JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 8, No. 5, September–October Z

A Semantic Definition of Separate Type Checking in C++ with Concepts

Marcin Zalewski and **Sibylle Schupp**, Chalmers University, Sweden

We formalize the informal definition of C++ *concepts* that is currently considered by the C++ standardization committee for inclusion in the next version of the language. Our definition captures the basic semantics of separate type checking, where *concept-constrained templates* are checked separately from their uses and comprises of three main parts: non-standard name lookup, type checking of constrained templates, and *implementation binding* in concept maps. The formalization reveals two possible problems in the informal definition: hiding of names is not respected and incompatible implementations can be bound to concept entities. Furthermore, our definition allows formulating intuitively correct code that is rejected by the informal specification.

1 INTRODUCTION

Generic libraries in C++ have long relied on *templates*, pieces of code parameterized by types and values, to implement reusable data structures and algorithms. Such generic code can be applied to families of types; in C++, application of generic code requires *instantiation* of templates at compile time. While the families of types to which generic libraries can be applied are infinite, types in each family share the characteristics specific to the domain *concept* that these types represent.

A concept, then, can be considered, on the one hand, as a predicate on types, and, on the other hand, as a specification of an abstract data type. Indeed, concepts have long been used in C++ generic libraries, beginning with the Standard Template Library [15], to document the syntax and semantics required from template parameters. True to the specification aspect of concepts, the origins of this tradition can be traced back to the algebraic specification language Tecton [11]. Since concepts have become an integral part of generic programming in C++, direct support for concepts in the C++ language has been proposed by several authors [6, 18]. Concepts are now de facto accepted (by the C++ ISO standardization committee [13]) into C++0x, the next revision of C++, which is expected to be completed by the end of this decade. As a language construct, concepts allow for *separate type checking* where polymorphic, generic code (templates) and concrete types are checked separately, against the contract represented by concepts. Then, at the point of *template instantiation*, when the concrete types are substituted for template parameters, a compiler only has to verify that the contract was checked earlier; if it was, the template is guaranteed to work correctly (the guarantee may be weakened to permit optimizations, see [9]). The definition of concepts for C++0x is given in natural language [7], in the style of the C++ standard. The wording of the definition takes about 80 pages alone, and the complete C++ language definition spans several hundreds of pages.

Our main contribution is a formal definition of separate type checking with concepts, with semantic judgments defined by inductive relations over the abstract syntax of a language that captures the core functionality of concepts. Much of our semantic definition treats the interaction of name lookup and implementation binding with *concept refinement*, a mechanism that facilitates construction of concept hierarchies. Name lookup in concepts differs from the usual lookup rules in C++; for example, function overload sets, which are typically limited to the scope of one syntactic entity, can span multiple concepts in a concept hierarchy. The implementation binding mechanism is entirely new to C++ in that implementations are entirely separated from the types for which they are defined. Furthermore, implementation binding may be non-local—the refinement relation propagates implementations automatically, creating *implicit concept maps*. Every concept map introduced implicitly must be *compatible* with the concept maps introduced previously; this condition is quite difficult to define and is different from other mechanisms of C++. Our formalized account of these features enables one to clearly understand all intricacies of the design, disambiguating the natural language wording and, also, making hidden design choices visible. In addition, our semantic rules may serve as a starting point for a compiler implementer and provide a basis for a future, formal separate type checking safety proof.

The premise of separate type checking with concepts is that once constrained templates are checked against their *requirements* and concept maps are well-formed, templates can be safely instantiated *without* any additional type checking, provided that all necessary concept maps have been defined. The checking of templates against constraints and the binding of implementations in concept maps is captured, in our formalization, by the *program instantiation* judgment, which relates a program to an environment if and only if the program is correct. This judgment is the main premise of an extended version of the separate type checking safety theorem proposed by Dos Reis and Stroustrup [18] as a future work task for the C++ concept system. Program instantiation and the safety theorem are described in detail in section 4 and the complete set of semantic judgments is available in the accompanying technical report [31]. We introduce C++ concepts informally in section 2, supporting the discussion with code examples.

