Abstract

In modern C++ development, templates are essential to achieve maintainable and reusable code. On compilation of the source code, the compilers have to perform semantic checks of the template definition and instantiation, to satisfy the rules of the C++ standard and to make sure, the code related to the templates is correct. This task can be accomplished in different ways. In this paper, the process of the semantic verification of C++ templates in CLANG is analyzed. The mechanism of checking the template definition and instantiation is explored separately by examining selected functions from the CLANG source code. Beside this, statements about the code quality and implementation are given. In addition, an analysis of the capability of CLANG to support Concepts, a type-system for template parameters, originally proposed for the C++0x standard is included in this paper. Based on the findings of the template definition and instantiation semantics, necessary changes to the code of CLANG are described as well as possible problems that can appear.

1 Motivation

The template mechanism is an important part of the C++ programming language. Templates are widely spread over the Standard Template Library [4] as in std::vector or std::map, used in libraries such as boost [5] and applied in application code. Templates increase reusability of code written by a software engineer by using generic template parameter types instead of an explicit type. Thus code with common logic only have to be written once, increasing the quality of software and make applications more reliable [1]. In this paper, we will analyze the semantic analysis code responsible for template definition and instantiation mechanism in CLANG [2], a C language family compiler frontend for LLVM [3]. Our goal is to review the quality and understandability of the code itself, since no higher level documentation for these features exists.

In the second part of this paper, we evaluate the capability of CLANG to support concepts, a "type-system" for templates originally proposed as a language feature for C++ templates [8]. We will describe necessary changes to the code to support concepts, as it may be reintroduced in future standard releases, or to use it as a compiler level type checking facility. The layout of this paper is as follows: We start with an overview of CLANG and LLVM, an introduction to template instantiation and concepts in section 2. In section 3 we analyze the template definition and instantiation mechanism based on the semantic analysis code of CLANG. In section 4 we evaluate the capability of CLANG to support concepts including a small overview of necessary code adjustments. We finish with a discussion in section 5 and the conclusion in section 6.
2 Introduction

To understand the main sections of this paper, a short overview of LLVM/CLANG, template instantiation and concepts is given in this section. This should allow our readers to better understand the further content of this paper.

2.1 CLANG and LLVM

LLVM is the abbreviation for Low Level Virtual Machine. "It is a collection of modular and reusable compiler and toolchain technologies [3]" The project was thought to be an alternative for the GNU Compiler Collection [18]. Due to the nature of GCC, it is hard to add or change functionality. LLVM aims to have an easier base to start new projects.

CLANG is a C language family frontend for LLVM. "The development of such a frontend was started out of a need - a need for a compiler that allows better diagnostics, better integration with IDEs, a license that is compatible with commercial products, and a nimble compiler that is easy to develop and maintain. All of these were motivations for starting work on a new front-end that could meet these needs [2]."

2.2 Template Instantiation Mechanism

The compile-time "life cycle" of a template in C++ can be divided into two parts:

1. Template definition

2. Template instantiation

Template definition refers to writing the template code without having code calling the template. An example of a template definition is presented in listing 1.

```c
// A. h
1
2 template<typename T>
3 const T& min(const T& a, const T& b) {
4   return a < b ? a : b;
5 }
```

Listing 1: Template Definition

Using a template definition includes the template instantiation. For example, the call-side of listing 1 may look the following way:

```c
// Main.cpp
1 // Main.cpp
2 #include "A.h"
3 4 int main() {
5   //...
6   int x = min(5,7); // instantiation
7   //...
8   return 0;
9 }
```

