in *Projectional C++*.

Virtual Functions Polymorphism in C++

Starting the description of the projectional implementation, I find it good to spend some time describing the way, the polymorphism is implemented in the C++ programming language itself, pointing out the places, where it could be improved.

Dislike many other popular object-oriented programming languages in the C++ there is no pure notion of an interface. Instead, a base class, its public part, is used as an interface declaration for the descendants. Functions designated to be a part of the interface must be declared `virtual` and they can be overloaded in the subclasses.

The virtual functions in the base-interface class can be implemented as well, providing some "default" common enough behavior. Otherwise they are left *pure virtual*, meaning that no implementation is provided and the pointer to the function in the table of pointers to virtual functions is zeroed. The syntax for *pure virtual* functions is rather not obvious[1]

---

[1]especially when not knowing about the zero pointer value semantics

and a bit cumbersome requiring to type one reserved word, one punctuation sign and one digit to express a simple fact of pure virtuality or, simply, absence of implementation, Listing 4.4.

Listing 4.4: Pure Virtual Function Syntax Example

```
class Animal
{
public:
    virtual void voice() = 0;
};
```

The approach of classes with virtual functions as interfaces is more flexible compared to languages with the notion of an interface is directly introduced. In C++ it is possible to create partially implemented base classes, what can not be done when implementing and interface in Java or C#, where the approach is "all or none" regarding the implementation of interface functions. The presence of interfaces in the language can be though considered a more clean way to program.

In order to implement a declared in a base class function a descendant must declare and implement a virtual function, matching the full (including a return type) signature of a declared in the base class function, Listing 4.6. The connection to the "interface" function declaration stays subtle however. It is not immediately clear, whether the declared new function in the descendant is an override of an existing in the base class function or an independent declaration of an entirely new function in the descendant class.

This knowledge affects the changing process greatly, as the override should change from the interface, together with all the implementations. The absence of a clear, explicit override syntax I call here an "*override syntax absece*".

Whenever a class and all of its ancestors do not provide an implementation of a certain virtual function, created as a pure virtual in the declaring class, the class is called an *abstract* class. It is not possible to construct instances of an *abstract* class. C++ however does not have any special syntax to explicitly decalre a class *abstract*, Listing 4.4. The programmer usually has to be aware (from documentation, implementation, or, the worst case, compilation errors) whether a given class is *abstract*. I will call this phenomenon an "*abstract class syntax absence*".

Overriding a function is an active action of the programmer, and it is initiated by the programmer. I.e. the programmer want to state, that the new funciton is designed to override an exiting one, and which is the overriden function. The abstract property of a class, oppositely, is not a quality a programmer directly gives to a class. It can be rather deduced from the analysis on the base classes automatically, by editor. So in this case no actions are needed from the programmer side. Because of this two similar absence of syntax phenomena are resolved or or have to be improvved on differently from the *Projectional C++* point of view.

In order to use an interface, declared in some class, the using code has to get a pointer to an instance of any inheriting the interface descendant class instance. Thus typing system has to allow a pointer to the descendant to be treated in the way as a pointer to the base

class would be. The same should hold, normally, for the reference types, but it is not used very often on practice, and is omitted in the implementation.

As this typing rule represents the core of polymorphic behavior, I will start from it, describing further the polymorphism in the *Projectional C++* implementation.

Pointer to Class Special Typing

The problem solved here is enabling the usage of pointers to descendants instead of pointers to ancestors, Listing 4.5.

Listing 4.5: Example Usage of a Pointer to Descendant Class

```cpp
class Shape {};

class Circle: public Shape {};

void draw(Shape* shape){
  // ...
}

int main(){

  Circle* c = new Circle();

  // Call with a Circle* parameter instead of the declared Shape*
  draw(c);

  return 0;
}
```

The **PointerType** *concept* is implemented in the *pointers* language in *mbeddr*. Following the goal of non-invasive changes to *mbeddr* the *Projectional C++* implementation needs to add the typing rules for pointers *to classes* without changing the *pointers mbeddr* language, where the typing system for pointer type is defined.

In the case when a type of a pointer to a base class is expected, a pointer to a subclass should also be accepted, like in the Listing 4.5.

The type system of the pointer language will try to check the compatibility of the two pointer types. It will fail to do so, as the *mbeddr* languages are not aware of the *Projectional C++* extensions[2] by design.

When none of the existing type system rules can resolve the given situation, the so-called replacement rule can be invoked in *JetBrains MPS*. The *Projectional C++* provides such a replacement rule for pointer type, which checks, whether a class pointed to, is a passing descendant of the class required by an expression, where the pointer was used.

In the Section 5.3 I discuss more on the approach taken here, its limitations and the potential ways to overcome them.

---

[2] **Class** *concept* in this case

Overriding a Virtual Function

The Listing 4.6 demonstrates, how overriding of a virtual function happens in C++. The function `getArea()` is initially defined in the class `Shape` and is then overridden in the class `Circle`. The Listing 4.6 demonstrates as well the *override syntax absece* in C++.

Listing 4.6: Example of an Overridden Function - Text Code

```cpp
class Shape {
  public:
    Shape();
    virtual double getArea();
};

class Circle : public Shape {
  public:
    Circle(double r);
    virtual double getArea();
  private:
    double mRadius;
};
```

In the Figure 4.7 the same example is demonstrated, but in the *Projectional C++* implementation. Indeed, the Listing 4.6 is the generation result (part) from the demonstrated projectional sample in the Figure 4.7.

When a method declaration is created, it can be set to be an override of a method in a base class. Right after an empty method declaration is created, it can be set to be an override, so that the only thing which stays, is to pick a method from a base class to be overridden. The override is automatically named, the parameters and the return type are set accordingly, the `virtual` property is immediately set. This is yet another work saver for the programmer.

