

Concepts for CDT

Author

Lukas Wegmann

Supervisor

Prof. Peter Sommerlad

Technical Adviser

Thomas Corbat

UNIVERSITY OF APPLIED SCIENCE RAPPERSWIL

Term Project, January 31, 2015

Abstract

Concepts is one of the most awaited language constructs of the C++17 standard and is going to simplify the work with templates significantly. The new language feature introduces a way to describe syntactical requirements for template parameters and check template arguments during instantiation. Requirements on template parameters are no longer only implied from their use in the template body. This enables cleaner error messages during compilation and more self-explanatory APIs. But, up to now there is only one working implementation of the current Concepts proposal as part of the GCC C++ compiler and the support in IDEs is non-existent.

During this term project, we have worked on the integration of the Concepts proposal in the Eclipse CDT development environment. We have analyzed the new language features and identified the required changes and extensions to both parser and semantic analysis.

The resulting implementation includes complete support for the new syntax introduced by the proposal. Furthermore, the binding resolution algorithm has been extended such that all names used for constraining template arguments and defining concepts can be correctly resolved and checked for binding errors. This gives basic support for the new language features and enables the further implementation of more advanced features like concept checks in the IDE, constraint-based auto-completion and new refactoring tools.

Management Summary

This report describes the integration of *Concepts*, a new language feature for C++, into Eclipse CDT.

C++ Concepts

Concepts is a long awaited language feature for C++ and is supposedly part of the upcoming C++17 standard. The goal behind the Concepts proposal is to allow library designers to explicitly describe syntactical requirements for template arguments as a part of the public API. This enables a more precise error reporting when using template based libraries. It also reduces the required amount of documentation because Concepts are part of a formal and unambiguous description of how templates can be used.

In contrast to earlier versions of the Concepts proposal, the current specification describes a language feature that can be iteratively adopted to existing code. This makes it also interesting for owners of big legacy projects to gradually migrate to Concept based libraries.

Because the standardization of this feature is still in progress, tooling that supports the additional syntax and capabilities is currently almost non-existent.

Goals

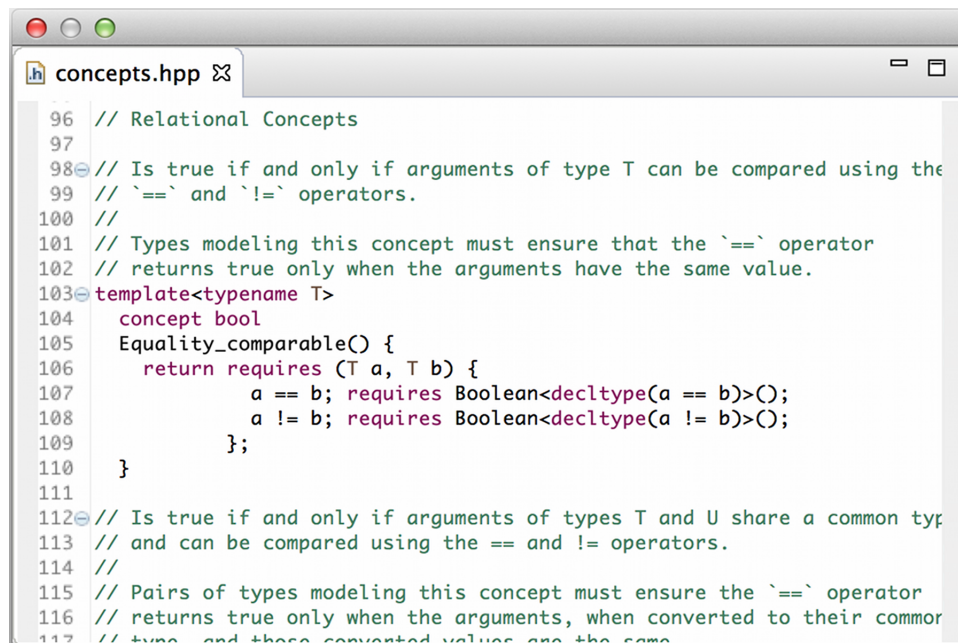
Integrated development environments (IDE) like Eclipse CDT do currently not support the new language features from the Concepts proposal. This makes using Concepts in a modern IDE almost impossible. Because the IDE does not know the additional language rules, it spuriously reports errors in the source code where syntax related to Concepts are used.

The goal of this term project is to enable the use of Concepts in Eclipse CDT and lay a foundation for additional tools that became possible through the new language features. Thus, IDEs should be ready to support Concepts when they finally become part of the C++ standard.

Results

The resulting implementation supports the new syntax introduced by the Concepts proposal. The IDE can successfully parse syntactically correct C++ code that uses Concepts and creates an unambiguous internal representation (AST) for further processing.

Figure 0.1.: A concept in CDT's editor view with correct syntax coloring



```
96 // Relational Concepts
97
98 // Is true if and only if arguments of type T can be compared using the
99 // `==` and `!=` operators.
100 //
101 // Types modeling this concept must ensure that the `==` operator
102 // returns true only when the arguments have the same value.
103 template<typename T>
104     concept bool
105     Equality_comparable() {
106         return requires (T a, T b) {
107             a == b; requires Boolean<decltype(a == b)>();
108             a != b; requires Boolean<decltype(a != b)>();
109         };
110     }
111
112 // Is true if and only if arguments of types T and U share a common type
113 // and can be compared using the == and != operators.
114 //
115 // Pairs of types modeling this concept must ensure the `==` operator
116 // returns true only when the arguments, when converted to their common
117 // type, and those converted values are the same.
```

Developers can now use Concepts and the IDE reports syntax errors in case of incorrect use of the new language features. Furthermore, basic supportive features are available. For example syntax coloring, jumping from concept references to their definition and highlighting other occurrences of a selected concept name.

Beside this basic integration into Eclipse CDT, we've analyzed how this work can be used to provide more advanced tools that support developers while working with Concepts.

Contents

1. Introduction	1
2. Introducing Concepts	2
2.1. Why Concepts?	2
2.2. History and Specification	3
2.3. Motivating Example	4
2.4. Concept Definitions	6
2.4.1. Constraint Expressions	7
2.4.2. Requires Expressions and Requirements	7
2.5. Constraining Templates	8
2.5.1. Requires Clause	8
2.5.2. Constrained Parameters	9
2.5.3. Constrained Template Parameters	10
2.5.4. Template Introductions	10
3. Concepts and IDEs	12
3.1. Syntax Checking and Coloring	12
3.2. Concept Checks	13
3.3. Assistance in Constrained Generic Code	14
3.4. Automated Refactorings	15
4. Parser Infrastructure of Eclipse CDT	17
4.1. General Overview	17
4.2. Parser	18
4.3. Abstract Syntax Tree	19
4.4. Bindings	20
4.4.1. Binding Resolution	20
4.5. Codan	21

5. Implementing Concepts Lite in CDT	22
5.1. Concept Definitions	22
5.1.1. Parsing the Concept Keyword	23
5.1.2. Concept Bindings	24
5.1.3. Overloaded Concepts	25
5.2. Requires Expressions and Requirements	26
5.2.1. Parsing Requires Expressions	27
5.2.2. Scope Lookup for Requirement Expression Parameters	28
5.3. Requires Clauses	28
5.3.1. Parsing Require Clauses	29
5.3.2. Persisting Constraints	30
5.4. Handling Syntactic Sugar	30
5.5. Template Introductions	32
5.5.1. Resolve Introduced Parameters	34
5.6. Constrained Parameters	34
5.6.1. Ambiguity Resolution	36
5.6.2. Invented Template Parameters	36
5.7. Constrained Template Arguments in Parameters	37
5.7.1. Combinational Explosion in Ambiguous Template Arguments	37
5.8. Constrained Template Parameters	39
5.8.1. Deriving Bindings for Constrained Template Parameters	41
6. Testing and Verification	42
6.1. Testing Parser and Binding Resolution	42
6.2. Verifying with Origin	43
6.2.1. Setup	43
6.2.2. Checklist	44
6.2.3. Findings	44
6.2.4. Adequacy	45
7. Conclusion	47
7.1. Accomplishments	47
7.2. Remaining Tasks	48
A. Project Management	I
A.1. Tools	I
A.2. Time Report	I
A.3. Project Schedule	II

B. Bibliography

III

1. Introduction

Concepts, as proposed for C++, are a relatively unique feature in the landscape of programming languages. This is why we start this report with explaining why they became necessary and how they will be implemented in C++ in chapter 2.

In the following chapter 3 we analyze the implications of Concepts on IDEs. At one hand, we define what syntax checking stands for when applied to such a complex language like C++ that has no clear borders between syntax and semantic. At the other hand, we give an overview over how the support for Concepts could be further extended with supportive tools like automated refactorings.

Chapter 4 then focuses on the existing parsing infrastructure of Eclipse CDT, our target platform.

Based on this preliminary work, we give a detailed insight into the implementation in chapter 5. This includes descriptions of fundamental design decisions and how we approached the more severe obstacles during the implementation. Chapter 6 then shows how we ensured that our implementation is correct and meets the specifications from the Concepts proposal.

Finally, chapter 7 concludes this project and gives an overview over the remaining work necessary to fully support Concepts in Eclipse CDT.

2. Introducing Concepts

In this chapter we want to give a brief overview over the motivations and ideas behind the Concepts proposal and an insight in its language constructs.

2.1. Why Concepts?

One of the strengths of the template based approach to generic programming in C++ is the fact that it does not require an explicitly specified interface on template parameters. Instead of that, a template defines an implicit interface for its parameters based on the syntactic elements that are applied on them. During template instantiation, the compiler checks if the concrete arguments actually implement those implied interfaces by replacing the placeholders by their concrete types (or values in case of non-type templates) and then type check the substituted code.

Listing 2.1: A simple function template with one type parameter T

```
template<typename T> T twice(T x){ return x + x; }
```

Given the function template in Listing 2.1 the template defines an implicit interface that requires that the addition operator is defined on T and its return type must be also T. This function is now applicable on every type fulfilling this contract, e.g. `twice(21)` or `twice("foo")`.

This example also demonstrates another property of C++ templates: Types implicitly implement the interface required by a template as soon as the required syntax is supported. Hence, the call `twice(21)` is valid code because the addition operator is defined on `int` without the need to explicitly denote that `int` is intended to support this interface.

When we compare this to the way how genericity is achieved in languages that mainly use subtyping for polymorphism (like Java or C#) we see that C++ templates definitely offer more flexibility. Listing 2.2 shows the implementation of a similar generic function as in Listing 2.1 in Java. But in contrast to the C++ version we have to constrain the type parameter T to be an implementation of the

`Addable` interface. Otherwise, the compiler would object on the call to the `add` method. This limits the usefulness of the `twice` method as it is only applicable with types that are marked as implementations of `Addable`.

Listing 2.2: A generic method in Java with one type parameter

```
public static <T extends Addable> T twice(T x){ return x.add(x); }
```

But the greater flexibility of C++ templates comes with some downsides that can be summarized in two points:

Late Error Detection

Because C++ compilers first substitute template arguments and then type check the instantiated template, errors caused by invalid template arguments are reported where the unsupported syntax is used and not right where the user invoked the erroneous template instantiation. This often results in hard-to-read compiler errors. Especially when the substitution failure occurred in nested templates.