Notation

The semantics of concepts given in this paper is based on the latest version of the concept wording at the time of writing [7] (available online at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/), hereon referred to simply as "the wording." Moreover, we refer to parts of the wording by its sections and paragraphs, for instance, "clause 3.3.10 para. 1" for the first paragraph of section 3.3.10. The C++0x language extended by the wording is referred to as ConceptC++ and the current version of the language is simply C++. The language used for our formalization is referred to as X++.

2 CONCEPTS, NAME LOOKUP, AND IMPLEMENTATION BINDING

In C++, a template is either a class or a function parameterized by types (also by values but we do not discuss that here). When a template is defined, the compiler postpones all but syntactic correctness checks for the parts of the template that depend on type parameters. At instantiation time, the template is type-checked again with type arguments substituted for parameters. Since full type checking does not occur until instantiation time, the developers and the users of generic libraries must agree on the semantics and the syntax that template arguments must provide to guarantee the correct operation of generic code when applied to these arguments. In the current C++ practice the agreement is described in the documentation of a library where *syntactic* and *semantic* requirements of each template are listed. Concepts for C++ provide direct support for expressing and checking of requirements of templates within the language. Simply, concepts enable *separate type checking* of the generic template code and of the particular types to which the generic code can be applied.

In the remainder of this section, we provide an informal introduction to separate type checking, including the details of refinement, name lookup, and implementation binding. The discussion is supported by code examples that comply with the language specification in the concept wording [7].

Introduction to Separate Type Checking

Listing 1 shows a simple example, illustrating the basics of separate type checking. Two template functions, adding two values of an addable type, are defined, one is a usual, *unconstrained* C++ template (lines 14–15) and the other is a *constrained* template with a `requires` clause (lines 17–18). The unconstrained template returns a value of type `T::result_type` and, consequently, requires that types with which it is used have a member type `result_type`. Since nothing is known about the type parameter `T` and its members, the `typename` keyword must be placed before the name `T::result_type` to explicitly inform the compiler that the name will designate a member type. Then, two values `a` and `b` are added and the result is returned. At this point, a compiler cannot check that the addition produces a result of the expected type `T::result_type` because the definition of the addition operation depends on template arguments and can only be looked up when the template is instantiated. The constrained template is very similar but it requires that its arguments model the `Addable` concept. The concept, defined on lines 5–8, has two requirements, one for an associated type `result_type` and one for an associated function `operator+`, which takes two arguments of the addable type `T`. The constrained template is checked at the time of its definition, against the syntax introduced in the `Addable` concept. Since `Addable` provides the associated type `result_type` and the associated `operator+` that returns this type, the template type-checks. In difference to the unconstrained template, `result_type` does not have to be preceeded with the `typename` keyword or qualified by the parameter `T` because it can be looked up in the `Addable` concept.

When the unconstrained template is called on lines 21–22, the types of the arguments

```
 1 struct NotNumber {
 2   typedef NotNumber result_type;
 3   result_type operator+(NotNumber) { return result_type(); }
 4 } a, b;           // declare a and b of type NotNumber
 5 concept Addable<typename T> {
 6   typename result_type;
 7   result_type operator+(T, T);
 8 }
 9 concept_map Addable<int> {
10   typedef int result_type;
11   result_type operator+(int a, int b) { return a+b; }
12 }
13
14 template<typename T>
15 typename T::result_type plus(T a, T b) { return a+b; }
16
17 template<typename T> requires Addable<T>
18 result_type cplus(T a, T b) { return a+b; }
19
20 int main() {
21   plus(1, 2);     // error, int has no result_type
22   plus(a, b);
23   cplus(a, b);    // error, no Addable<NotNumber> concept map
24   cplus(1, 2);
25 }
```