After the override has been linked to the overridden method, the projectional editor checks, if the override full signature stays precisely the same as the one of the overridden method. Thus if the latter changes, it is going to be indicatted in the former. The overridden method is shown next to the override declaration Figure 4.7, which compensates on the *override syntax absece*.

```
class Shape /copyable and assignable/ {
  public:
    Shape() (constructor)
    virtual double getArea()
}

class Circle : public Shape /copyable and assignable/ {
  public:
    Circle(double r) (constructor)
    virtual double getArea() overrides Shape::getArea()
  private:
    double mRadius
}
```

Figure 4.7: Example of an Overridden Function - Projection

The additional, out of C++, visual syntax in the *Projectional C++*, should not confuse the reader. It is only present in the projectional editor, and, when an *AST* is generated into a text code, the regular C++ syntax is achieved. However in this case the *AST* stores *more* than needed to generate and compile the C++ code.

This subsection is one of the examples, that storing more information can be useful, it is generalized in the Section 5.2.

Pure Virtual Functions

In the example above, Listing 4.6, one improvement to the code can be made. As the `getArea()` for a random shape can not be determined, it makes sense to make the `getArea()` function *pure virtual*. As said before, a *pure virtual* function has no implementation in the declaring class, and serves only the overriding purpose. Semantically in C++ it sets the pointer in `vptr` to 0 and thus has reflecting it syntax, see the similar example in the Listing 4.4.

This syntax can be seen as not obvious, as it reflects more the under-the-hood implementation of the mechanism, rather then the original programmer intention to built a basis for an overrides chain, while taking advantage of polymorphism.

Additionally, as discussed above, declaring a *pure virtual* function requires a significant amount of the syntactical overhead.

These were the disadvantages of the C++ programming language the *Projectional C++* tries to improve on.

In the *Projectional C++*implementation, one *intention* is reserved to make a function *pure virtual*. The *intention* automatically sets the needed pure virtual and virtual flags of the function declaration, and the projection changes. That is why out of the statement `virtual double getArea() = 0;` the programmer has to input only the name `getArea`, pick the type `double`, and toggle the pure virtual intention for the declaration in the `Shape` class.

The result of the intention work in projection can be seen in the Figure 4.8.

<div align="center">

pure virtual double getArea() **= 0**

</div>

<div align="center">

Figure 4.8: Example of a Pure Virtual Method - Projection

</div>

The word `pure` is added by the projectional editor. This makes the reading of the code easy and natural. The `= 0` part is preserved for the C++ programmer with habbits, used to the original C++ syntax. And, as in many other cases, the semicolon is omitted as it is nothing but syntactic help to the compiler, which does not have to appear in the projection.

This example demostrates, how the lower level syntax can be made more readable. The general principle on it is formulated in the Section 5.2.

Abstract Classes

If any of the *pure virtual* functions, in the inheritance chain leading to a class, is not implemented by this chain, or inside the class itself, the class is called an *abstract* class. It is not possible to construct an instance of an *abstract* class, as such classes have not implemented methods.

The programmer has to know which classes are *abstract*, and are intended for inheritance and further implementation only. One of the reason for this, is to not to try instantiation of the abstract class. Another reason is to exaclty identify *abstract* classes, designed to serve as extension points.

Above I discuss the *abstract class syntax absence* in C++. It leads to the need for the programmer to determine somehow him/herself if a given class is *abstract*, the source code representation of a class does not give any information on it, unless some naming conventions require *abstract* classes to be named specially.

And improvement to this situation whould be a behavior, when an editor can perform an analysis and determine if a class is *abstract*, hinting on it to the user. In the case of the projectional editor, this analysis is especially computationally efficient[3], as the *AST* is readily available, and the quick analysis can be performed by a simple recursive algorithm on the inheritance chains.

After an *abstract* class has been determined, it is possible to modify the editor representation for it, and show that it is abstract, Figure 4.9.

In this example a typical class hierarchy is created to support user interface programming. A user of this *API*, when searching for a button, could try using the `Button` class, which is designed to be *abstract*, and serve as a base for the further implementations, e.g. `PushButton` in the example, or check boxes, radio buttons and similar, later.

---

[3]in comparison to textual editors, with the need to parse, see the Section 5.1 on comparison to the textual approach, and Section 5.4 on complexity of analyses

```
abstract class Widget /copyable and assignable/ {
  public:
    explicit Widget(Widget* parent) (constructor)
    pure virtual Size getDimensions()  = 0
}

abstract class Button : public Widget /copyable and assignable/ {
  public:
    Button() (constructor)
    pure virtual boolean isPressed()  = 0
}

class PushButton : public Button /copyable and assignable/ {
  public:
    PushButton() (constructor)
    virtual Size getDimensions() overrides Widget::getDimensions()
    virtual boolean isPressed() overrides Button::isPressed()
}
```

Figure 4.9: Determining Abstract Classes

The projectional editor, however, checks on the fly, if a certain class is abstract, adding a special `abstract` word in front of its declaration. This makes the reading easier, and allows for quicker understanding of the code.

This example demostrates a general principle, see the Section 5.2, of a advised practice to perform quick analyses and inform the user.

Additionally, the creation and usage of abstract class instances is checked, and forbidden by type analysis. This is described separately in the Section 4.5.

4.3 Operator Overloading - to be written

4.4 Templates - to be written

4.5 Advanced Editor Functionality

As the projectional editor works directly with an *AST*, it is possible to provide some programming on the *AST* to improve the user experience. I call this additional programming as an advanced editor functionality, and discuss it in this section.