API Documentation and Readability

The signature of a template class or function does not include requirements to its template parameters. Instead of that, API designers have to state these requirements in documentation comments or separate documents which requires discipline and leads to redundancy between code and documentation.

Both of these issues can be resolved by extending the language such that it becomes possible to describe requirements on template parameters that are checked during compile time. Hence, template arguments that do not satisfy all of the required properties can be reported right where the developer passed the wrong template argument which allows more precise error messages. Also the second issue can be addressed because such requirements are available to library users and are easily comprehensible for both developers and development tools like IDEs, linters and compilers.

2.2. History and Specification

One of the earliest discussion on how template arguments can be constrained in C++ can be found in “The Design and Evolution of C++” [Str94]. Stroustrup states that constraints would improve readability and early error detection and discusses several approaches to add this feature to the language.

From the C++98 standard [ISO98] on, concepts are used to describe syntactic and semantic requirements to template parameters. These concepts are mainly written in prose and describe the conventions to follow in order to work with template functions of the standard libraries.

The first major attempt to finally include concepts into the C++ language was part of the standardization of C++0x [ISO11]. In contrary to the current version of the concept proposal, this version proposed a radical implementation of concepts that made the iterative migration to constrained generic code almost impossible. Furthermore, it included the additional language constructs axioms and concept maps. Axioms were used to describe semantic requirements in concepts and concept maps to provide implementations of a concept for a certain type. Due to various reasons the standardization committee finally decided that concepts should not be included in C++0x [Str09].

Currently, there is some work in progress to include a simpler version of concepts (formerly known as “Concepts Lite”) into the upcoming C++17 standard. The working draft [Sut14b] is maintained by A. Sutton and is currently under continuous revision. A less formalized discussion of the current concept proposal can be found in [SSR13]. Please note that this document is not entirely up to date with the current revision of the proposal but should still give a deeper insight into the new language constructs.

The repository containing the most recent version of the concept proposal can be found at [Sut15b]. Because there is a lot of ongoing work on the proposal at the time of this term project, it is possible that some implementation details are no longer up to date with the current specification.

2.3. Motivating Example

The STL header specifies various generic functions that are mainly useful to work on containers, collections or - more general - on ranges of elements [ISO13]. One example of such a function is `all_of` which tests a condition on all elements of a range of elements. The standard specifies that `all_of` has a function signature as specified in Listing 2.3,

Listing 2.3: A function of the standard library using concepts by convention

```
template <class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate
    pred);
```

The name of both template parameters `InputIterator` and `Predicate` already gives some hints on how potential arguments will be used and what requirements on them are implied. A predicate, for example, is commonly known as a function that takes at least one argument and returns true if the value belongs to a set described by the predicate or false otherwise. Hence, despite the fact that the template parameters are not formally specified, there are conventions that help users working with template libraries.

In case of the standard library there are already precise descriptions of all template parameters. E.g. for predicates, the standard states “The `Predicate` parameter is used whenever an algorithm expects a function object that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true[...]`”. Furthermore, there are also similar descriptions for `InputIterator` and other concepts used in the standard library.

Ideally, we should be able to formally describe these syntactic requirements as illustrated in Listing 2.4. This template signature uses the three types `It`, `Pred` and `Elem` as template parameters and a special notation called *requires clause* to constrain these types. The first part of the conjunction states that `It` and `Elem` must provide the syntax described by the `Input_iterator` concept and the second part requires `Pred` and `Elem` to be a model of the `Predicate` concept.

Listing 2.4: A function of the standard library constrained by two formal concepts

```
template<typename It, typename Pred, typename Elem> requires
    Input_iterator<It, Elem> && Predicate<Pred, Elem>
bool all_of(It first, It last, Pred pred);
```

These two concepts are listed in Listing 2.5. They describe the required syntax and how their template arguments relate to each other. As we can see, syntactic requirements in concepts are described by example. For instance, the `requires` expression of the `Predicate` concept states that if there is a `p` of type `Pred` and an `e` of type `Elem`, the expression `p(e)` must be valid and evaluate to a value of type `bool`.

The `Input_iterator` concept uses similar examples with the variables `a` and `b` of type `Iter` to state that the dereference operator, the pre-increment operator and the inequality comparison must be available.

Listing 2.5: Two simplified concepts for Predicate and Input Iterator

```
template<typename Pred, typename Elem>
concept bool Predicate =
    requires(Pred p, Elem e) {
```

```
    {p(e)} -> bool;
};

template<typename Iter, typename Elem>
concept bool Input_iterator =
    requires (Iter a, Iter b) {
        {*a} -> Elem;
        {++a} -> Iter;
        {a != b} -> bool;
    };
```

The advantage of describing requirements by example is that they do not restrict how the syntax must be provided. Hence, in order to implement the `Predicate` concept for a class `C` one can provide the overloaded application operator as a member of `C` or as a static global function. This gives great flexibility as it is possible to provide implementations of a concept for types not owned by oneself.

Finally, the constrained signature of `all_of` provides detailed information about the expected template arguments in an unambiguous and verifiable way.

2.4. Concept Definitions

A concept is either a function or variable template of type `bool` declared with the new `concept` specifier. Additionally, every concept definition is implicitly declared as a `constexpr` declaration but the `constexpr` specifier must not be used.

Listing 2.6: Two concepts defined as a variable and a function respectively

```
template<typename T> concept bool Any = true;

template<typename T> concept bool None(){
    return false;
}
```

The first example definition in Listing 2.6 demonstrates how a concept can be defined using a template variable (a *variable concept*). The resulting concept `Any` accepts every type, or to put it the other way around: *Every type is a model of Any*. The second example defines the contrary concept `None` as a *function concept*. Because `None` always evaluates to `false`, there are exactly zero types that are a model of `None`.

Every concept has at least one template parameter that is not necessarily a type parameter. For example the concept in Listing 2.7 restricts integers to be divisible by two. Such concepts could be used to constrain sizes and limits of containers or similar structures or to provide specializations for containers with a certain size.

Listing 2.7: A concept with a non-type template parameter

```
template<int I> concept bool Even = I % 2 == 0;
```

2.4.1. Constraint Expressions

The body of every concept definition must consist of exactly one constraint expression. A constraint expression is an expression of the form **E**; where **E** is a constraint. The exact definition of a constraint is given in [Sut14b, 14.10] but can be summarized in a heavily simplified form to: A constraint expression is a logical expression of type `bool` that can be evaluated at compile time.

Furthermore, a constraint expression is only valid if it can be normalized to a logical expression consisting of only atomic constraints and the logical **and** and **or** operators. An atomic constraint is a single expression that evaluates to `bool` without using implicit conversions.

2.4.2. Requires Expressions and Requirements

Requires expressions are a new language construct to describe syntactic requirements by example. They can only be used within a constraint expression.

A requires expression can introduce local parameters as “prototypes” to describe the required syntax. The requires expression in Listing 2.8 uses two local parameters `a` and `b` of type `T`.

Listing 2.8: A concept using one requires expression with two local parameters

```
template<typename T> concept bool Numeric =  
    requires(T a, T b){  
        a++; // a simple requirement  
        typename T::result; // a type requirement  
        {a + b} noexcept -> T; // a compound requirement  
        requires Copyable<T>; // a nested requirement  
    };
```

The body of a `requires` expression contains at least one requirement separated by semicolons. These requirements can use all symbols available from the surrounding context like local parameters, template parameters or other visible declarations. Listing 2.8 gives an example for each available type of requirement:

Simple Requirement

Asserts that the expression is valid.

Type Requirement

Asserts that the type stated exists.

Compound Requirement

A simple requirement with optional `noexcept` and return type requirements.

Nested Requirement

Asserts the nested `requires` clause.

The `noexcept` requirement asserts that the function enabling the according syntax uses the `noexcept` specifier.

2.5. Constraining Templates

Besides the syntax for concepts definitions the proposal describes four different notations to constrain template parameters.

2.5.1. Requires Clause

Requires clauses are the most expressive notation for constraining templates. In fact, every constraint using one of the other notations can always be transformed to an equivalent template definition with a `requires` clause.

A `requires` clause can be used after the closing angle bracket of every template definition as shown in Listing 2.9 and consists of the `requires` keyword followed by a constraint expression (see subsection 2.4.1).

Listing 2.9: A `requires` clause referring to the `Number` concept

```
template<typename T> requires Number<T>  
T add(T a, T b);
```

During template instantiation, the compiler evaluates the according `requires` clause and reports which constraint has been violated if the constraint expression results in `false`.

Sometimes it may also be necessary to define constraints that refer to parameters of a function template. In this case it is also possible to additionally use a `requires` clause after the function signature as shown in Listing 2.10.

Listing 2.10: A constrained function

```
template<typename T>
auto add(T a, T b) -> decltype(a+b) requires Number<decltype(a+b)>;
```

2.5.2. Constrained Parameters

The shortest notation for writing simple constraints on templates are constrained parameters. A constrained parameter is a parameter declaration that uses a concept name instead of a type specifier. The use of a concept name instead of a type specifier is also called a constrained type specifier in the proposal. As we can see in Listing 2.11 this notation obscures the fact that `printNum()` is actually a template.

Listing 2.11: A constrained parameter and the equivalent template declaration using a `requires` clause

```
void printNum(Number a);
// ...is equivalent to...
template<typename T> requires Number<T>
void printNum(T a);
```

Constrained type specifiers may also have template arguments if the referred concept has more than one template parameter. Hence, using `Input_iterator<int>` as a constrained type specifier is equivalent to using a template parameter `T` and the `requires` clause `requires Input_iterator<T, int>`.

Additionally, constrained type specifiers can also appear as template arguments within parameter declarations. This is illustrated in Listing 2.12

Listing 2.12: A constrained template argument and the equivalent template declaration using a `requires` clause

```
void printAllNums(Container<Number> c);
// ...is equivalent to...
template<typename T> requires Number<T>
```

```
void printAllNums(Container<T> c);
```

Because constrained type specifiers conceal the actual template parameter this notation is only useful for simple constraints as it can not be easily used with an additional `requires` clause. Furthermore, it is not intuitively recognizable if using the same constrained type specifier more than once in a parameter list results in one template parameter for all occurrences or in separate template parameters for each occurrence. The proposal specifies that the former is the case as illustrated in Listing 2.13.

Listing 2.13: Multiple identical constrained parameters

```
void printNums(Number first, Container<Number> remaining);  
// ...is equivalent to...  
template<typename T> requires Number<T>  
void printNums(T first, Container<T> remaining);
```

2.5.3. Constrained Template Parameters

The name of a concept may also be used to directly constrain a template parameter as demonstrated in Listing 2.14.

Listing 2.14: A constrained template parameter

```
template<Number T>  
void printNum(T a);  
// ...is equivalent to...  
template<typename T> requires Number<T>  
void printNum(T a);
```

In contrast to constrained parameters, using constrained template parameters makes it easier to add additional constraints because the template parameter is visible and can be referenced in an optional `requires` clause.

2.5.4. Template Introductions

The last shorthand notation for constraining templates are template introductions. A template introduction replaces the template definition by the name of the constraining concept followed by curly braces containing the names of the constrained template parameters. An example of a template introduction and the according template definition using a `requires` clause is given in Listing 2.15.