Listing 2: Template Instantiation

On compilation of listing 1 and 2 the compiler first checks the template definition and in a second step the template instantiation. The template definition must be valid considering all non-dependent names. A non-dependent name is, as its name says, not depending on one of the template parameter. Thus checking whether the template definition is correct is based only on the non-dependent names because the absence of an instantiation does not allow to check dependent names. The example of listing 1 contains only dependent names, since only the parameters $a$ and $b$ are used with the generic type $T$. If the non-dependent names of the template definition are correct, the dependent names are checked. Dependent names depend on the template parameter (for example in listing 1, the less-than ($<$) operator is a dependent name, because the type of $a$ and $b$ is not known). Since the dependent names depend on the template parameter, they can only be checked at instantiation time (as in listing 2). This mechanism of using dependent and non-dependent names is referre

```c
auto concept
1 2 LessThanComparable<typename T> {
3   bool operator<(const T& a, const T& b)
4     ;
5 }
```

Listing 3: LessThanComparable Concept

d to as two-phase name lookup [6].
2.3 C++ Concepts

Concepts is a C++ language feature proposed for the C++0x standard [7]. Unfortunately, concepts were removed from the standard since they were considered to be "not ready" for C++0x [9]. Concepts introduce a type-system for C++ templates. In the current definition of C++, a complete template definition is itself the only expression of a templates assumptions about its parameters [10]. Thus, concepts allow to decouple the checking of the template definition and the checking of the template parameter types. As an example, remember the template definition of listing 1. According to the template definition, the typename template parameter $T$ must comply to the following constraint:

- A less-than ($<$) operator must be present for arguments (const $T$& , const $T$& ) with a return type convertible to bool.

This assumption about the typename template parameter $T$’s capabilities is called the concept of the template parameter. According to the standard drafts for concepts in C++ [8], this concept can be expressed with the following code:

```cpp
class LessThanComparable<T> {
    bool operator<(const T& a, const T& b) {
        return a < b;  // Mapping "<" to ">
    }
};
```

Listing 4: LessThanComparable Concept

Note that an operator with a return type convertible to bool will match this concept as well, due to the fact that this conversion is implicit. An implementation of such an operator as a member-function is allowed as well, although the signature of the operator is different.

To use this concept of our example in listing 1, we replace the keyword *typename* with our concept *LessThanComparable*:

```
// A.h
template<LessThanComparable T>
const T& min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

Listing 5: Template Definition with Concept

The compiler is now able to check the template definition and the type used in an instantiation separately.

In addition, concepts introduced a mechanism called concept-maps. This technique allows to map parts of a concept to a different implementation. In our example of listing 3, we specify that the template parameter must have a less-than operator. Instead of explicitly defining such an operator, it would be possible to map an existing greater-than operator to this less-than operator. This is easy, since we only have to change the position of the arguments passed to the operator.

```cpp
concept_map
LessThanComparable<MyType> {
    bool operator<(const MyType& a, const MyType& b) {
        // Mapping "<" to ">
        return b > a;
    }
};
```

Listing 6: Concept Map Example

3 Template Instantiation in CLANG

To understand the template instantiation in CLANG, we only have the possibility to look at the code and the source file documentation. Unfortunately, no other documentation is available. Therefore we will analyze the two main source files related to template instantiation semantics and inspect the code. The source files to begin with are located in lib/Sema/:

- lib/Sema/SemaTemplate.cpp
- lib/Sema/SemaTemplateInstantiate.cpp

Of course, there are other source files which participate in the template instantiation process. Although we mention them during the code analysis, we will not have a look at them.
in depth.

3.1 Overview of the Template Semantics

The semantic analysis of template definitions is available in the source code file `lib/Sema/SemaTemplate.cpp`. The include dependencies are not surprising. There are references to nodes of the abstract syntax tree specifying template related functionality, diagnostic mechanisms and template deduction (see [11] for a full overview of the dependencies).

Figure 1: Subset of SemaTemplate.cpp includes

The functionality of the template semantics is divided into three main sections:

1. Actor
2. Builder
3. Checker

The actors (ActOn... functions) are used by the parser to "act" on the occurrence of a template declarator and different template parameter types. To build the abstract syntax tree (AST) the CLANG parser uses these functions to generate the correct AST nodes. The implementation of the parsing mechanisms related to templates is available in `lib/Parse/ParseTemplate.cpp` [14] which uses a reference (defined in `include/clang/Parse/Parser.h` [15]) to the semantic classes of CLANG.

Listing 7: Semantic reference in Parser.h