4.5.1 Renaming Refactoring

Directly from the nature of the projectional editor, without any additional effort, comes a feature to perform the renaming refactorings.

If a node of an *AST* gets to be referenced somewhere else, it is referenced by the use of its unique internal identifier. The name of the node is a property of the node, which is not playing any role in referencing the node from somewhere else. Thus renaming, dislike

the way it is performed in text editors, does not involve replacing the name all around the code. Instead, just the name property of a node is changed.

The renaming refactoring comes out of the box, without any additional efforts, thanks to the nature of the projectional editing.

As an example - renaming a class or a method would mean just changing its name where it is declared first. No search for usages and multiple replacements are going to be involved.

As a disadvantage here one can think of moving a code from one place to another. For example, moving a method from one class to another. The member fields of the source class, even if present in the destination class, will not immediately fit into the moved code, as it is still referencing not available anymore identifiers of the source class. This effect is also mentioned in the Section 5.1.

4.5.2 Getter and Setter Generation

In order to provide an access to encapsulated class properties, often expressed as member fields in C++, two access functions are usually defined, known as a getter and a setter. The getter is used to read the property, and the setter is used to set the property to a new value, after checking the validity of the new value.

The job of declaring and prototyping the two functions can be automated with a *JetBrains MPS intention*, Figure 4.10.
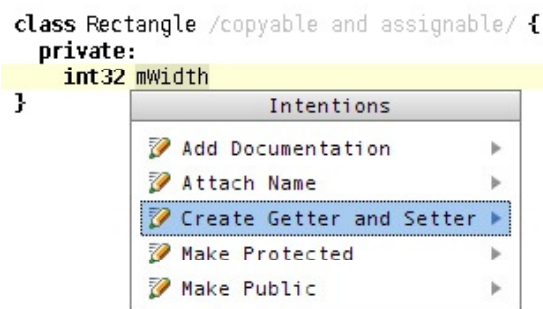

Figure 4.10: Calling the Generation of Getter and Setter

The result of the *intention* work is, as expected, two methods declared and prototyped, Figure 4.11.

```
class Rectangle /copyable and assignable/ {
  public:
    int32 getWidth()
    boolean setWidth(const int32 theWidth)
  private:
    int32 mWidth
}

boolean Rectangle::setWidth(const int32 theWidth) from Rectangle {
  this->mWidth = theWidth;
  return true;
}

int32 Rectangle::getWidth() from Rectangle {
  return this->mWidth;
}
```

Figure 4.11: Getter and Setter Generated

The way the editor names the getter and setter can be controlled through the **NamingConventions** *concept*, which is discussed in 4.5.3.

The getter and setter are declared in the public section of the class, which is automatically created, if not found there already.

The getter is rather simple, it just returns the value of the member field. The setter is somewhat more complicated.

Firstly, it is designed to return a `bool` value by default. This is made to remind the programmer, to include some checking of the value and return `false`, if the check locates a wrong value passed.

Secondly, the parameter of the setter is typed appropriately. As C++ by itself imply the performance maximization, the type of the parameter for the setter depends on the member field type. And when the type represents a composite structure, like a class, it is passed by a constant reference, instead of value, Figure 4.12.

```
boolean Widget::setSurface(const Rectangle& theSurface)
  this->mSurface = theSurface;
  return true;
}
```

Figure 4.12: Setter Works with a Constant Reference for Classes

Passing a composite parameter as a value involves an overhead of allocating the necessary memory and then copying the contents in the newly allocated instance. To access the parameter in both cases pointer arithmetic is still going to follow.

4.5.3 Naming Conventions

In the C++ development projects some code writing conventions are usually agreed upon. For example, in Google coding style guide for C++, [24], it is stated that all member fields must end up with "_" sign.

For the consistency and uniformity purposes each project has to have some agreements on the way the code is composed. They can include code formatting rules and naming rules.

To some extent the formatting conventions are fixed by the way the projectional editor shows the *AST*, see, for example, 4.2.1.

The naming rules, however, should be additionally controlled. The naming rules, accepted in a project, I call naming conventions here.

In order to perform the naming conventions controll a new *concept* has been introduced, called **CppNaming Conventions**, Figure 4.13. It is created once per *JetBrains MPS model*.

```
C++ Naming Conventions
  Member prefix:  m
  Getter prefix:  get
  Setter prefix:  set
  Setter argument prefix: | the
```

Figure 4.13: C++ Naming Conventions Concept

In the **CppNamingConventions** one can give a standard prefixes for getters and setters, which are used during the code generation, see 4.5.2. The argument prefix is used in the setter generation, to name the setter argument.

The member prefix is used on the member fields to check their naming. It possible to change the prefix and the editor will control each of the member field names to comply.

Whenever a field is not named in a proper way, an error *intention* is available, see 3.1.8, which automatically renames the field, Figure 4.14.
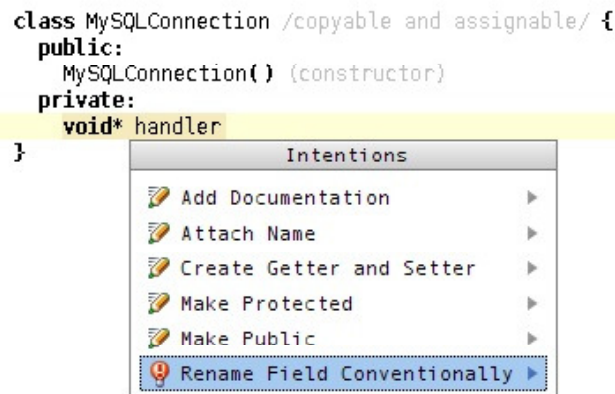

Figure 4.14: Intension to Rename a Field - Naming Conventions

The **CppNamingConventions** *concept* was created to demonstrate some new possibilities, which can be added to the projectional C++. In this case the naming conventions can be seen as a part of project configuration, in a very common sense. This is generalized in the Section 5.2.