Listing 2.15: A function template constrained by a template introduction

```
Convertible{T, U}  
U convert(T t);  
// ...is equivalent to...  
template<typename T, typename U> requires Convertible<T, U>  
U convert(T t);
```

Template introductions are easier to use than constrained template parameters or pure requires clauses in a few cases. Especially when more than one of the arguments passed to the concept are used in the following function signature like it is the case in Listing 2.15.

3. Concepts and IDEs

This chapter examines how IDEs can support the work with the new language features introduced by the Concepts proposal and how these features can be used to assist IDE users.

3.1. Syntax Checking and Coloring

Syntax checking is the minimum viable product for concepts support in IDEs: It is the core feature that must be available in order to support more sophisticated features based on concepts.

A complete integration of syntax checking should fulfill the following property:

For every program that compiles on a reference compiler the IDE must not report any errors.

This does not necessarily imply that the IDE must report an error if a program does not compile. But the IDE should be capable of detecting obvious syntax errors.

Unfortunately, in case of C++ there is no clear line between syntactical and semantical errors and checking the syntax requires more than just porting the languages production rules to the editor (see section 4.2 for more details).

This is why we use a more technical definition for syntax checking in this term project:

For every program that compiles on a reference compiler the IDE must be able to construct an unambiguous abstract syntax tree without reporting any errors.

This also describes the level of integration we want to aim for during this project. Thus, we say that every program for which the IDE can build an AST is syntactically correct. This AST must not contain any ambiguities which requires, in some cases, the resolution of names to inspect if it is used in the correct context. Afterwards, further semantic analysis can be used to reduce the number of programs

that the IDE reports no errors for but do not compile on the reference implementation (false positives). Two of such semantic tools are described in the following sections.

After successful syntactical analysis of a program IDEs can use this information to highlight certain language structures in the editor. This is commonly known as syntax coloring or syntax highlighting. A minimal level of syntax coloring is highlighting built-in language keywords like `concept` and `requires`. Concerning the features of the concepts proposal also more sophisticated coloring, like highlighting requirements for templates, would be possible.

3.2. Concept Checks

The ability to detect violations of constraints during template instantiation and precise reporting of the error is the core feature of concepts. Integrating this level of error reporting in IDEs would probably bring many benefits to developers.

Unfortunately, providing concept checks requires a complete implementation of many C++ language features:

- `constexpr` evaluation
- Evaluation of compiler specific expressions like `__is_class` or `__is_base_of`
- Evaluation of the `sizeof` operator which requires knowledge about the target environment
- Template instantiation
- Template overload resolution

Most of these features are not or only partially implemented in current IDEs. Thus, providing an accurate implementation of concept checks would require great effort from IDE developers. But even a correct implementation can not guarantee that it behaves analog to a specific compiler implementation because some of the language features depend on compiler internals and the target environment.

Given the limited support for certain language features in current IDEs there are mainly two options for integrating concept checks:

Partial Concept Checks

The IDE checks only those template instantiations whose requirements can be interpreted.

Delegate Concept Checks

The IDE uses a concrete compiler and interprets its error messages. Because compiler errors usually include positions it is possible to create markers in the editor accordingly.

Because Eclipse CDT displays compiler errors in the editor window, the second approach is already available. Another advantage of this approach is that there is no need for compiler and target platform specific configurations.

3.3. Assistance in Constrained Generic Code

Even if concept checks are not provided by the IDE it is still possible to use the additional information of the template constraints to help users write generic code. Both features presented in this section use the fact that it is often possible to derive a prototype type from constrained template type parameters.

Given the template declaration in Listing 3.1 one can derive an abstract class `X` that provides the syntax required by `Equality_comparable` and `Output_streamable` either through members or global functions. An implementation is not necessary. The IDE now can use `X` instead of the unconstrained type `T` to provide similar functionalities as if `f` takes a type deriving from `X`.

Listing 3.1: Deriving type `T` from template constraints

```
template<typename T> requires Equality_comparable<T>() &&  
    Output_streamable<T>()  
void f(T x){  
    // implementation  
}
```

The probably most usable features on derived types for writing constrained generic code are:

Highlight Unsupported Syntax

Uses of syntax that is not required from constraints and therefore not supported on the derived type `X` can be marked with a warning. Note, that the concept proposal does not state that this should result in a compilation error. Thus, while using a not declared member of a concrete type is an error, using a not required member of a derived type may be bad practice but can nevertheless result in valid code.

Code completion

Like when working with instances of concrete types it is possible to propose code completion on instances of derived types. When the user uses the `.` or `->` operators on such an instance the IDE can propose all known members through the completion menu.

In contrast to concept checks, these features can probably be implemented in a useful manner without the ability to interpret and instantiate C++ code within the IDE.

3.4. Automated Refactorings

Especially for the migration towards concepts, automated refactoring tools can be of great use for developers. This section presents some refactorings that may become common for the work with concepts:

Extract Concept from Usage

The implicit requirements of an unconstrained template can be made explicit by introducing a new concept. The requires expressions of this new concept can be derived from the syntax used within the template definition. This refactoring is already described in more details in [Sto09] for an earlier version of the concepts proposal. Please note, that the current version of the proposal also allows incremental refactoring toward concepts and does no longer require that either all requirements are explicitly described with concepts or none.

Extract Concept from Constraints

A requires clause containing a constraint expression with many constraints is probably a sign of a missing concept. An appropriate refactoring would be to create a new concept that abstracts some or all of those constraints and replace the original constraint expression with a reference to the new concept.

Propagate Constraints

Passing a template parameter to a constrained template implicitly propagates the constraints from the inner template to the outer. To clearly denote this requirement to users of the outer template it is advisable to ensure that the requirements of the outer template are at least as strict as implied from the inner one. An example of this refactoring is given in Listing 3.2.

Listing 3.2: The constraints of the template function `all_of` are implicitly propagated to the `InputIterator` template parameter

```
template<typename InputIterator>
bool allGreaterThan5(InputIterator first, InputIterator last) {
    return std::all_of(first, end,
        [](int i){ return i > 5; });
}

// after applying Propagate Constraints
template<Input_iterable<int> InputIterator>
bool allGreaterThan5(InputIterator first, InputIterator last) {
    // ...
}
```

Switch Notation for Constraints

Not all of the four possible notations for constraining templates are equally expressive and some may state the programmers intentions clearer than others. Hence, switching the notation used for constraining a template can become necessary due to additional constraints or improves the code readability. Because there is a relatively high edit distance between the various notations (e.g. rewriting a template introduction to a `requires` clause) an automated tool would heavily reduce the typing involved.

Naturally, this is only a limited overview of possible refactorings. Many well known refactorings and quick fixes that apply to classes and interfaces can also be ported to concepts. For example tools like `Implement Concept` or `Pull Up/Push Down Requirement`.

4. Parser Infrastructure of Eclipse CDT

This chapter gives an overview over the CDT parser infrastructure. It mainly focuses on the parts that have been touched during the implementation phase. This omits many further aspects of the CDT parser like scanning, preprocessing and auto-completion. For more details on these topics please refer to [Ecl11].

4.1. General Overview

A fundamental part of the CDT plugin is its ability to parse C and C++ code and to extract syntactic and semantic meaning of symbols and words in a source file. Based on this information, the IDE is able to provide services like syntax coloring, error reporting, source code navigation, code completion and many more.

In general, the IDE needs information about source files similar to that required by a compiler. Nonetheless, IDEs make quite different demands to its underlying parser infrastructure. Based on [Ecl11] we can infer the following requirements that mainly influenced the design of the CDT parser:

Performance

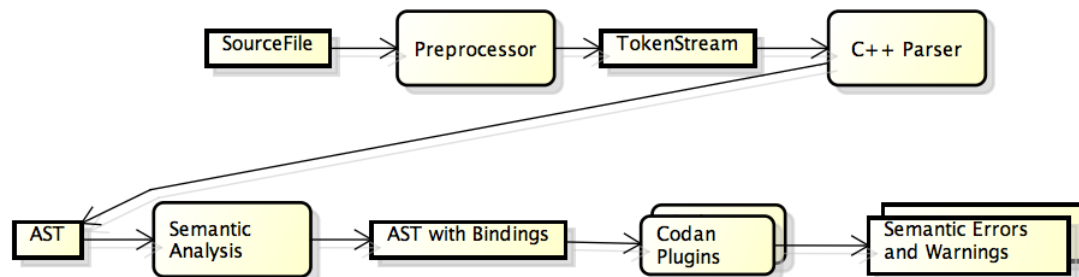
In order to achieve a responsive UI a fast parser is inevitable

Robustness

IDEs must also provide its features on only partially written source files and code with syntax errors

Figure 4.1 gives an overview of how CDT processes source files. Please note, that this is a rather idealized and high-level view on the actual process. The implementation does unfortunately often blur the border between the processing steps and mixes the responsibilities of the distinct phases. Often, we can trace these trade-offs back to the focus on performance mentioned above and the lack of a clear distinction between syntax and semantic in the C++ language specification.

Figure 4.1.: Simplified Overview over the Parsing Process



Nevertheless, we can identify several processing phases that have a more or less clear boundary to the adjacent processes. First, a source file is processed by the preprocessor which applies macro expansions and `#include` and `#ifdef` directives on the document and transforms it to a stream of tokens (lexing). Because the editor displays the unexpanded original document but the parser has to work on the expanded macros, the preprocessor must also keep track of the positions of the tokens which involves some difficulties that are not further covered in this report. The remaining phases are discussed in the next sections as they directly relate to the implementation of the Concepts proposal.

4.2. Parser

Parsing C++ is difficult. Its grammar is “ambiguous, context dependent and potentially requires infinite lookahead to resolve some ambiguities” [Wil01, 5.2] which implies that traditional parsing techniques cannot be easily applied to C++.

There are two main properties of the C++ grammar that makes it difficult to implement a parser:

1. Ambiguities that require infinite lookahead to be resolved
2. Ambiguities that depend on semantic to be resolved

The first category of problems is illustrated in Listing 4.1. Before the parser reaches the final `z++` expression, it is not clear if `int(x)` is an expression or a declaration of `x` with superfluous parentheses followed by a declaration of `y`.

Listing 4.1: A list of three expressions

```
int(x), y, z++;
```

In CDT's C++ parser these kind of problems are solved by its implementation as a hand-written recursive decent parser with backtracking and infinite lookahead. Therefore, the parser can first try one variation and backtrack if it encountered an error. Additionally, there are further optimizations in the parser implementation that allow to consider the current context. For example when parsing a declaration there apply different rules depending on whether it is in a statement, part of a parameter list or part of a template parameter.

The second category can be demonstrated by the template instantiation `T<A>`. If `A` refers to a variable it is an expression. If, on the other hand, `A` is a type the fragment is a type-id which adheres to different grammar rules. The concept proposal makes this example even harder to disambiguate because it adds a third possibility: The name `A` may also refer to a concept which would result in a template instantiation with a constrained template argument.

Some parsers, like the one used in GCC's C++ frontend [GCC], overcome this issue by using semantic information to resolve such ambiguities. The implementors of CDT chose another approach and decided to parse in such cases both variations and delay the disambiguation to the semantic analysis. The reasoning behind this decision was probably that it allows a cleaner separation between parser and the semantic resolution.