Listing 1: Type checking of constrained and unconstrained templates

are substituted for parameters and the template is type-checked. In the first call, the template called with two integers fails to instantiate, because `int`, a built-in type, does not have a member `result_type`. Even worse, there is no way to add such member to a type that has already been defined or to add it at all to built-in types such as `int`. On line 22, `plus` is called with two arguments of type `NotNumber`, which, as its name suggests, is not a number. Incidentally, however, `NotNumber` matches the syntactic requirements of the `plus` template and the instantiation succeeds even though the addition operator does not have the desired semantics. The constrained template has the exactly opposite behavior. When, on line 23, `cplus` is instantiated with arguments of type `NotNumber`, a concept map `Addable<NotNumber>` does not exist and the instantiation process is halted with a corresponding error. For `int`, however, an `Addable<int>` concept map is defined on lines 17–18. This map provides implementations for the associated names of the concept `Addable`; in particular, `result_type` is bound to `int` and `operator+` to integer addition.

The process of separate type checking consists, in summary, of 3 major steps:

1. Checking of *constrained templates* in the context of *concept requirements* on their type parameters;
2. *implementation binding* in *concept maps* checked against the signatures in corresponding concepts;
3. instantiation of templates that were successfully type-checked with arguments for which appropriate concept maps exist.

Type checking of constrained templates, thus, is essentially a name lookup problem; the constraints on the template parameters establish an environment in which names used in a constrained template are looked up. Once the names are found, a template body is checked as if the template parameters were usual types. In the second step, particular implementations are bound to the names provided by concepts. Name lookup and implementation binding must match; for example, if an associated function is chosen between duplicate signatures, these signatures must later have identical implementation so the semantics of a program does not depend on the order of definitions.

Refinement, Name Lookup, and Implementations Binding

In the example in listing 1, there was only one concept and only one concept map but generic libraries are always more complex. Instead of giving many large concepts, generic libraries are founded upon conceptual hierarchies in which concepts *refine* other, simpler concepts. The semantics of name lookup and implementation binding is complicated by refinement. Name lookup in constrained templates must be performed throughout refinement hierarchies. Implementation bindings can be made at different points of these hierarchies, requiring implementation compatibility checking and propagation.

Name Lookup

Name lookup in constrained templates applies to names of types and functions. We first discuss the two independently, in listings 2 and 3, and then, in listing 4, give an example of how the two lookups interact.

```
concept A<typename T>              { typename t; }
concept B<typename T> : A<T>       { typename t; }
concept C<typename T>              { typename t; }
concept D<typename T> : B<T>, C<T> { }
concept E<typename T> : A<T>, C<T> { }

template<typename T> requires D<T>
struct Test1 { typedef t test1;         typedef A<T>::t test2;
               typedef B<T>::t test3;   typedef C<T>::t test4;  };

template<typename T> requires E<T>
struct Test2 { typedef t test1;                                 };
```

Listing 2: Name lookup of associated types

In listing 2, a refinement hierarchy is formed where concept `B` refines concept `A`, concept `D` refines concepts `B` and `C`, and concept `E` refines concepts `A` and `C`. Each of the concepts `A`, `B`, and `C` introduces a requirement for an associated type `t`, while concepts `D` and `E` do not introduce any new requirements but only refine other concepts. Two template structures demonstrate how name lookup proceeds. The type parameter of the first structure, `Test1`, is constrained by `D`. In definition of `test1`, which simply aliases the name `test1` to `t`, `t` is looked up in the scope of requirements, specifically in `D<T>`, which is the only requirement. In D there are three types named t, from A, B, and from C, but the t of A is *hidden* by the t in B. Still, there are two ts to choose from, in B and C. In our semantic definition (section 4), we state that the two ts must be the same in this case, although the concept wording [7] requires that for the lookup to succeed without ambiguity, the equivalence of the two types must be declared explicitly, through compiler-supported concepts. Our decision to assume the equivalence of types stems from a fix to an error that we have discovered in the wording—we discuss the details in section 5. In cases like this, where types are equivalent, a representative still has to be chosen as the lookup result; the wording proposes the one found in the depth-first, left to right traversal of the refinement hierarchy in the order prescribed by the syntactic order in refinement lists. Consequently, test1 becomes an alias for the B<T>::t. In the remaining definitions, t is looked up in the qualifying scopes; B<T>::t and C<T>::t are the same type since they are both visible and in the same refinement hierarchy, but A<T>::t may name a different type since it is hidden by B<T>::t. In the second template, Test2, constrained by the requirement E<T>, there are two ts visible, one in A and one in C. Since there is no hiding, t in the definition of test1 resolves to A<T>::t because of the depth-first traversal strategy (note that A<T>::t and C<T>::t are the same type).

```
concept A<typename T>              { void f(); }
concept B<typename T>              { void f(); }
concept C<typename T> : A<T>, B<T> { void f(); }
concept D<typename T> : A<T>, B<T> { }

template<typename T> requires C<T>
void f1() { f(); }  // C<T>::f

template<typename T> requires D<T>
void f2() { f(); }  // A<T>::f

template<typename T> requires A<T>, B<T>
void f3() { f(); }  // error, ambiguous
```