Few remarks are needed to explain potential changes to the **CppNamingConventions** *concept* and its difference from the similar functionality taken in the regular text editors.

First, the marking of special variables or object, keywords or type names, can be performed by changing editors for them. The special naming will not be needed then. Such way would mean however no opportunity to introduce a project-dependent marking, as it starts to be common for all projects, being a part of the projectional C++ implementation.

Second, the naming checks can be generalized to use compliance to regular expressions, instead of just prefixes. As an advantage comes the ability to adopt a much broader set of different naming conventions. Disadvantageous is the increase of the computational load in the projectional editor, this is discussed also in the Section 5.4.

Third, the **CppNamingConventions** in p*Projectional C++* is just a *concept* in a *JetBrains MPS model*. It is no different from the **CppImplementationModule** or the **BuildConfiguration** *concepts*. Dislike many conventional editors, which store such settings as a workspace, project or *IDE* configuration, the *Projectional C++* makes it a part of the program code itself. It is advantageous, since all the programmers in a team have to follow the same naming conventions set up *once* in a project on the start phase, and no environment (re-)configuration is needed to work on the project by any of the joining developers. This principle is generalized in the Section 5.2. The projectional approach is also discussed and compared to the conventional one in the Section 5.1.

And last, the naming conventions can of-course be extended to incorporate class names, function names, additional naming for methods, like "is"- prefix for the boolean return type.

4.5.4 Method Implemented Check

When using the C++ programming language it is possible to declare a class in one file, and then implement its methods in other, potentially several, files. As there is no clear concept of a module in the language itself (more on this in the Section 4.1.2) usually different coding standards have to improve on it, requiring, for example in the Qt Project [28], one class to be declared in a the-same-named header file, and the implementation for it to be in one similar named .cpp file.

As the language itself does not argue about the place for methods, *IDEs* usually do not check if the method is implemented. Neither do compilers. And only at the linking stage it can appear, that the linker is not able to locate the method. The error message from the linker can be rather obscure, as the linker operates on a much lover abstraction level, rather than the programmer using the C++ language.

After introducing the new module *concept* as in the Section 4.1.2, it is possible to check each method and find out if they all are implemented on the very early coding stage.

A check is introduced in the *Projectional C++*, which respects inheritance chains, with overloaded functions and *pure virtual* functions, and creates a warning for the programmer hinting that a certain method has not been implemented. Together with the class declaration, generating the assignment operator declaration and a copy constructor, the method implemented check makes the programmer to fully define the class together with the most important methods, affecting the memory operations behind the class instances life-cycle.

4.5.5 Abstract Class Construction Check

In this work I have already defined the notion of *abstract* classes in C++ and some related to them questions in 4.2.3 and in 4.2.3. Mainly the *abstract class syntax absence* and the problem of determining *abstract* classes were discussed.

```
abstract class MyObject /copyable and assignable/ {
  public:
    pure virtual string getName() = 0
}
                   Error: Can not be an abstract class
void greet(MyObject object) {

} greet (function)
```
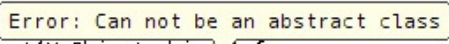
Figure 4.15: The Abstract Class ConstructREal Zaj Zion Check Signaling an Error

An opened question which stayed, however, is providing some help to the programmer in the usage of such classes. Namely, any instantiation of an *abstract* class has to be forbidden. The *abstract* class as a type, however, should stay, and the pointer type to it should not be anyhow affected by the fact that the class is *abstract*.

The decision was taken to check the places, where an instance of an *abstract* class can appear, and deprecate such situations. *JetBrains MPS* supports so-called non-type-system checks, which come useful in this situation. Any node on the *AST* can be analyzed, and if the error happens to be detected, there is a chance to associate it with the node and a textual error message, which will clarify the programmer, what exactly went wrong.

Among the places, where a class can be instantiated are variable declarations field declarations, method and function arguments. The return type of the function can imply an *abstract* class instantiation. For each of such places a check is performed, and if the class type is detected, it is checked against being *abstract*. Error message appears to indicate the problem, if found, Figure 4.15.

The procedure of determining if a class is *abstract*, can happen every time, when the check is performed. These are all variable declarations, function and method arguments and return types, and memory allocations. The amount of computational load can be significant. The question of this complexity is discussed in the Section 5.4.

4.5.6 Array Deallocation Check

Whenever some allocated on the heap data is about to be deallocated, in C++ two operators can be involved, `delete` or `delete[]`. The first one is dedicated to work with memory, allocated for a single object be he operator `new`, the second deallocates memory, allocated for an array of object by the operator `new[]`.

The C++ standard does no specify any concrete behavior for wrong deallocation of an array with an operator `delete` (without square brackets), describing

5 Evaluation

5.1 Comparison with Textual Approach

It is important to compare the *projectional approach* to build editors for languages to the well-adopted textual approach. To the opinion of author the *projectional approach* has a number of advantages, which could make it more popular in the future, as well as, naturally, some disadvantages.

First, in a projectional editor the programmer can potentially type less, as the editor is aware *completely* about all the possible constructions, dislike various code-completing helpers in the textual editors, which can refuse to work at all in some cases[1]. Thus the programmer may rely on the auto-completion much more, which can make the typing itself faster.

Second, as an *AST* is readily available in a projectional editor, there is no need to pre-parse the code in order to perform analysis on it to detect mistakes or for another purpose. In a textual editor before any analysis the program has to be parsed first. Parsing is a computationally expensive operation. Repeating it after each time a piece of code is modified puts a high load on the developer's machine and can slow it down significantly.