Another point that speaks for this approach are performance considerations for use cases that do not require all names in a source file to be resolved. E.g. when displaying the outline of a file it is sufficient to only resolve types used in declarations whereas function bodies are of no interest.

The implementation of the C++ parser can be found in *org.eclipse.cdt.internal.core.dom.parser.cpp.GNUCPPSourceParser*.

4.3. Abstract Syntax Tree

CDT's abstract syntax tree is a more or less accurate representation of the source code. Hence, semantically identical code fragments written using different syntax usually translate to different ASTs. For example the call `f<int>(1)` to the template function `f` is semantically equivalent to `f(1)` but the AST of the later does not include the inferred template argument.

Furthermore, the structure of the AST wont change after its creation by the parser expect for ambiguity resolution.

4.4. Bindings

CDT uses bindings to represent the particular occurrences of names in a program. Each distinct language construct that has an identifier is represented by exactly one binding which contains at least the following information:

Declarations

Locations of all declarations of this identifier.

Definitions

Locations of all definitions of this identifier.

References

Locations where the identifier is used.

Owner

The binding that “owns” this binding. E.g. a parameter is owned by its enclosing function declaration.

Scope

The scope in which this binding is declared (or defined if there are multiple declaring scopes).

Form

What is it exactly the identifier represents. Including all information necessary for checking if references to the identifier are used correctly or not.

Especially the last point should give a taste about how powerful bindings are. E.g. a binding for a function includes references to all its parameter bindings, its type, what specifier have been used to declare it and - in some cases - even how its return statement can be evaluated.

4.4.1. Binding Resolution

Every `Name` node in the AST has a `resolveBinding()` method that tries to resolve the identifier to a binding if not yet available. The binding resolution is mainly performed by `CPPVisitor.createBinding()`. This method decides whether the name is used within a declaration/definition or as a reference.

If the name is used within a declaration or definition, binding resolution creates a new binding representing the declared language construct. The binding creation

includes analyzing the declaration AST (often the declarator) and deriving the according type of the new binding.

If the name is used as a reference, binding resolution has to find an existing binding that matches the referenced name. This task is performed by `CPPSemantics.resolveBinding()`. Binding resolution also checks if the name is used appropriately or produces a binding resolution error otherwise. For example, a name used as a function reference within a function call expression that resolves to a variable binding of type `int` results in such an error.

4.5. Codan

Codan is a static analysis framework built into CDT and stands for “CODE ANalysis”. Its intention is to support plug-in developers to write lightweight checkers to find semantic errors, violation of coding guidelines or similar flaws in C/C++ code. Checkers are applied to the AST after binding resolution and produce markers with messages that are displayed by the IDE.

CDT itself already contains some checkers that are part of the `org.eclipse.cdt.codan.checkers` package. Beside the checkers that inform users of common sources of errors like assignment statements in if-conditions, there are several more that perform semantic checks not already spotted by the binding resolution algorithm. For example `AbstractClassInstantiationChecker` reports a problem when the user tries to instantiate a class with one or more pure virtual member functions.

5. Implementing Concepts Lite in CDT

This chapter covers details from the implementation and various major design decisions.

Please note, that we use an abbreviated notation when referring to AST classes in class diagrams and text. For example `RequiresClause` stands for the corresponding AST class `org.eclipse.cdt.internal.core.dom.parser.cpp.CPPASTRequiresClause` and the associated public interface if present. This is in case of `RequiresClause` the interface `org.eclipse.cdt.core.dom.ast.cpp.ICPPASTRequiresClause`. The same convention applies to references to binding classes. Thus, `VariableConcept` refers to `org.eclipse.cdt.internal.core.dom.parser.cpp.CPPVariableConcept`.

5.1. Concept Definitions

A concept definition is either a function template definition or a variable template declaration with an initializer clause whereat in both cases the `concept` specifier must be present. Additionally there are further rules that must apply to a concept definition [Sut14b, 7.1.7]. E.g. a concept must be of type `bool` or its body must consist of exactly one constraint expression [Sut14b, 14.10.7]. Listing 5.1 shows some simple concept definitions that are either correct or violate at least one rule of the specification.

Listing 5.1: Valid and invalid concept definitions

```
// valid
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2(){ return true; }

// invalid
concept bool C3 = true; // must be a template
template<typename T> concept int C4 = 1; // must be of type bool
```

```
template<typename T> concept bool C5 = 1; // must be a
    constrain-expression
```

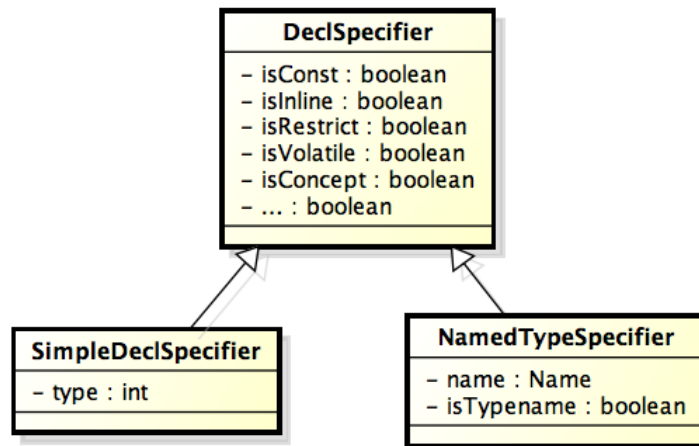
5.1.1. Parsing the Concept Keyword

The syntax for concept definitions is specified by only one additional production rule:

```
decl-specifier:
    "concept"
    ...
```

This enables the use of the `concept` specifier in declarations. Hence, everything that is required to parse concept definitions is to recognize the `concept` keyword at a specifier position. This is pretty straight forward and can be accomplished by adding the new `concept` token type to *org.eclipse.cdt.core.parser.IToken* and *org.eclipse.cdt.core.parser.Keywords*. And applying the above addition to the *decl-specifier* production rule to the `declSpecifierSeq()` method of `GNUCPPSourceParser`.

Figure 5.1.: Excerpt of the modified DeclSpecifier hierarchy



In the AST, the additional specifier is implemented as a boolean flag `isConcept` in `DeclSpecifier` nodes that indicates whether the `concept` keyword was used in the declaration. Figure 5.1 illustrates this modification.

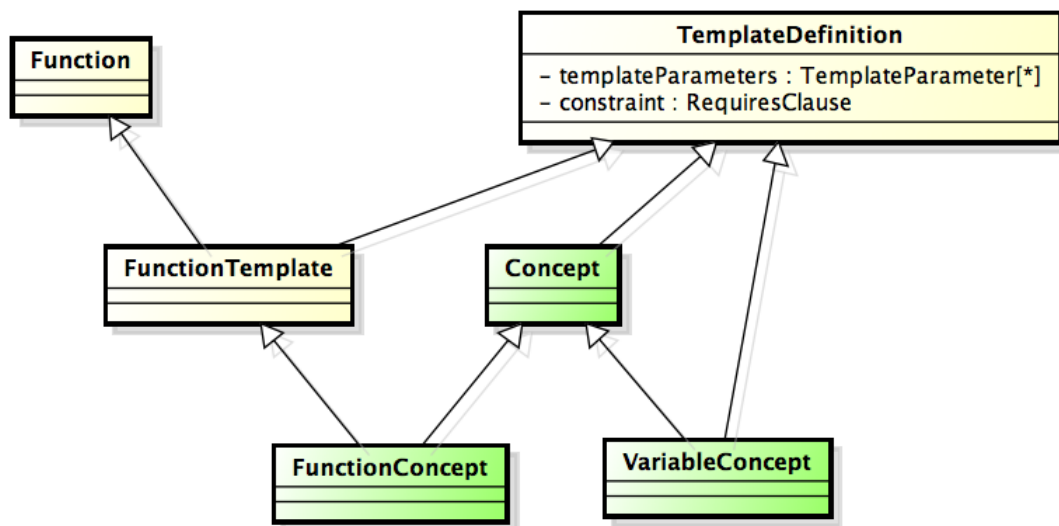
Although there are several kinds of `DeclSpecifiers` like `SimpleDeclSpecifiers` as in `const int i`; or `NamedTypeSpecifiers` as in `Person p`; the `concept` specifier is technically only valid on `SimpleDeclSpecifiers` with the type `bool`. But because we've decided to defer the check for invalid concept definitions to a later phase, it is necessary that all `DeclSpecifiers` come with the `isConcept` flag.

5.1.2. Concept Bindings

Concepts are a slightly odd language construct because they can be used both like types or like values and functions. Given the function concept `C` with one type parameter, it is used as a value in the expression `C<int>() + 1` but like a type in the parameter declaration `C c`.

This leads to the complex type hierarchy for the new concept bindings as illustrated in Figure 5.2.

Figure 5.2.: Concept Bindings Hierarchy



As a base for all concepts we use the `Concept` marker interface. `Concept` cur-

rently defines no methods on its own but inherits the interface of `TemplateDefinition`. Hence, a concept is always a template definition. The two concrete implementations of `Concept` are `FunctionConcept` and `VariableConcept`. Whereas the former directly inherits from `FunctionTemplate` and the later from `TemplateDefinition`.

Instances of `Concepts` are created during the binding resolution in `CPPVisitor` whenever it encounters a function or variable definition with the `concept` specifier.

5.1.3. Overloaded Concepts

Like with functions and function templates, it is possible to overload function concepts (but not variable concepts). Listing 5.2 demonstrates how the function `C` is overloaded three times. Once as a concept with one type parameter, once with two type parameters and once as a regular function with a non-type template parameter.

Listing 5.2: Concepts and Function Overloading

```
template<typename T> concept bool C(){ return true; };
template<typename T, typename U> concept bool C(){ return true; };
template<int I> bool C(){ return I == 0; };

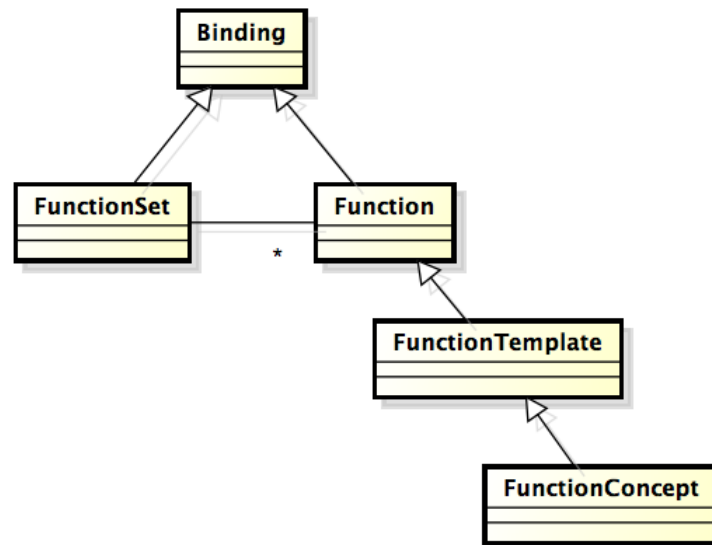
bool f(C c){
    return C<1>(1);
}
```

When the semantic analysis tries to resolve the references to `C` in the above example there are several candidates to choose from. In the case of the constrained parameter declaration `C c` the correct overload would be the first declaration of `C`.