```
// Parser.h

class Parser :
public CodeCompletionHandler {
  // ...
  Sema &Actions;
  // ...
}
```

Having a look at this source file, we can easily identify other template related parsing options, e.g. `ParseTemplateDeclarationOrSpecialization(..)`.

The builders (Build... functions) are used by the declarator semantics and by the template instantiation process to build expression results by evaluating the passed arguments. We will not have a deeper look at them since they do not add valuable information to understand the semantic analysis of templates.

The checkers (Check... functions) perform the semantic analysis of the template definitions. To understand the semantic analysis, we will have a deeper look at them in the following sections.

3.1.1 Basic Semantic

To gain insight into the semantic tests, we will analyze the checks performed for class templates in the function `CheckClassTemplate(..)` of `SemaTemplate.cpp`. These tests can be applied to function templates as well with slight differences in default values for template parameters (which is not possible for functions in the current C++ standard). The same applies to template parameter deduction, due to the fact that in a template function, all template parameter types may be calculated from the function arguments. It is important to interpret the return type of the `CheckClassTemplate(..)` function correctly when analyzing the function body. The function returns an instance of `DeclResult` which is a `typedef` for a `ActionResult<Decl*>`.

```
// Ownership.h

typedef ActionResult<Decl*> DeclResult;
```
Listing 8: DeclResult Typedef in Ownership.h

This is nothing special. However, a DeclResult has a constructor for a boolean. Thus, in the function CheckClassTemplate(..) a boolean can be returned as well, since an implicit conversion from boolean to DeclResult takes place. Unfortunately, returning true is likely to be interpreted as a success indicator. Though, the DeclResult uses this boolean as flag to show that the result is invalid. This can generate confusion.

Listing 9: ActionResult Definition in Ownership.h

At the beginning of the CheckClassTemplate(..) function, there are four basic semantic tests:

- Template parameter
- Template usage
- Template scope
- Template type

The template parameter check verifies that at least one template parameter exists:

Listing 10: Template Parameter Size

If the template parameter size is not a least one, the template declaration contains an extraneous template keyword, which is invalid. The template usage validates that the template keyword is used correctly according to the underlying declaration (i.e. defining a template on a declaration of a pointer is invalid). Testing the scope of the template verifies the fact that a template declaration only can appear in namespace or class scope declarations [17]:

Listing 11: Template Scope

This test is performed using CheckTemplateDeclScope(..) in SemaTemplate.cpp. Testing the type of the underlying declaration of the template makes sure that the template is not an enumeration type, which is invalid.

3.1.2 Scope Semantic

After the basic tests, the template declaration is searched inside the specified scope generated by the parser. In case the scope is valid, it can either be namespace or a class scope, since these are the only scopes valid to declare or define a template in (validated by the template scope test in section 3.1.1). Invalid scopes will evaluate to an invalid semantic analysis result. If the scope is valid, a pointer to the declaration context of this scope is looked up. A declaration context contains all declaration in a given scope. For example, assume the following code:

Listing 12: Declaration Context Example

In listing 12, there are three declaration contexts:
1. Global declaration context containing declaration of foobar

2. Namespace A declaration context containing declaration of foo

3. Namespace B declaration context containing declaration of bar

If the lookup for the declaration context of the scope fails, the class template is invalid. It is not possible to have a template declaration with a scope specifier, but without having a declaration context for this scope or an empty declaration context, because at least the template declaration must be contained in the declaration context for the given scope.