Third, in order to support refactoring an ultimate "understanding" of the code is required by the environment. It is not always possible, however, especially this problem is well-known with complicated languages like the C++ programming language. For example, one of the best C++ *IDE* Microsoft Visual C++, a problem of parsing complex C++ code and manipulating on it is well-known, and parsers behind it are constantly updated, to support more of the C++.

Fourth, is the flexibility to adopt new language constructions. As a projectional editor allows for language modularity, 2.2.2, it is possible to extend a language, without modifying an editor significantly, installing extensions for it or similar.

Fifth, different naming conventions and coding standards can be replaced by decorations and the way a projectional editor projects the *AST*. For example, a member variable, can be shown differently from a local variable in an method. This eliminates the need in many of the coding conventions. Naming conventions are, however, usual to programmers, and their are implemented in a shape of analysis in the *Projectional C++*, 4.5.3.

Nowadays, in *JetBrains MPS* the approach of generating a text code is taken, before the projected *AST* could be further processed. However, taken into account the fact, that an *AST* in a projectional editor represents the same, or even more, information as an *AST* resulting after parsing the text code generated, in future it is possible, to process the projected *AST* itself, without generating a text code at all. For example, produce object code from it directly. Thus, in future, the compilation/processing of code from a projectional

---

[1] when a program is in unparsable state, or when there is no source code for a library, sometimes, when a code base is to big to be indexed

editor can be made much faster and effective when compared to the textual approach, where parsing takes place.

A special problem with the projectional approach is moving the code around. As all nodes of the *AST* get referenced by the use of their internal identifiers, and their names do not participate in it, after a piece of code has been moved to another location on the *AST*, a special processes of binding the nodes has to be executed. For example, if there was a variable x in the code defined before the moved snippet, and in the new location for the snippet another variable with the same name x is defined, the new x will not get referenced by the moved code snippet, as it references the old one (not available anymore) by its internal identifier. In *JetBrains MPS* the manual process of rebinding is always required, i.e. the programmer will have to go over all nodes, which reference the old context, and input them again, to connect to the new context. In textual editors, there is no such problem, but another one exist - the moved code snippet can change its semantics in a wrong way after being associated with a new context, and not checked entirely. For example, a reference to a variable can start to reference some another, locally defined, more close to the new code snippet location, variable with the same name as one, more global, used before.

Another very specific issue relevant to projectional editors is the format to store the projectional code and the version control for it. In *JetBrains MPS* an *AST* gets serialized as XML, and is stored in a file. The XML resulted, is, generally speaking, not a human-readable code, despite the XML nature. The line-by line merging as employed in regular text-oriented version control systems (CVS, SVN, Git, Mercurial) does not apply, breaking the XML, or asking the user to merge, presenting with unreadable XML code. The approach taken in *JetBrains MPS* is providing a special merge driver, which handles merging for the projectional code in a proper way. The driver is not perfect however, still requiring the user to finish the job manually sometimes.

The new languages are developed themselves using a special defining projectional languages in *JetBrains MPS*. So the evolution of a language under development, and its version control is also an issue. Each new incremental iteration of a language in *JetBrains MPS* gets internally an increment in the language version number. When a second language is referencing the modified one, the version number is taken into account. The update for the first language then requires an update to the second, using the first, language, as well as to all other languages, which use the updated one. If two updates happened at the same time for a language, then two *different* versions with the same version number appear, which presents a problem for the referencing the changed language languages. All this version control issues are still up to be thoroughly though of, and are not yet implemented well enough.

Additionally the question of language evolution and code in it is to be researched on. When a language defined in a projectional editor is modified, the old code may turn to be not matching the language description anymore. For example, nodes may stop satisfying constraints, or have a child/reference with a role, deleted in the new language version, or, vice versa, have a nothing in a place where the new version language requires some child, reference or property. This renders the code in a former language version incorrect in a newer one. The question of the code update to a new language version appears. For example, a script can be provided with each new language version, which updates the code to the version in question. This is not automated however in *JetBrains MPS*, so that the update process is seamless for the language user. This is another problem to work on

in the future. The problem described, of-course, is not relevant to the text code editors, as it is, since the language evolution there is rather untypical.

5.2 Generalized Principles of the Projectional Approach

In this chapter I formulate some of the general principles, which can be taken into account when designing new languages in the projectional editors, which are meant to represent, especially, the existing already text languages. The principles come out as generalizations of the practical experience, achieved during the work on this Master Thesis.

5.2.1 Targeting Semantics

When implementing a language for projectional editor, one should target semantics of it, rather than an existing syntax. For example, extensions can be provided, which raise the abstraction level to be closer to the application of the language. The code in the target text language is generated then from the higher level constructions.

Another place to think of targeting semantics is, where the target language constructions are low-level and full of compiler-helping syntax. These constructions can be cleaned out, helping the programmer to focus on their semantics, instead of typing and syntax.

5.2.2 Store More Information

The language in the projectional editor can, and often should, contain *more* information as it is needed to just generate a text in the target language. This information may be used to improve the generation results, or analysis. Example is the overridden method link in the override in the *Projectional C++*, 4.2.3.

The disadvantage of this way can be a problem to extend the information, taken from an importer of the native text language, 2.2.3.

5.2.3 Configuration as a Part of Source Code

Usually a project consists not only of the source code, but of some configuration for it, like naming conventions, generator configuration, build configuration, other specific to the project information.

It presents advantageous to store this configuration together with code as a part of it. This eliminates the need to separately configure an environment of each developer before the development process may start.

Usually editor preferences are not shared among users, like in Eclipse for example, stored in a individual for each machine workspace part. This brings a need to configure each development environment separately, and maintain the similarity of configuration.