The name resolution algorithm used in `CPPSemantics` returns instances of `FunctionSet` containing all candidates that are accessible in the lookup scope if there are multiple possibilities. A `FunctionSet` is a collection of possible bindings a name may refer to and an example of a *two-phase-binding*. Two-phase-bindings require two steps to be resolved whereas the second step is usually performed after related bindings have been resolved. In case of overloaded function names, the binding resolution algorithm must first resolve all function arguments before it is possible to resolve which overload has the best matching signature.

Figure 5.3 shows how function sets and function concepts relate in the binding structure. It is not clear from source code and documentation why `FunctionSet` does only extend `Binding` and not `Function`. This is probably an error in the

Figure 5.3.: Function Set Bindings for Managing Calls to Overloaded Functions



current implementation because other two phase bindings implement the most concrete interface that are shared by all bindings it can resolve to. Which would be `Function` in the case of `FunctionSet`. Nevertheless, function sets have to be considered when resolving names that may refer to concepts.

5.2. Requires Expressions and Requirements

Requires expressions and requirements are specified in [Sut14b, 5.1.3] and are used for defining syntactic requirements in concepts.

Listing 5.3: A Concept Consisting of Exactly one Requires Expression with a Type Requirement and a Compound Requirement

```

template<typename T> concept bool C(){
    return requires(T a, T b){
        typename A<T>;
        {a + b} -> T;
    }
}

```

5.2.1. Parsing Requires Expressions

The proposed syntax for requires expression introduces the following new production rules:

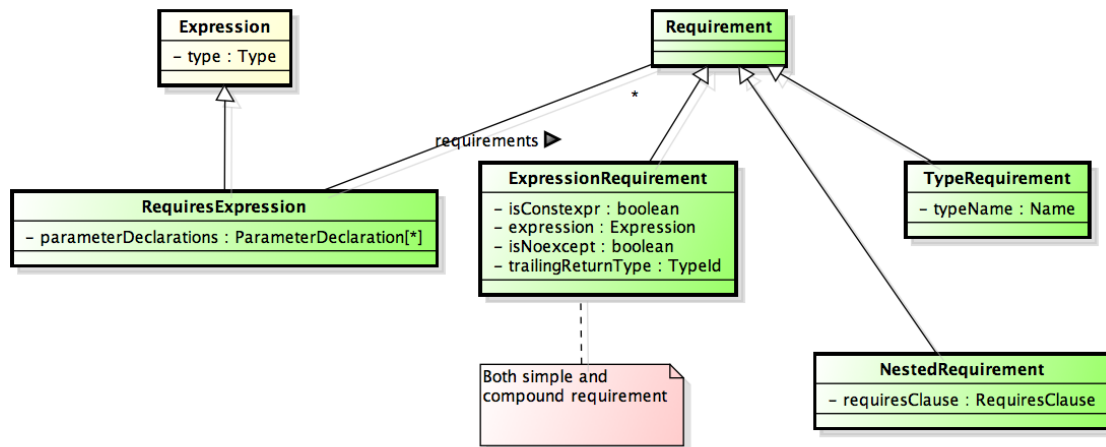
```
requires-expression:
    "requires" requirement-parameter-list requirement-body
requirement-parameter-list:
    "(" parameter-declaration-clause? ")"
requirement-body:
    "{" requirement-list "}"
requirement-list:
    requirement
    requirement-list requirement
requirement:
    simple-requirement
    type-requirement
    compound-requirement
    nested-requirement
simple-requirement:
    expression ";"
type-requirement:
    typename-specifier ";"
compound-requirement:
    constexpr? "{" expression "}" "noexcept"? trailing-return-type?
    ";"
```

Because there is no existing language construct that interferes with requires expressions and the syntax does not introduce new ambiguities, its implementation in CDT is relatively straight-forward. There is the additional `requires` keyword which is implemented analogous to `concept` in subsection 5.1.1. Furthermore, `GNUCPPSourceParser` is extended by the `requiresExpression()` and `requirement()` methods whereas both methods map the above production rules more or less as-is to its Java equivalent.

Figure 5.4 shows the AST structure that represents requires expressions. A `RequiresExpression` is an `Expression` with a list of parameter declarations and a non-empty list of requirements.

A `Requirement` is either an `ExpressionRequirement`, a `TypeRequirement` or a `NestedRequirement`. For the sake of simplicity, we've decided to map simple and compound requirements to the same type. Hence, a simple requirement is represented as an `ExpressionRequirement` with `isConstexpr` and `isNoexcept`

Figure 5.4.: AST Structure of Requires Expressions



set to false and no `trailingReturnType` (set to null).

5.2.2. Scope Lookup for Requirement Expression Parameters

Because the parameter list of requires expressions can introduce new names, each requires expression forms a new scope. This must be considered in the scope lookup mechanism in `CPPVisitor`'s `getContainingScope()` method.

Every AST node introducing a scope has an associated `ICPPScope` object that can be retrieved by `getContainingScope()`. In the case of `RequirementExpression` we decided to use the same scope implementation as used for block scopes. Because parameter lists of requirement expression can only introduce local parameters without default arguments it seemed feasible to reuse block scopes.

5.3. Requires Clauses

A require clause is the `requires` keyword followed by a constraint expression and can be used after template and function declarations. Its syntax and semantic is described in [Sut14b, 14] for requires clauses on templates and in [Sut14b, 8] on function declarations.

5.3.1. Parsing Require Clauses

The concept lite specification proposes a minor modification to support requires clauses on declarators:

```

declarator:
    basic-declarator requires-clause?
basic-declarator:
    ptr-declarator
    noptr-declarator parameters-and-qualifiers trailing-return-type

```

Whereas the optional requires clause is only allowed when the declarator declares a function.

Beside that, requires clauses may also appear in template declarations:

```

template-declaration:
    template "<" template-parameter-list ">" requires-clause?

```

Or in nested requirements:

```

nested-requirement:
    requires-clause ";"

```

A requires clause itself is defined by the following production rule:

```

requires-clause:
    "requires" constraint-expression
constraint-expression:
    logical-or-expression

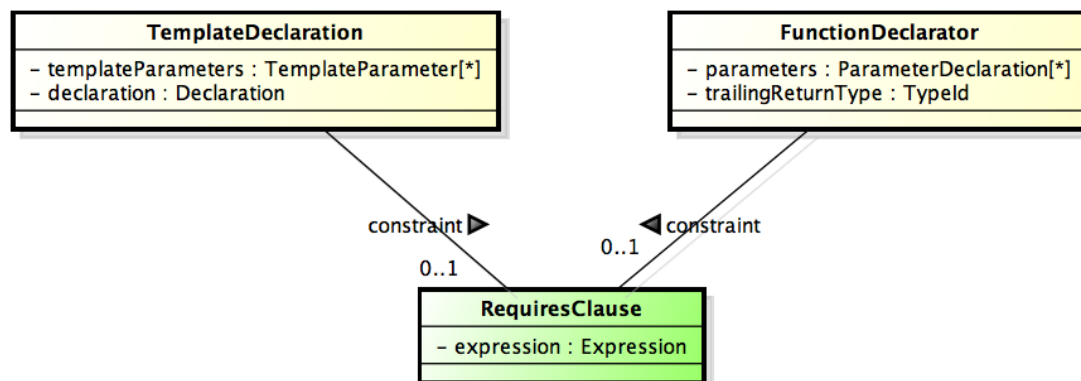
```

A `logical-or-expression` is basically an expression that can be used as the first operand of the conditional operator `?:`. In contrast to `expression` a `logical-or-expression` must not contain the assignment and comma operators. This type of expressions can be parsed analog to `constant-expression` and is implemented in `constraintExpression()` in CDT's C++ parser.

The representation in the AST for requires clauses is quite simple and illustrated in Figure 5.5. `RequiresClause` is basically just a wrapper for an expression which is its only property.

Instead of using a distinct AST node, it would also be possible to directly refer to the according expressions in the constraint properties of `TemplateDeclaration` and `FunctionDeclarator`. However, we've decided to use a distinct AST node in order to ease the traversal of constraints.

Figure 5.5.: New AST node for requires clauses



5.3.2. Persisting Constraints

Because constraints on templates and functions are part of their signature, it is crucial that requires clauses with the constraint expressions are available through the index. Unfortunately, ASTs cannot be serialized which raises the need for an alternative representation of expressions.

A possible solution for this problem is to use CDT's evaluation infrastructure which is used to evaluate `constexpr`. This functionality is only capable of representing and evaluating some parts of C++ 11 style `constexpr` and is limited when it comes to evaluating template instantiations. But there is ongoing work by Silvano Brugnoli to improve the current implementation parallel to this term project.

5.4. Handling Syntactic Sugar

Beside the core concepts of concept declarations and require clauses the concept proposal also specifies alternative syntax for defining constraints on template parameters. These alternative forms are syntactic sugar. Hence, there is an exact mapping of such an alternative form to an according requires clause and the syntax is only introduced for giving developers a more convenient way to express some common language idioms.

Another example of syntactic sugar in C++ are lambda expressions: Every lambda expression can also be written as a class with an overloaded application operator `()` whereas every captured variable is a constructor argument that be-

comes a field of the class. Lambda expressions are just an easier way to use the common language idiom of functors.

While syntactic sugar is usually easy to handle in compilers it is often a source of additional complexity in IDEs. Some features like syntax coloring and automated refactoring benefit from a bijective transformation from source code to AST. But other features, mainly semantic analysis, are simpler when they can be performed on a desugared representation of the source code. Unfortunately, desugaring is often a destructive transformation such that it is no longer possible to map AST nodes to the according positions in the source. Furthermore, desugaring often introduces new AST nodes that are not present in the original source like the functor class of a desugared lambda expression.

The reason why it is simpler to perform semantic analysis on desugared trees is mainly that there are fewer cases that have to be considered when processing the AST. For example lambda expressions must not be considered because they are already transformed to a functor.

Up to now, syntactic sugar was not a big deal in CDT because there are only a few notations that can truly be considered as syntactic sugar in C++. For example the addition assignment operator `+=` - one of the prime examples of syntactic sugar - does not exactly fall into this category: The expression `a = a + 1` is not semantically equivalent to `a += 1` depending on how the `=`, `+` and `+=` operators are overloaded.

This changes with the concept lite proposal as it introduces several equivalent ways to declare a constrained template as demonstrated in Listing 5.4. All three alternative forms in this example can be transformed to the first declaration which is using a `requires` clause. This is not only the case in this specific example but for every template introduction, constrained parameter and constrained template parameter there is an equivalent `requires` clause.

Listing 5.4: Four semantically equivalent function template declarations

```
// using a requires clause
template<typename T> requires Any<T> void foo(T x);

// using a template introduction
Any{T} void foo(T x);

// using a constrained parameter
void foo(Any x);

// using a constrained template parameter
```

```
template<Any T> void foo(T x);
```

Implementing the various alternative notations for template constraints required a fundamental decision on how syntactic sugar should be handled in this case. There were three strategies that have been considered:

Desugared AST

This would be the optional solution for semantic analysis. The AST uses only one common representation for all kinds of constrained templates. To support AST rewrites and syntax coloring, the AST must also contain information about how it can be transformed back to its original source.