The semantic analysis of CLANG is missing a diagnostic at this place to give a reasonable output to a user if this situation occurs. However, the developers are aware of this issue as the code comments implies. If a declaration context for this scope exists, a test whether the template declaration is complete enough for a name lookup is performed. This is the case, if all names of the template declaration are either dependent names (they are always considered complete, because they have to be checked during template instantiation), or, for all other names, a definition is present.

The remaining semantic analysis of the CheckClassTemplate(..) function deals with redeclaration, specialization, injected names and name lookup for declarations of a friend template declaration. This also includes a comparison of the template parameter list when dealing with redeclaration and specialization. Although this is a large part of the semantic analysis implemented in this function, it only deals with rules defined in the C++ standard ([17] section 14 and subsections).

Figure 2: Scope with Declaration Context

The semantic analysis of template instantiations is available in the source code file lib/Sema/SemaTemplateInstantiate.cpp [12] and lib/Sema/SemaTemplateInstantiateDecl.cpp [13]. Similar to the template definition semantics, the functionality of the template instantiation can be divided into three main sections:

1. Instantiation
2. Substitution
3. Instantiator

The instantiation (InstantiateXXX functions) deals with class template instantiation. Interestingly, instantiation of function templates is not available in this sourcefile, but can be found in lib/Sema/SemaTemplateInstantiateDecl.cpp [13]. We did not find any reason for why this separation was made. The instantiation does also manage the point of instantiation (see [6] and [17]).

The substitution (SubstXXX functions) performs the replacement of types, names, expressions and similar. The substitution mechanism is used by the instantiaton and the instantiators (either the class TemplateInstantiator which deals with class templates, or the class TemplateDeclInstantiator which deals with function templates).

The instantiator (class TemplateInstantiator in [12] and TemplateDeclInstantiator in [13]) transforms (instantiates) different entities of a template class or function.

To understand the template instantiation in CLANG, we will look at the InstantiateClass(..) function in lib/Sema/SemaTemplateInstantiate.cpp [12], which performs a template instantiation for a class template and analyzes the semantic tests.

3.2.1 Definition Lookup Semantic

A class template instantiation requires a valid definition of a template. However, it is possible that the InstantiateClass(..) function receives a class template declaration to instantiate. Therefore, the first step is to successfully retrieve the definition of such a declaration. If this is not possible, an instantiation process can not take place and the according diagnostics are performed to give a valuable feedback to a user. When retrieving a template definition was successful, the point of instantiation for the template is recorded in the instantiation context.

3.2.2 Attribute Instantiation and Base Classes

In the next step of the process, the attributes of the class definition are instantiated. The attribute instantiation uses the substitution described in section 3.2 to replace the generic template types with the type of the template parameter found at the call-side of the template class. These step is completed with the attribute instantiation in the inheritance hierarchy, if there are any template base classes. Although we would think that the members are instantiated here as well, this happens after traversing the inheritance hierarchy. Actually, this have something to do with the lexical and semantical scope of class template members. For example, in listing 13 the class B has the lexical scope of the template class, though semantically it is introduced in the global namespace (see [12]):

```cpp
1 template<int i>
2 class A
3 {
4   class B *g;
5 }
```

Listing 13: Member Instantiation Scope

To finish the instantiation, it is checked whether the field instantiation is valid.

3.2.3 Partial Specialization

Before finishing the instantiation of the class template, a test is performed whether any partial specializations for the current class template are available. If there are partial specializations available for the class template, each one is instantiated by using the non-partial specialized class template instantiation (i.e. the template instantiation we have instantiated before). This is done by reusing the non-partial specialized template instantiation and substituting the types known from the partial specialization. For example, assume the following template definition and partial specialization:

```cpp
1 // Template Definition
2 template<typename T, typename U>
```


struct something
{
    // do something
};

// Template Partial Specialization
template<typename T>
struct something<T, int>
{
    // do something special with ints
};

Listing 14: Template Definition with Partial Specialization

The template definition from listing 14 is reused by the template partial specialization, substituting an int for the second template parameter. The first template parameter does not have to be substituted, because this is already done in the original template instantiation. The responsible code for this task is available in listing 15.

for (TemplateDeclInstantiator::delayed_partail_spec_iterator P = Instantiator.
delayed_partail_spec_begin(), PEnd = Instantiator.
delayed_partail_spec_end(); P != PEnd; ++P)
{
    if (!Instantiator.InstClsTempPartSpec(P->first, P->second))
    {
        Invalid = true;
        break;
    }
}

Listing 15: Instantiation of Partial Specializations in CLANG

(The original name of the instantiation method is Instantiator.InstantiateClassTemplatePartialSpecialization(...). We have shortened the name for simplicity). The instantiation of partial specializations will induce further semantic checks on the newly substituted types from the partial specialization. This includes checking whether the type substitution was valid, or in case of an invalid substitution, to stop the instantiation with a diagnostic returning an error message to a user.

4 Concepts in CLANG

In the previous section, we described the semantic analysis of template definition and instantiation in CLANG using chosen functions from the source files. Having concepts as a language feature would change this process. We will analyze and describe changes that are necessary to support concepts by CLANG in the following sections.

4.1 Grammar and Parser Changes

Apparently, extending the grammar and the parser is inevitable for having concepts as a language feature. This includes adding new keywords such as concept, concept_map and the alternative syntax of applying a concept to a template by using the keyword with. These changes must include extensions to the abstract syntax tree as well. However, We will not further investigate on how to apply these changes to CLANG, since our focus lays on template instantiation.

4.2 Internal Representation Changes

The current C++ standard [17] allows the usage of class or typename when defining a template parameter in a template declaration (assuming we do not use types explicitly and no partial specialization). In CLANG, this difference is represented by a flag whether class (flag is false) or typename (flag is true) was used [16].

class TemplateTypeParmDecl :
public TypeDecl
{
    bool Typename : 1;
    //...
};

Listing 16: Typename or Class

This internal representation has to be changed to point to the appropriate concept(-map) definition or declaration, if one is given. This could be done by adding a pointer to a concept definition or declaration:

class TemplateTypeParmDecl :
2 public TypeDecl
3 {
4 ConceptDecl* Concept;
5 // . . .
6 public:
7 bool hasConcept() const
8 {
9 return Concept;
10 }
11 // . . .
12 };

Listing 17: Possible Implementation of a Template Parameter with Concept

In case the template parameter is unconstrained, a *Null-Concept* using the Null-Object design pattern could be used [21]. However, this could increase the compile-time memory usage in cases where most of the templates in the source code are unconstrained (i.e. have no associated concept - specified by *typename* or *class*).

### 4.3 Template Definition Semantic Changes

During the semantic check of a template definition, the template definition has to be checked whether concepts are defined for a template parameter. It is possible to distinguish between three cases:

1. All template parameter have an associated concept
2. A subset of all template parameter have an associated concept
3. None of the template parameters have an associated concept

To illustrate these cases, assume the following example:

```c++
1 // Case 1: Complete Constraints
2 template<LessThanComparable T,
3 Incrementable U>
4 void foobar1(T t, U u)
5 {
6 // . . .
7 }
8 // Case 2: Partial Constraints
9 template<LessThanComparable T,
10 typename U>
11 void foobar2(T t, U u)
12 {
13 // . . .
14 }
15 // Case 3: No Constraints
16 template<typename T, typename U>
17 void foobar3(T t, U u)
18 {
19 // . . .
20 }
```

Listing 18: Concepts in Template Definition

The third case does not require any further concept related processing. It is an "original" template definition as it already exists in the current C++ standard [17]. The first and second case do require additional processing. The difference between the first and the second case is that in case 1, all template parameters have an associated concept. But in case 2, an associated concept exists only for the first template parameter $T$. For any template parameter where an associated concept exists, the definition of this concept, or at least the declaration of the concept has to be found. It may make more sense to only search for the concept declaration, since the process of checking a template parameter against a concept takes place in the template instantiation mechanism. If a template definition with template parameters having associated concepts is never instantiated, the lookup for the definition is not necessary. Therefore, having only the declaration of the concept available at template definition time is sufficient. This also implies, that a concept can be considered as a dependent-name, according to the two-phase name lookup [6]. In addition, the separation of concept declarations at template definition and concept definitions at template instantiation in the lookup can increase compile-time efficiency in time and memory-space due to the delayed lookup mechanism.

### 4.4 Template Instantiation Semantic Changes

The instantiation process requires the largest amount of extensions or changes to achieve
concepts support. In template instantiation, the substitution types of the template parameters have to be checked against the associated concepts. There are two different possibilities:

1. An associated concept exists for the template parameter

2. An associated concept does not exist for the template parameter (or a Null-Concept [21])

Case 2 is equivalent to the existing template situation in the current C++ standard [17]. The template parameter was defined with type-name or class and no explicit constraints are defined for the template parameter. However, the substitution still has to be checked for validity, since a non-existing concept does not imply that the substitution type does automatically comply to the usage of the type in the template. It is up to CLANG to generate accurate diagnostics indicating missing requirements on a substitution type, decoupled from the concept validation diagnostics.

In the first case where an associated concept exists, there are several steps necessary to check the template parameters against the concepts. Depending on whether the definition or declaration of the concept was retrieved during template definition, the definition of the associated concept has to be found. A missing concept definition should yield an error to the user, because a concept declaration is not sufficient to validate a substitution type for a template parameter. The concept definition is subsequently used to validate the substitution type of the template parameter. This allows to check the type independent of the template definition. This has the advantage, that the type-checking is decoupled from the template definition. The same process has to be applied for concept maps, although checking the concept against the substitution types is different. For concept maps, the mappings (e.g. less-than operator to greater-than operator) must be validated as well. The benefit of applying the concept mechanism as described, is that nearly all diagnostics related to template parameter checking during instantiation can be placed together into the code representation of concepts. Having a Null-Concept (Null-Object) [21] would assist this clean encapsulation. However, it is not sufficient to just check the template parameter against the associated concept. Even an existing concept does not imply that the substitution type does automatically comply to the usage of the type in the template.

The diagnostics of concept-checking should also be aware of the inheritance hierarchy during the instantiation process. Thus making sure that the Liskov Substitution Principle (LSP) [22] is not violated. In the example of listing 19, the class $B$ overrides the member function $foobar(\ldots)$, but the template parameter is more constrained compared to the base class $A$. This violates LSP. However, checking for this violation may not be appropriate, or should only yield a warning, because a software engineer may not be fully aware of this paradigm in the given context. In addition, such a check could break framework code where it is likely to unintentionally violate the principle, but still having valid behaviour.

```cpp
// Concept LessThanComparable
auto concept
LessThanComparable<typename T> { 
  bool operator<(const T&, const T&); 
};

// Concept LessThanEqualComparable
auto concept
LessThanEqualComparable<typename T> { 
  requires LessThanComparable<T>;
  bool operator==(const T&, const T&);
  bool operator!=(const T&, const T&);
};

// For brevity, ignore the possible 
// mapping for equal to unequal, v.v.

// A.h
template<LessThanComparable T>
struct A 
{ 
  virtual void foobar(T t) 
  { 
  // ... 
  }
};
```


### Listing 19: Example of LSP Violation with Concepts