This should exclude, however, the pure developer-machine-specific configuration.

5.2.4 Hide Redundant Syntax

Usually, textual languages contain a lot of syntax, which helps parsers of the language to process the code. In C-like languages these are semicolons, curly braces, braces and

brackets.

This syntax has nothing to do usually with the code semantics, so it could be not projected at all, without changing the meaning of the code. Decreasing the amount of projected symbols the code can be made more readable.

Formatting[2] can also be considered the redundant syntax, and addressed by projection.

5.2.5 Make Old Syntax Readable

Whenever a syntax of the target language happened to be not well readable by itself, a projectional editor can change it. The amount of punctuation can be lowered and some phenomena named in a human language.

As an example I am bringing here a *pure virtual* method declaration, 4.2.3.

5.2.6 Show the Core, Hint on Details

Not necessarily all the information represents the core meaning of the language construction. The most important information can be shown first, and the rest can be shown as a hint, especially when it can be figured out by the projectional editor automatically.

As an example we can consider a friend function declaration in the *Projectional C++*, 4.2.2.

5.2.7 Perform Analyses and Inform the User

Performing various analyses on the *AST* in the projectional editor, it is possible to improve the overall editing experience.

I suggest the logical division of analyses on two types: *informative analysis* and *preventive analysis*.

An *informative analysis* can be performed to find out more about the edited code. Various newly found properties can be shown as hints for the user. Abstract classes determination is an example of informative analysis, 4.2.3. Another example is class copying and assignment analysis, 4.2.1.

A *preventive analysis* can be performed to prevent programming mistakes. In general, this topic can go deep, and it is considered separately in, e.g., [7]. In this work as an example of such an analysis for C++ the detection of abstract class instantiation attempts can serve, 4.5.5.

When implementing analysis, one should mind the complexity of it, and take the decision, whether to implement it as a *self-running analysis*, or on as an *analysis on demand*, see the Section 5.4.

5.3 Projectional Language Extensibility

In general the extensibility and modularity are very important concepts for projectional editing, 2.2.2.

---

[2]it is important for some languages, like Python, and less relevant to C as syntax complication

In *JetBrains MPS* the extensibility consists of creating a new language and there some new *concepts*, which represent an extension. When creating *concepts* in *JetBrains MPS* one deals with several views of them, see Section 3.1. In this section I consider extensibility, and the most important, how well it is possible in *JetBrains MPS*, per view on the language.

5.3.1 Structure

This view is the same as the view on *concept* declarations.

Extensibility is implemented here by inheriting from the base *concept*. It is rather straightforward, since it is very similar to the object oriented programming languages concept of inheritance, with one difference, actually originated from the Java notion of interface, as there is one base *concept*, but several *interface concepts* which could be implemented.

5.3.2 Editor

It is hard to extend a language in terms of changing an editor concept (see 3.1.2) for an existing *concept* in *JetBrains MPS*. Namely, there are no direct ways to perform inheritance there.

So, if the way a *concept* is edited or displayed has to be changed, one should take work around ways to achieve the needed behavior.

One way is inheritance from the *concept*, and changing the editor concept for the descendant.

Another way, is mixing some interface calls into the editor, and overriding the methods in the inheriting concept. This way limits the editor to the way it was originally programmed is limited.

These both ways, however, do not change the way instances of the old *concept* are presented.

In the new 3.0 version of *JetBrains MPS* an ability to have several editors for one *concept* is present. It is up to the future research, on how it will change the editor extensibility.

5.3.3 Constraints

Constraints, similar to the editor, does not extend well in *JetBrains MPS*. Namely, there is no way to newly define the constraints for an existing *concept*. When creating a new concept, there is no way to reuse constraints from existing one, as they are not accessible.

As an example, when extending the *mbeddr* C to the *Projectional C++*, the naming of identifiers had to be changed, so that C++ keywords are accounted as well. They naming rules for identifiers are constrained in the **IIdentifierNamedConcept** in *mbeddr*. So in its constraints concept a limiting behavior is programmed. There is, however, no way to reuse it explicitly.

Another example would be the reference constraints, where a variable declaration would be forbidden to use an *abstract* class as a type. It is not possible to redefine those constraints, just adding a language, and not editing the existing one directly.

As a workaround two methods are proposed. First, the constraining behavior can be taken out into a separate code fragment in a Java-like class, the abstract factory pattern could be implemented, [8], and the polymorphic behavior achieved in this way. Second,

is to create a check, additionally constraining and marking as errors the places, where the new constraints have been violated.

The first workaround is work intensive to implement originally in the base language. The second workaround does not allow for extending the constraints, making them weaker, and lets the not allowed nodes in the code completion.

Additionally, it is noteworthy to mention, that it is possible to redefine a constraint for a certain role, when inheriting from a *concept*.

In the future, a research could be made, how *JetBrains MPS* could be improved to allow for easier constraints extensibility.

5.3.4 Behavior

Behavior is very similar to Java classes and their methods. It is easy to extend behavior of a *concept* in *JetBrains MPS* as all the object oriented methods for polymorphism can be involved in the very same way as for Java classes.

5.3.5 TextGen

TextGens are also very well extensible. When generating a text for a *concept*, the children are output with the textgen **append** command, without specifying, how exactly it should be done. Polymorphically the matching *TextGens* are invoked, and this extends the *TextGen* of the previously defined *concept*.

For example, when generating a text for a function, each statement of the body is generated to text. For each statement the *TextGen* of the statement itself is executed. Thus, if we add a new kind of statement, all we need is to define a *TextGen* for it, and it will be polymorphically invoked from the *TextGen* for the function. The text for the function will be generated correctly, without the need to change the function *TextGen* implementation anyhow.