Multiple ASTs

Use an AST close to the source code and desugar it further when required.

Undesugared AST and Normalized Bindings

Don't desugar the AST but create normalized bindings. Hence, in Listing 5.4 all four declarations should result in exactly the same binding.

All three options come with their disadvantages. The first two approaches don't fit well into the existing infrastructure. To directly desugar the AST the parser must know about the semantic meaning of names to distinguish concepts from types in parameter declaration. This violates the desired separation between parsing and semantic analysis. On the other hand, having multiple versions of the AST is difficult because of its design based on mutability and lazy loading. Because bindings are computed on-demand, it would be hard to ensure that they are not unnecessarily derived more than once.

The third approach is problematic because it complicates the mapping from declaration ASTs to bindings. Up till now, the form of the declaration implied the type of the binding. This is no longer true as a function declaration may become a template declaration if one of its parameters is a constrained parameter. Nevertheless, we've finally decided to take this approach. It requires the fewest changes to the existing infrastructure and fits more or less to the current AST structure.

5.5. Template Introductions

Template introductions is one of the alternative notations to constrain templates in a more concise style. On the syntax level, template introductions introduce an alternative production rule for template declarations:


```

template-declaration:
  "template" "<" template-parameter-list ">" requires-clause?
  declaration
  template-introduction declaration

template-introduction:
  nested-name-specifier? concept-name "{" introduction-list "}"

introduction-list:
  introduction-list "," introduced-parameter

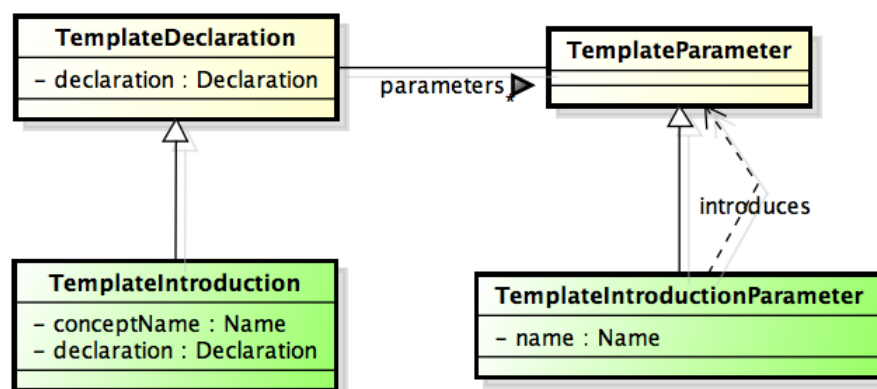
introduced-parameter:
  "..."? identifier

```

The nice aspect of this notation from a compiler builder point of view is that it is not ambiguous like constrained parameters and constrained template parameters. Therefore, the according changes to the parsers are straight forward and include only the implementation of the additional production rules.

The resulting AST is illustrated in Figure 5.6. A `TemplateIntroduction` is implemented as a specialization of a `TemplateDeclaration` and a `TemplateIntroductionParameter` is a specialization of a `TemplateParameter`. Unfortunately, this subtype relationship holds not true in all cases because the parameters field of `TemplateIntroduction` must consist only of `TemplateIntroductionParameters`.

Figure 5.6.: New AST nodes for template introductions



5.5.1. Resolve Introduced Parameters

As discussed in section 5.4 bindings for template introductions must be normalized such that they are identical to the according binding when declared using only requires clauses.

The binding creation algorithm for template introductions uses the referenced concept for deriving prototypes for each introduced parameter. These prototypes can then be used to create the effective parameter bindings.

Listing 5.5: Resolution of introduced template parameters

```
template<typename T, int I> concept bool C = true;

C{A, B} void foo(A a, Array<B> b);

template<typename A, int B> requires C<A, B>
void foo(A a, Array<B> b);
```

Listing 5.5 gives an example of how the template parameters for a desugared template introduction can be derived. The algorithm uses the following steps:

1. Find the concept with the identical name and the identical number of template parameters
2. Each template parameter of the concept becomes the prototype for the introduced parameter at the same position
3. The derived parameter declaration is identical to the prototype's declaration but it uses the name of the template introduction

5.6. Constrained Parameters

Constrained parameters are specified through `constrained-type-specifiers` and the following additional production rules:

`simple-type-specifier`:

```
...
constrained-type-specifier
```

`constrained-type-specifier`:

```
nested-name-specifier? constrained-type-name
```

```

constrained-type-name:
  concept-name
  partial-concept-id

```

```

concept-name:
  identifier

```

```

partial-concept-id:
  concept-name "<" template-argument-list? ">"

```

More specific, a constrained parameter is a parameter declaration with a constrained type specifier. This syntax is highly ambiguous as the parameter declaration `C a` may declare a constrained or an unconstrained parameter depending on whether `C` refers to a type or a concept. Hence, every parameter declaration that does not use a built-in type specifier like `int` or `float` is potentially constrained.

Figure 5.7.: New AST structure for supporting constrained parameters

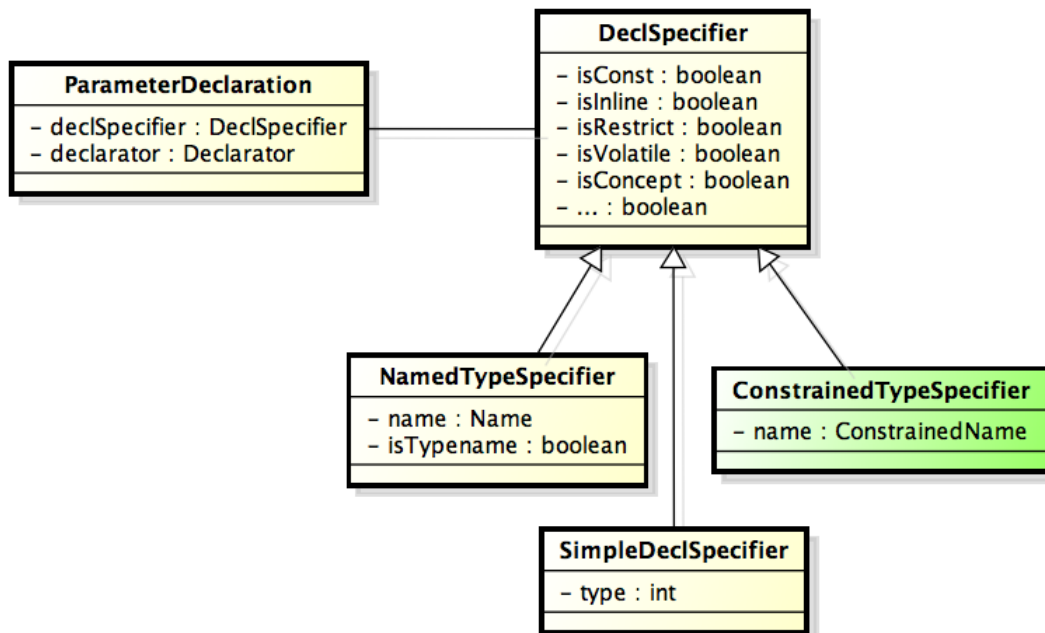


Figure 5.7 illustrates how the AST structure has been extended to support constrained parameters. The `declSpecifier` of a `ParameterDeclaration` now

can also be a `ConstrainedTypeSpecifier`.

5.6.1. Ambiguity Resolution

The ambiguous nature of constrained parameters require the use of CDT's ambiguity resolution mechanism. Disambiguation is achieved by parsing all possible alternatives and store them in an ambiguity node. Ambiguity nodes must extend `ASTAmbiguousNode` and implement the common subset of interfaces of all candidate types.

After parsing, the ambiguity resolution algorithm in `ASTAmbiguousNode.doResolveAmbiguity()` tries for each alternative to resolve all names in its subtrees. The first alternative that does not result in any resolution errors or the one with the smallest error count is then chosen as the correct replacement for the ambiguity node.

Given the additional constrained type specifier we have a new ambiguity that appears whenever a constrained type specifier is allowed. In this case, the parser creates a `CPPASTAmbiguousDeclSpecifier` including both alternatives of type `NamedTypeSpecifier` and `ConstrainedTypeSpecifier`. This new ambiguity node is also a sub-type of `DeclSpecifier`.

5.6.2. Invented Template Parameters

A function with a constrained parameter is actually a function template with one template type parameter. Listing 5.6 shows how the function `foo` with two constrained parameters `a` and `b` can be desugared to an identical declaration using the `requires` clause notation. It also demonstrates that parameters with the identical constraints share the same type parameter. Another possibility would have been to desugar `foo` to a template declaration with two type parameters one for each parameter.

Listing 5.6: Desugared constrained parameters

```
template<typename T> concept bool Any = true;

void foo(Any a, Any b);

template<typename X> requires Any<X>
void foo(X a, X b);
```

As discussed in section 5.4, the binding resolution must result in the same bindings for both function declarations in Listing 5.6. And because template bindings also include information about all template parameters we have to “invent” additional type parameters.

Invented template parameters are derived in `CPPFunctionTemplate::initInventedTemplateParameters()`. The algorithm assigns to every constrained type specifier in the functions parameter list a template parameter binding such that distinct constraints have distinct template parameters but identical constraints are associated to the same parameter.

5.7. Constrained Template Arguments in Parameters

Placeholders introduced by constrained type specifiers may also be used for template arguments of parameter types. This allows defining constraints on template arguments using the same shorthand syntax as discussed in section 5.6.

Listing 5.7 gives an example of this feature. Function `f` takes any vector whose type argument is a model of the concept `Any` (which is any type in this case).

Listing 5.7: Desugared constraint on template argument

```
template<typename T> concept bool Any = true;

void f(vector<Any> v);

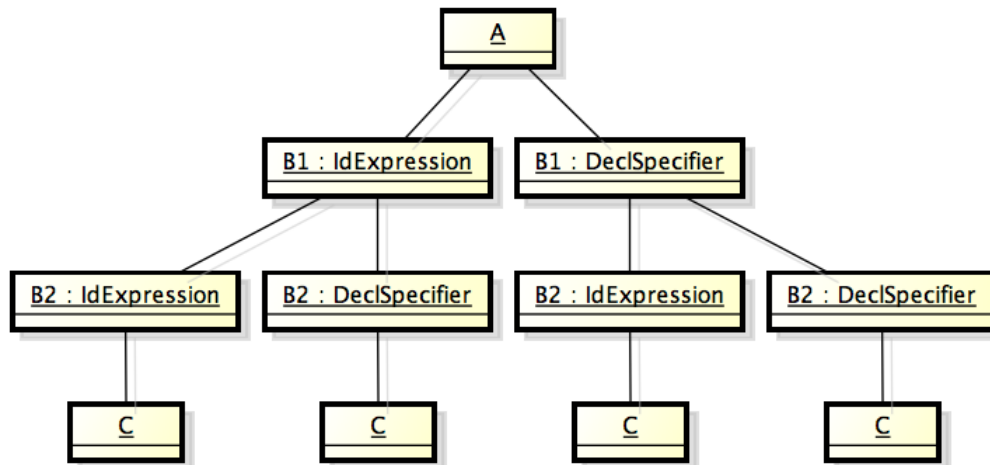
template<typename X> requires Any<X>
void f(vector<X> v);
```

5.7.1. Combinational Explosion in Ambiguous Template Arguments

CDT’s strategy of separating ambiguity resolution from parsing introduces difficulties with ambiguous template arguments. Namely, parsing nested template arguments leads to exponential growth in the number of possible disambiguations. E.g. when parsing `A<B<C>>` the name `B<C>` can be a type-id or an id-expression where `C` may be again an ambiguous fragment. Hence, parsing `A<B<B<B<B<C>>>>>` already leads to $2^4 = 16$ ambiguous trees.