Listing 3: Name lookup of associated functions

Listing 3 gives an example of function name lookup in the presence of refinement. Two concepts, A and B, have an associated function void f(). The concepts C and D both refine A and B but C introduces its own associated function f. In the template f1 constrained by C, the functions from A, B, and C are visible and f from C is selected. Although incidentally the result of the name lookup is very similar to that in inheritance, name lookup proceeds entirely differently. The refinement hierarchy is traversed, starting at C, and an overload set that contains all functions named f found in the hierarchy is constructed—this is different from class hierarchies where name lookup may only result in overload sets containing names from a single class (clause 10.2 of the C++ standard). The overload set may contain functions with identical signatures, declared in different concepts in the refinement hierarchy; these signatures are assumed to refer to the same *specification* of a function. Consequently, rather than causing ambiguity, one of the identical signatures is chosen as a unique representative while the others are removed from the overload set. As in type lookup, the strategy for picking representatives is to pick the signature found first in a depth-first traversal of a refinement hierarchy and `C<T>::f` is thus chosen. In the template `f2`, the situation is similar and `A<T>::f` is chosen. In the template `f3`, there are two requirements, `A<T>` and `B<T>`. In this case, the two signatures `A<T>::f` and `B<T>::f` are visible, but since they are not part of the same refinement hierarchy, as in `f2`, it can be no longer safely assumed that the two signatures represent the same function and the lookup results in ambiguity.

```
concept A<typename T>        { typename t; void f(t); }
concept B<typename T> : A<T> { typename t; void f(t); }

template<typename T> requires B<T>
void f1(A<T>::t a, B<T>::t b) { f(a); f(b); }  // calls A<T>::f and B<T>::f

template<typename T> requires B<T>, A<T>
void f2(A<T>::t a, B<T>::t b) { f(a); f(b); }  // error, ambiguous
```

Listing 4: Name lookup of associated types and functions

The last example of name lookup, in listing 4, shows how lookups of associated types and functions interact. The refinement hierarchy is simple, concept `B` refines concept `A`. Both concepts require an associated type `t` and an associated function `void f(t)`. The `t` in `B` shadows the `t` in `A` and the two functions thus have different signatures because the two `t`s are not equivalent. The function template `f1` takes two arguments of types `A<T>::t` and `B<T>::t` and then calls function `f` with each of the arguments. In the first case, `A<T>::f` is called and in the second `B<T>::f`, as a result of overload resolution with both functions in the overload set and the correct calls chosen based on the arguments. The second template, `f2`, adds `A<T>` to the list of requirements. Name lookup is performed in each of the requirements and `A<T>::f` is discovered twice, causing ambiguity in the call `f(a)` but the second call succeeds since there is only one `void f(B<T>::t)` available. In a conceivable alternative design, the second requirement for `A<T>` could be "merged" with the same requirement introduced through `B<T>` but this is not the semantics in the current concept wording [7].

Implementation Binding

Concepts provide an environment of function and type names. In the previous section, we have shown how this environment is used to check constrained templates. Next, we show how particular implementations are bound to the names provided by concepts. For convenience, only some implementations have to be provided at certain points of the refinement hierarchy; these are then propagated along the refinement relation. At the same time, the compiler checks that no entities have multiple, possibly incompatible, realizations, and that no entities are left unimplemented. Consequently, the main parts of implementation-binding semantics are *implicit concept map generation*, which propagates implementations, and *compatibility checking*, which ensures there are no conflicting implementations.