5.3.6 Generators

Generators perform language transformations in *JetBrains MPS*, 3.1.7. As this work does not directly implement language transformations, it is left for future research, to investigate, how extensible the generators are.

5.3.7 Intentions

Intentions are described in 3.1.8.

There may be a need to modify an *intention* after extending the language. There is no direct way to do it in *JetBrains MPS*.

For example, there is explicit way to forbid an *intention* to work on a newly defined *concept*, no matter it is based on some *concept* for which the *intention* is defined.

As a workaround an injection of external polymorphic code from a Java-like class can be taken, similarly to the way it is described above for constraints, with the same practical outcome of a high programming intensity when creating the base language.

Alternatively, an *intention* can delegate its function to the *concept* under question, so that the whole manipulation is made by the target *concept* itself. This will lead, however in a necessity to define a general behavior in the base concept, which can be behaving wrong in one of the particular cases, or to the need to implement the polymorphic methods called in the *intention* in *each* of the descendant *concepts*.

5.3.8 Type System

Type System in *JetBrains MPS* is generally speaking another way of constraining. And exactly as in the situation with constraints there is no direct way to change the type system behavior, when extending an existing language with a new one. For example, when a new kind of expressions is added, which has to be typed differently, when used together with some existing expression type, and the existing expression ty[e system is determining the type, it is not possible to redefine this behavior or extend in naturally with some provided by *JetBrains MPS*tools.

The workarounds consist from injecting polymorphic behavior in the type system calls, or from using *JetBrains MPS*own workaround, like the replacement rule, as for example in 4.2.3.

Hopefully, in the future versions of *JetBrains MPS* this is going to be taken care of, and some ways to extend naturally, by the means of the typing system itself will be present.

5.3.9 Analyses

Analyses are also subjects to challenge the language extensibility. If an analysis is relying on the exact *concepts* to work, and is not assuming that a new language could extend those, a problem happens.

For example, *mbeddr* data flow analysis, can analyse a C++ code snippet, using some class `Circle`, to instantiate it `Circle c;` and produce an error, that the variable `c` has not been initialized, which is not true in the case of a class with a constructor. Either the analysis should delegate some features to *concept*, or employ some external polymorphic calls as possible workarounds. In the first case the analysis stops to be a separate programming from the language itself, as delegated analysis methods are found now in the *concept* behaviors. In the second case the development can get more complex initially, as the code should be written in the very beginning polymorphic and abstract enough to allow non-invasive modification in future.

Again, some specific support for analyses from *JetBrains MPS* itself could improve the situation, by providing some special means to develop extensible analyses.

5.3.10 Extensibility Overview

Here I summarize in a table the degree of extensibility in *JetBrains MPS*, provided for languages, per view on a language, and the quality of known workarounds.

View	Extensibility	Workarounds Quality
Structure	High	-
Editor	No	Poor
Constraints	Low	Good
Behavior	High	-
TextGen	High	-
Generators	-	-
Intentions	No	Medium
Type System	Low	Medium
Analyses	No	Medium

As an outcome it appears, that *JetBrains MPS* does not provide a high degree of extensibility, which would allow a pure modular language engineering. The views, which are not extensibility aware could be improved, and the task of improvement can be no trivial[3].

5.4 Analyses and Complexity

As it is stated before, 5.2.7, when constructing a language with an editor for it, it makes sense to provide the language with some analyses. Two types of analysis by application are considered: *informative analysis* and *preventive analysis*. Two types of analysis by the way they are initiated are defined: *self-running analysis* and *analysis on demand*.

A work flow of an analysis can be split into three major steps: initiation, running and reporting the result to the user. On each of the steps the environment (*JetBrains MPS* in this case) can provide certain support for the analysis, making the development of it easier. Below I discuss it step-by-step.

5.4.1 Analysis Initiation

It can be beneficial for the overall *IDE* performance to implement all computationally heavy analyses as *analyses on demand*. Thus the user invokes them when needed, and the question of initiation is solved.

For the *self-running analyses* the question of automatic initiation is to be solved. In *JetBrains MPS* there is no way to define, when a certain check is to be run. Thus two problems occur: some checks run too often, and decrease unnecessarily the system performance, or other checks do not happen often enough, and the user is not informed on time on the results[4].

As a solution to this an *API* extension for can be proposed: for each analysis it should be possible to define an event, on which the analysis has to repeat, as well as which nodes of the *AST* have to be covered.

For example, for the *informative analysis* identifying *abstract* classes, 4.2.3, the running event can be defined through the declaration of a new method or a new inheritance relationship established. The affected nodes of the *AST* can be correspondingly described as the changed by the modification classes.

---

[3]the author implies does from the absence of good workarounds currently

[4]There is a way to re-run *self-running analyses* in *JetBrains MPS* manually, by pressing F5

Thus the analysis will never run when changes made do not require it, and will always be run when it has to, and only on the affected nodes.

5.4.2 Analysis Running

After an analysis has been initiated, the running phase comes. Some analyses can be computationally complex, as they can require external tools running, like SMT solvers, for example, [7], or involve themselves complex algorithms.

As an example of a computationally intensive analysis, the *preventive analysis* for *abstract* classes instantiation can be taken, 4.5.5. At first, the amount of nodes of an *AST* is high, as they include local variable declarations, function parameters, function return types, class member variables and more. Each of the nodes is checked like this: at first the type is extracted, then the type is checked to be a `ClassType`. When an instance of the `ClassType` is found, the `Class` is checked on being abstract. For this all of its method declarations are checked on being *pure virtual*, and then all the base classes are checked on being abstract. If we take a reasonably big code base, like, for example the Qt library, which contains about one thousand classes[5], the amount of nodes to check and the associated computations can be immense[6].