Figure 5.8 gives a simplified illustration of the resulting AST when using a naive approach to parse the ambiguous expression $A\langle B1\langle B2\langle C\rangle\rangle\rangle$.

Figure 5.8.: Simplified AST of the ambiguous expression $A\langle B1\langle B2\langle C\rangle\rangle\rangle$

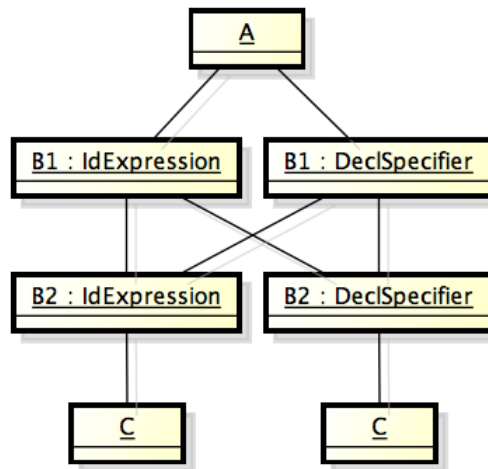


This exponential complexity in both space and time led to out-of-memory errors in Eclipse CDT prior to 7.0.1. This was especially a problem in libraries using macro expansions to generate deeply nested template arguments as discussed in [Ecl10]. This bug has been addressed by adopting the parser such that child nodes are reused between ambiguities. The improved algorithm in `GNUCPPSourceParser.templateArgument()` uses the following principles:

- Construct alternative AST nodes from already parsed nodes instead of back-track and reparse every possible alternative. This reduces time complexity to $O(n)$ where n is the nesting level of template arguments.
- Share ambiguous children between ambiguous parents as illustrated in Figure 5.9. This reduces space complexity to $O(n)$.

As it becomes evident in Figure 5.9 the output of the parser is no longer a valid AST. Some nodes are children of more than one other node which violates the definition of a tree. Furthermore, the `parent` attribute of nodes can reference at most one node which leads to unidirectional parent/child relationships.

Figure 5.9.: Improved object graph with linear size



This is why `AmbiguousTemplateArgument` uses the `beforeAlternative()` hook to fix the parent/child relationship before an alternative is processed by the ambiguity resolution algorithm (see also subsection 5.6.1).

The concept proposal introduces with constrained template arguments a further, possibly ambiguous, alternative for template arguments. Which increases the theoretical size of the ambiguity tree to 3^n when parsing template arguments in parameters. That's why we had to integrate constrained template arguments into the improved parser and ambiguity resolution algorithm and add `ConstrainedTypeSpecifier` as an additional alternative to `AmbiguousTemplateArgument`.

5.8. Constrained Template Parameters

Constrained template parameters have similar semantics to constrained parameters but can be used to constrain any kind of template arguments additionally to types. The additional production rules are:

```
template-parameter:
```

```
...
```

```
constrained-parameter
```

```
constrained-parameter:
```

```
nested-name-specifier? constrained-type-name "..."? identifier?
```

```
default-template-argument?
```

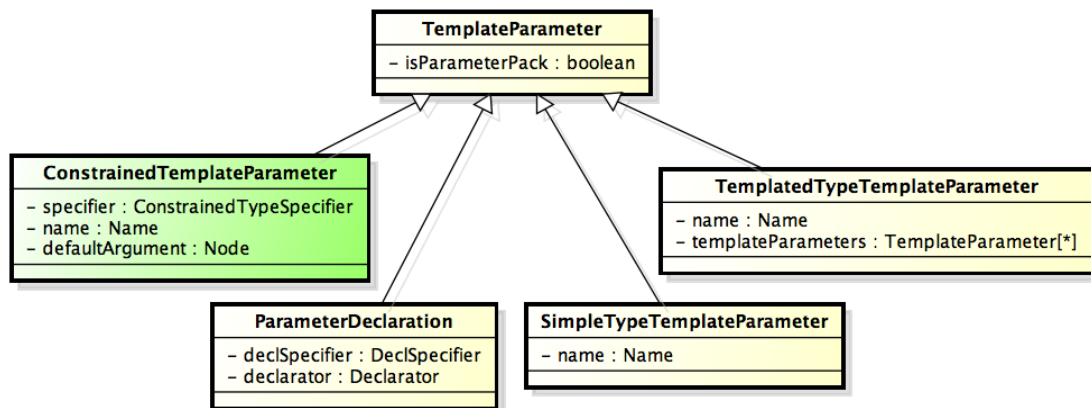
```

default-template-argument:
  "=" type-id
  "=" id-expression
  "=" initializer-clause

```

Please note, that the symbol name `constrained-parameter` is slightly misleading because these production rules do not apply to constrained parameters of functions. A probably better name would have been `constrained-template-parameter`.

Figure 5.10.: Additional AST node for constrained template parameters



The required changes to the AST are shown in Figure 5.10. Unfortunately, the implementation is based on an older version of the concepts proposal as of November 2014. This is why `ConstrainedTemplateParameter` uses a constrained type specifier instead of just the name of the concept as suggested by the production rules above. This is not technically wrong, but introduces unnecessary indirection and should probably be changed in further development.

Because the default template argument can be a type, an expression or an initializer clause the `defaultArgument` property uses the common supertype `Node` of all three possible argument types. This may be too unspecific for further implementations but was sufficient up till now because default arguments are not yet checked.

The constrained template parameter syntax also introduces an ambiguity with non-type template parameters. This ambiguity is resolved by `CPPASTAmbiguous-`

TemplateParameter nodes using the same approach as described in subsection 5.6.1.

5.8.1. Deriving Bindings for Constrained Template Parameters

While constrained parameters require “invented” template parameters for binding resolution, constrained template parameters already are template parameters but of a unspecified kind. In the template signature `template<C X>` where `C` is a concept `X` may refer to a non-type parameter, a type parameter or a templated template parameter depending on the definition of `C`.

Thus, in order to create the correct binding for a constrained template parameter, binding resolution must derive the kind of the template parameter from the according concept. Listing 5.8 gives an overview over how the kinds of constrained template parameters are derived by comparing constrained templates with their desugared equivalents. The examples illustrate that constrained template parameters are of the same kind as the first parameter of the constraining concept.

Listing 5.8: Deriving template parameter kinds from concept definitions

```
// concepts of different kinds
template<int I> concept bool C1 = true;
template<typename T> concept bool C2 = true;
template<template <typename> class T> concept bool C3 = true;
template<typename T, typename U> concept bool C4 = true;

// constrained templates with the desugared equivalent
template<C1 X> void f();
template<int X> requires C1<X> void f();

template<C2 X> void f();
template<typename X> requires C2<X> void f();

template<C3 X> void f();
template<template <typename> class X> requires C3<X> void f();

template<C4<int> X> void f();
template<typename X> requires C4<X, int> void f();
```

This behavior is implemented in `CPPTemplates.createBinding()` by copying the binding of the first template parameter of the concept with the name of the constrained template parameter.

6. Testing and Verification

This chapter describes the testing infrastructure and how the implementation has been verified.

6.1. Testing Parser and Binding Resolution

Most aspects of the implementation described in chapter 5 are covered in `org.eclipse.cdt.core.parser.tests.ast2.AST2ConceptsTests`. All of these tests use the C++ parser including binding resolution as one unit under test which leads to relatively coarse grained tests.

This test design is mainly a result of the parser architecture. Because there is no convenient way to construct and compare expected to actual ASTs, it is easier to assert the relationship between language constructs.

Listing 6.1: A test verifying the simplest form of a template introduction

```
// template<typename T> concept bool Any = true;
//
// Any{X}
// struct S{ X a; };
public void testTemplateIntroduction() throws Exception {
    IASTTranslationUnit tu = parseAndCheckBindings();

    IBinding x = findName(tu, "X").resolveBinding();
    assertInstance(x, ICPPTemplateTypeParameter.class);

    IBinding s = findName(tu, "S").resolveBinding();
    assertInstance(s, ICPPTemplateDefinition.class);
}
```

Listing 6.1 gives a simple example of how these relationships are tested. First, `parseAndCheckBindings()` parses the C++ code from the comment right above the test and checks for syntax errors and if every name can be resolved and does not

result in a resolution error. This ensures that the syntax for template introduction can be parsed (the syntax for concept definitions is checked in preceding tests).

Afterwards, we check that names resolve to bindings of the expected type. This shows that (in this case) the bindings resolution algorithm works as specified.

Splitting the checks for the parser and for the binding resolution into two distinct tests would only increase code duplication and lead to highly coupled tests.

Further tests concerning concepts can be found in `org.eclipse.cdt.core.parser.tests.prefix.BasicCompletionTest`. These tests check the proposals returned by CDT's code completion. Currently, code completion is only supported in requires expressions but further support for concepts would be possible.

6.2. Verifying with Origin

As far as we know, there is currently only one significant code base that heavily relies on the new features of the concepts proposal: The Origin library by A. Sutton et al. demonstrates how concepts can be applied to the STL. Its design is described in [SS12] and the newest version of the implementation is available at [Sut15a]. The library uses the syntax from the latest version of the concept proposal and builds with the GCC C++1z concepts branch [Sut14a].

During this project, we have periodically used the latest version of the origin library to verify our implementation. Because the missing index integration for concept bindings leads to many resolution errors that are not addressed yet, an automated verification was not yet feasible. Instead of that we performed a manual integration test.

6.2.1. Setup

For manual testing we executed our branch of CDT with the following plugins enabled:

CUTE [IFS15]

For displaying markers on binding errors. CDT does not report all resolution errors by default.

pASTa [Bru15]

For analyzing AST structures and bindings.

Afterward the current version of Origin was imported into CDT as a new project.

6.2.2. Checklist

During the manual integration tests we checked the following behavior:

Syntax Coloring

The new keywords `concept` and `requires` must be colored in the same color as other keywords.

Syntax Errors

The editor must not report any syntax errors in Origin.

Binding Resolution Errors

Names defined in the same translation units must be resolved. Hence there must be no resolution errors. Bindings across translation units are currently excluded as they may require a complete index.

Bindings

Identifiers of concepts must be correctly resolved to the according concept definition. This can be verified by using the “Open Declaration” feature of Eclipse.

Code Completion

Is code completion available for the new concept syntax?

Because the Origin library grew steadily during this project, we limited our manual checks to the files `origin/core/concepts.hpp` and `origin/sequence/algorithm.hpp`. The former defines various commonly used concepts and the later defines additional concepts and functions accordingly to STL’s `<algorithm>` header.

6.2.3. Findings

Testing our implementation with the Origin library (as of January 2015) revealed no major issues concerning performance or usability of Eclipse CDT. For example syntax coloring worked as expected as shown in Figure 6.1.