```cpp
template<LessThanEqualComparable T>
struct B : A<T>
{
  // LSP Violation: 'T' of 'B' has more constraints than 'T' of 'A'
  virtual void foobar(T t)
  {
    // ...
  }
};
```

We have to note that LSP violations are possible whether we have concepts or not. However, it may be valuable to have a validation option in case concepts are used, since the necessary informations are present.

#### 5 Discussion

In our opinion, CLANG does the job of template instantiation very well. Especially the generated diagnostic messages to indicate the type of error and the position are impressive compared to the compiler errors generated by the GNU Compiler Collection (GCC/g++) [18]. The reason that CLANG has become a serious competitor forced the GCC developers to improve their diagnostics as well over the last years. However, we still think that the diagnostics of CLANG are more clear and accurate compared to the GCC diagnostics. A good indicator for this is the usage of CLANG as the compiler frontend in the Xcode IDE since version 4 of Apple.

LLVM was thought to be an easier to use compiler infrastructure compared to the GNU Compiler Collection. CLANG does follow this rule as well, but getting into the project and understand the source code is barely easier compared to GCC. The lack of more expressive documentation than only source code comments and the source code itself made it hard to understand the process and draw reasonable conclusions about it. In addition, we encountered different programming styles mixed. We do not question that there is a style guide for ”hacking” on CLANG. However, it seems that sometimes this guide is not enforced (e.g. variable names in loops). This also includes mixing return results where sometimes a DeclResult is used, and sometimes only bools using the implicit constructor conversion of DeclResult, but the decision to return the type directly or using a conversion appears to be random and not by following specific rules (e.g. in CheckClassTemplate(..) of lib/sema/SemaTemplate.cpp). In documentation work of CLANG, we would recommend to write sequence diagrams and explanations to the template instantiation process (and maybe to other parts of the code as well, e.g. similar to figure 3). This makes it easier for beginners to understand the project, component collaboration and may lower the barrier for a user interested in improving CLANG to start collaborating in the project.

The CLANG project homepage [2] states as one of its main feature and goal to be ”A real-world, production quality compiler”. The results generated by the compiler can be considered production quality. However, the source code analyzed when writing this paper is in our opinion not production quality. A simple task search for ”FIXME” and ”TODO” in SemaTemplate.cpp and SemaTemplateInstantiate.cpp using Eclipse, yields 53 entries, where parts of the ”FIXME” notes do not have an entry in the bug tracker platform of CLANG (if we include other template related semantic sourcefiles, there are about 101 entries). Although this may be internal rules of the developer team, we would recommend to track this outside of the source code as well, as this makes things more reliable and comprehensible.

The separation of template definition semantic checks and template instantiation semantic checks is made on a file basis: SemaTemplate.cpp deals with template definition where SemaTemplateInstantiate.cpp implements template instantiation. However, some of the functionality is mixed between
these files. There are code parts responsible for template instantiation which is called from the template definition code (using the \texttt{ActOn...()} functions). In our opinion, this should be invoked from the parser, not from the template definition to achieve a better separation. However, this may be more complex, because going from definition to instantiation is more "natural" and the necessary informations already exists in the instantiation context, since CLANG starts at the definition.

Adding concept support to CLANG is possible, but includes changing several working parts of the current code. The most interesting question of a concept implementation would be whether the diagnostics become better, worse or stay at the same level. The performance of concepts in CLANG is a second important part of such an implementation. There is a GCC based version of a compiler supporting concepts [20], but the performance was very bad. ConceptGCC could be a good point to look at when implementing concepts, since they probably already solve several problems that can occur when implementing concepts.

6 Conclusion

CLANG has the capability to become one of the best and widely-spread compilers for the C language family. The template instantiation process of CLANG is implemented very well and can, once you understand the internals, be easy extendend. The performance of the instantiation is as good as the performance of GCC, or sometimes even better [19]. The diagnostics of CLANG are not always perfect, but far better compared to GCC, which in our opinion, is one of the most important features for end users. Therefore, we are confident that the CLANG project will displace GCC or at least be the best alternative.

7 Future Work

Even if concepts are no longer in the C++0x standard, compiler based concept checking could increase the reliability and quality of C++ code. It seems that a CLANG based concept compiler was started in mai 2011 [23]. However, this is in an early stage, therefore not much information is available.

8 Acknowledgement

I would like to thank Peter Sommerlad for the support and corrections while writing this paper. In addition, I would like to thank Sebastian Hunkeler for his help in reviewing the paper and for his correction-proposals. This paper is a term paper at the university of applied sciences (HSR) in Rapperswil, Switzerland.

References


Timestamp: 2011/06/02