As analyses can be started automatically, the computational load on an *IDE* can be increased as some analyses may start in parallel.

This all may present a performance problem, and a certain support from the *JetBrains MPS* is needed to handle the task of efficient analysis development.

First, as discussed in the previous subsection, it should be possible to control, when *self-running analyses* start, and the scope of their work.

Second, some value caching is needed, to store the information figured out by the analyses. For example, for each class, it could be saved, if it was found to be an *abstract* class, and the cached value should be valid, until the event to re-run the analysis is detected. It may be necessary, to retain the values after the *IDE* restart,

Third, as an analysis may take some time to execute, there has to be a way, to inform the user on the background process running, so that the user interface is more clear about analyses and their run time. Currently, *JetBrains MPS* does not inform the user about it.

Fourth, there has to be a way to limit and prioritize the analyses running at the same time. For example, *preventive analysiss* can be preferred over *informative analysiss*, and the total amount of analyses running in parallel could be limited to some number.

Fourth, as a change in one node may cause changes in other nodes, there has to be a
Take one analysis, research its complexity, figure out runtimes, MPS things
Analysis can go on the syntax tree
Which analysis can be needed in general
The analysis can take time, online running complexity
Naming conventions and regular expressions
Analysis for abstract classes instantiation can happen very often, and is complex.
RegExes for naming conventions - hard

---

[5]989 classes in Qt 5.1

[6]If we assume, that each class out of 1000 classes in a library has 1 parent, 3 member fields, 3 methods, each with 1 parameter, 1 return type, and 1 local variable, we get up to 100 000 nodes to be addressed by the check.

6 Conclusion - not started yet

Would appreciate some ideas here...

Appendix

Glossary

abstract a class in C++, having at least one method declared pure virtual in ancestors, but never implemented in the ancestors or the class itself, instances of such class can not be created. 36, 40, 46, 51, 54, 55

abstract class syntax absence a phenomenon in C++, consisting in absence of any syntax to explicitly declare a class abstract. 36, 40, 46

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

analysis on demand is an analysis, which the user must invoke explicitly in the *IDE*, it is usually computantionally expensive, c.f. *self-running analysis* . 50, 54

API Application Programming Interface. 3, 40, 54

AST Abstract Syntax Tree. 1–3, 5–10, 13, 14, 17, 18, 34, 39–41, 44, 46–48, 50, 54, 55, 61, 62

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, 20–22, 25, 27, 29, 32, 35, 37, 43–45, 51–53, 61

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, 45, 47, 54, 55, 61, 62

informative analysis is performed to inform the user about some code properties, to enhance understanding of the code and its properties. 50, 54, 55

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, 29, 32, 39, 42, 44, 52, 53

interface concept is a JetBrains MPS concept, which surves for inheriting a concept behavior interface, can not serve as a base concept. 14, 16, 51

JetBrains MPS is a language engineering environment allowing to construct incrementally defined domain specific languages. 1–3, 9, 11–18, 20–23, 29, 32, 35, 37, 42, 44–48, 51–55, 61

mbeddr same as mbeddr project. 1, 3, 4, 9, 13, 17, 20–22, 25, 26, 29, 37, 51, 53

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, 22

model correspond to one single JetBrains MPS file unit, in the mbeddr context it corresponds to C or C++ project, like one library or one executable.. 22, 44, 45

override syntax absece a phenomenon in C++, consisting in absence of any syntax to explicitly designate a method, beeing an override of another method. 36, 38

preventive analysis is performed to inform the user about the potential mistakes in advance, in order to prevent them. 50, 54, 55

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, 47

Projectional C++ a C++ flavor introduced by this work, based on mbeddr C++ implementation in the JetBrains MPS environment. 3, 22, 25–27, 32–39, 45, 47, 49–51

pure virtual are methods in a C++ class, for which intentionally no implementation is provided, they serve purely for overloading purposes, describing a pure interface. 35, 39, 40, 45, 50, 55

self-running analysis is an analysis, which determines itself the point of time, when it is performed, or this moment is determined automaticall by the *IDE*, there is no need to explicitly run such analysis. 50, 54, 55, 61

STL Standard Template Library. 1, 23, 61

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, 52

Bibliography

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

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

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

[4] Bjarne Stroustrup. Quote on Embedded C++, http://www.stroustrup.com/bs_faq.html#ec++.

[5] IAR Systems. Extended Embedded C++, http://www.testech-elect.com/iar/extended_embedded_c++.htm.

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, 2000.

[15] Herbert Schildt. *C++: A Beginner's Guide, Second Edition*. McGraw-Hill Osborne Media, 2003.

[16] Standard for programming language c++ (c++11).

[17] Stephen Prata. *C++ Primer Plus, Sixth Edition*. Addison-Wesley Professional, 2011.

[18] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[19] David Vandevoorde Nicolai Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

[20] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[21] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[22] Bjarne Stroustrup. Classes: An abstract data type facility for the c language. *Sigplan Notices*, 17:41–52, 1982.

[23] Class layout recomendations, Possibility.com. http://www.possibility.com/cpp/cppcodingstandar

[24] Google style recomendations for C++, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.

[25] Open Office C++ Coding Standard, http://wiki.openoffice.org/wiki/cpp_coding_standards.

[26] Qt Project Documentation Page, contains Q_DISABLE_COPY macro. http://qt-project.org/doc/qt-5.0/qtcore/qobject.html.

[27] Boost C++ Libraries. Official Website, http://www.boost.org/.

[28] Qt Project. The Qt Project Coding Guidelines, http://qt-project.org/wiki/category:developing_qt.