Besides the already excluded resolution errors due to the missing index we found the following issues that are not yet already fixed:

- Syntax errors when using default values for constrained template parameters

- Code completion works only within requires expressions; concept names are not proposed when writing template introductions, constrained template parameters or constrained parameters

Figure 6.1.: The Equality Comparable concept in CDT; the red marker refer to missing compiler intrinsics and functionality of the STL

```

94     },
95 }
96 // A pair of types T and U are (cross-type) equality comparable
97 // only when ...
98 template<typename T, typename U>
99 concept bool
100 Equality_comparable() {
101     return Equality_comparable<T>()
102         && Equality_comparable<U>()
103         && Common<T, U>()
104         && requires (T t, T u) {
105             { t == u } -> bool;
106             { u == t } -> bool;
107             { t != u } -> bool;
108             { u != t } -> bool;
109         };
110 }
111
112 // A type T is weakly ordered when it can be compared using the
113 // operators '<', '>', '<=', and '>='.
114 //
115 // TODO: Document semantics.
116 //
117 // Note that in a weakly ordered type, for all objects `a` and `b`

```

6.2.4. Adequacy

Naturally, using a manual approach to verify the implementation on such a small project is everything else than flawless. Especially when every marker has to be checked whether it can be traced back to a not yet implemented feature (like the persisted bindings for concepts) or it is actually an error.

Additionally, the Origin library does mainly use requires clauses and constrained template parameters for defining constraints on templates. Template introductions and constrained parameters on the other hand are only rarely or never used.

Hence, this integration tests are not meant for proving the implementation correct but rather for ensuring that there are no unexpected implications and performance bottlenecks.

7. Conclusion

This final chapter covers the results of this term project and gives an outlook on what is necessary to fully support concepts in CDT and what other developments are possible based on this work.

7.1. Accomplishments

During this project we brought basic support for concepts to Eclipse CDT and laid a foundation for more sophisticated tools based on the new language constructs. Except for a few corner cases, the achieved level of integration is as high as described in section 3.1. Thus, the IDE is able to create an unambiguous AST for most valid C++-with-concepts source code.

Our contributions to the CDT project include the following areas:

Parser

We extended the parser to support all of the new language constructs like requires clauses, requires expressions, template introductions, constrained parameters and constrained template parameters. In some cases, this required tailored solutions to handle the ambiguous parts of the language specification.

AST Structure

The main challenge during this project was to find an adequate representation of the new language constructs in the AST. The extended AST structure fits to the existing infrastructure but respects the new requirements due to the heavy use of syntactic sugar in the concepts proposal.

Binding Resolution

The binding resolution, which is also necessary for disambiguating ASTs, revealed some unexpected difficulties. For example, we had to use new approaches to handle “virtual” templates that are not explicitly defined in the source code but implied by the use of constrained parameters.

Beside these corner stones of the implementation we also enabled basic syntax coloring and auto-completion for requires expressions.

Finally, all these extensions have been applied to an existing code base that addresses an relatively complex problem: Parsing and analyzing C++ code. Beside this inherent complexity in the CDT source code, there is also a certain level of accidental complexity due to many performance considerations and several years old legacy code. This required extra caution when changing existing code to ensure that no existing features are impaired.

7.2. Remaining Tasks

In order to provide complete support for the new syntax introduced by the concept proposal there are some remaining issues not yet addressed:

Persisting Concepts

Concept bindings are not stored in CDT's index and therefore it is not yet possible to resolve concepts between translation units. This task does probably not result in a lot implementation effort but requires a detailed understanding of CDT's (un-)marshaling mechanism.

Persisting Constraints

Because constraints on template parameters are also part of the template signature they should also be persisted. This is especially crucial for advanced features like concept checks. One approach to persisting constraints may be to use CDT's evaluation infrastructure (implementations of `ICPPEvaluation`) and derive evaluation nodes from template introductions and constrained parameters to a normalized representation as requires clauses.

Default Arguments for Constrained Template Parameters

The current implementation misses the ability to use default arguments in constrained parameters. To address this issue the parser must be slightly extended. Binding resolution must then ensure that the default argument matches the kind expected by the constraining concept. Hence, a type as default argument is only valid if the constraining concept's first template parameter is a type parameter.

Fold Expressions

Fold expressions are a feature that has been added to the proposal during the last few weeks of this project and have not been considered during the

implementation. They are intended to write concepts constraining template parameter packs and introduce additional syntax. As they can be clearly disambiguated from other expressions, parsing fold expressions should be relatively straight forward to implement.

Reporting Ill-formed Requires Clauses and Concepts

Our implementation of the parser is relatively forgiving when parsing requires clauses and concepts. For example, it does not report syntax errors when encountering requires clauses calling non-constexpr functions or function concepts consisting of more than one expression. This allowed us to reuse the existing infrastructure for parsing expressions without introducing too much additional complexity. It is also arguable whether these errors should be reported by the parser or if they can be covered by an separated Codan plug-in.

Since the proposal is not yet approved it is also possible that further changes may require additional implementation effort.

A. Project Management

A.1. Tools

Eclipse 4.4 Luna

The development environment

Git For version management

Jenkins

Deployed on the project server and used during the project for continuous builds

Redmine

Deployed on the project server and used during the project for issue and time tracking

Multimarkdown

To create this document using \LaTeX as the backend

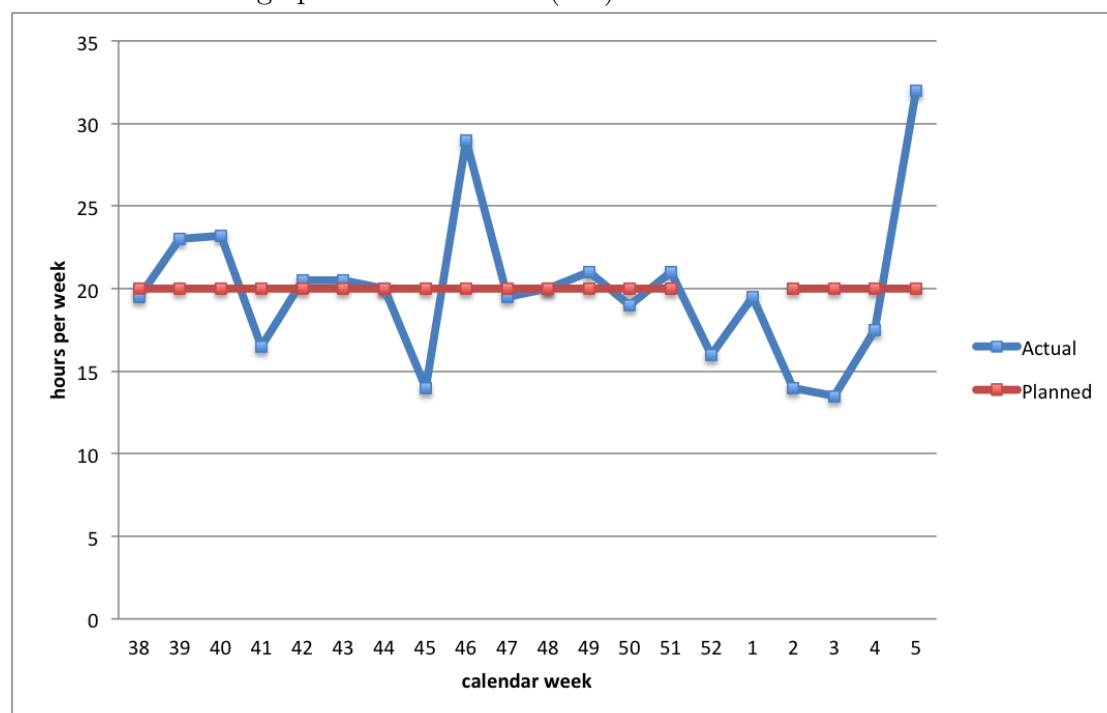
Astah

To create most of the figures

A.2. Time Report

The mandatory workload of this project is 12 ECTS which corresponds to a total of 360 hours. The project duration was initially framed to 18 weeks which gives an average workload of 20 hours per week. The sum of actual work hours was 400. The distribution of the average planned and actual workload is illustrated in Figure A.1.

Figure A.1.: Weekly summary of the reported work hours (blue) compared to the average planned workload (red)



A.3. Project Schedule

At the beginning of the project we've planned a stepwise schedule along the layers of the CDT parsing infrastructure: First design the mandatory changes for the AST, then extend the parser to accept the new syntax and then implement type checking in constrained template definitions.

Unfortunately, this schedule heavily underestimated the complexity of the CDT parser and the characteristics of the new language features. Furthermore, it turned out that the initially scheduled proceeding was too sequential (waterfall). Providing a practical design for the whole AST was not possible without a detailed understanding of CDTs internal processes.

Instead of that we then followed a more iterative approach: Implement more or less isolated features vertically through all layers one after another. This approach is also nearer to the parser's architecture as there is no clear distinction between decoupled phases. Furthermore, it allowed to quickly react to unexpected pitfalls.

B. Bibliography

- [Bru15] Silvano Brugnoli. pASTa. <https://github.com/silflow/pASTa>, 2015.
- [Ecl10] Eclipse CDT. Bug 316704 - [C++ Parser] Exponential complexity resolving template argument ambiguity. https://bugs.eclipse.org/bugs/show_bug.cgi?id=316704, 2010.
- [Ecl11] Eclipse CDT. Overview of Parsing. https://wiki.eclipse.org/CDT/designs/Overview_of_Parsing, 2011.
- [GCC] GCC. GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>.
- [IFS15] IFS. CUTE. <http://cute-test.com/>, 2015.
- [ISO98] ISO/IEC. C++Standard-ISOIEC14882-2003. *Office*, 1998:757, 1998.
- [ISO11] ISO/IEC. Working Draft , Standard for Programming Language C ++. 2011.
- [ISO13] ISO/IEC. Programming Languages — C ++. (30), 2013.
- [SS12] B Stroustrup and A Sutton. A Concept Design for the STL. *ISO/IEC JTC1/SC22/WG21 . . .*, 2012.
- [SSR13] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts Lite. pages 1–56, 2013.
- [Sto09] Mirko Stocker. Seminar Program Analysis and Transformation Concepts for C ++. pages 1–20, 2009.
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. 1st edition, 1994.
- [Str09] Bjarne Stroustrup. The C++0x ”Remove Concepts” Decision. *Dr.Dobb’s Journal*, 2009.

APPENDIX B. BIBLIOGRAPHY

- [Sut14a] Andrew Sutton. GCC C++1z Concepts Branch. [svn://gcc.gnu.org/svn/gcc/branches/c++-concepts](https://svn.gnu.org/svn/gcc/branches/c++-concepts), 2014.
- [Sut14b] Andrew Sutton. Working Draft , C ++ extensions for Concepts. 2014.
- [Sut15a] A Sutton. Origin C++11 Libraries. <https://github.com/asutton/origin>, 2015.
- [Sut15b] Andrew Sutton. Technical Specification: Concepts. <https://github.com/cplusplus/concepts-ts>, 2015.
- [Wil01] ED Willink. *Meta-Compilation for C++*. PhD thesis, 2001.