Listing 5 shows how implementations are bound to associated function requirements. Two concepts, `A` and `B`, each require an associated function `void f()`. Concept `C` only

```
1  concept A<typename T>              { void f(); }
2  concept B<typename T>              { void f(); }
3  concept C<typename T> : A<T>, B<T> { }
4
5  concept_map A<int> { void f() { return; } }
6  concept_map B<int> { void f() { return; } }
7  concept_map C<int> { }  // error, two implementations for f
8
9  concept_map C<bool> { void f() { return; } }
10 // concept maps for A<bool> and B<bool> implicitly generated
11
12 concept_map A<float> { void f() { return; } }
13 concept_map C<float> { }  // error, no implementation of f to push to B<float>
```

Listing 5: Implementation binding for functions

refines `A` and `B` without adding any new requirements. On lines 5–7, concept maps for `A<int>` and `B<int>` are defined, each binding an implementation to function `f`. Refining concepts must provide definitions for all the requirements from all the concepts they refine but they can do so *explicitly* or *implicitly*. An implicit definition "pulls" definitions from refined concepts and an explicit one "pushes" to refined concepts. The concept map for the refining concept `C` on line 9 defines the function `f` explicitly, while the concept maps on lines 7 and 13 make the definition implicit. When a concept map for `C<int>` is defined, the implementations from its refined concepts are propagated and there are two implementations for `f`. Even though the two implementations are obviously the same, the wording does not force compilers to compare functions because, in general, such comparison is not feasible. Consequently, the concept map for `C<int>` fails. The concept map for `C<bool>`, on line 9, includes a definition for `f` although it is not required by the concept `C` itself; this definition is used to implicitly generate maps for `A<bool>` and `B<bool>`. Finally, the concept map on line 13 fails because it does not contain an explicit definition to push to its refined concept `B` and the implementation from `A<float>` seen on line 12 is not allowed to be pushed "sideways" to `B<float>`. Implementation binding for associated types is very similar, except that more than one definition is allowed since it is easy to check whether two type names reference the same type. Thus, concept maps such as those on lines 5–7 would be accepted for types if both definitions were to the same type.

3 FORMALIZATION

Our formalization is carried out for a subset of ConceptC++, comprising concepts, templates, and concept maps; for ease of reference we name the language X++. In this section we discuss the definition, conventions, and the ConceptC++ constructs included in X++. In the next section, we give a semantic definition of separate type checking.

Definition and Conventions

X++ is defined using Ott [20], a tool for a "working semanticist." The metalanguage of Ott is used to define the syntax of X++ and the semantic judgments given as inductive relations over the syntax. From that metadescription, Ott can generate LaTeX output as well as formal definitions for Coq, HOL, and Isabelle/HOL. Our semantics of X++ comprises 62 judgment rules, corresponding to 62 relations, with the total of 229 rule clauses specifying those relations. In this paper we show only a fraction of the rules, the complete definition is available in the accompanying technical report [31].

The definition of X++ is checked by Ott: the grammar is parsed and sort-checked, inconsistencies in judgment forms and metavariable naming conventions are prevented, and bounds of list forms are checked. Sequences are indicated by $\overline{\text{overline}}$: $\overline{x}$ is a sequence of xs, $\overline{x_i}^{\,i}$ is the same but with indexes, $\overline{\overline{x}_i}^{\,i}$ is a sequence formed by concatenating a number of sequences of $\overline{x}_i$, $\overline{x_i\overline{x}_i}^{\,i}$ is a sequence formed by prepending an element x_i to a sequence $\overline{x}_i$ and then concatenating the resulting sequences. Formulas can also be sequenced; for example, $\overline{x_i = y_i}^{\,i}$ compares each x_i and y_i from some sequences $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$. Syntax printed in `monotype` signifies terminals representing C++ syntax. Every sequence may be treated as a set when necessary. Some rules use function syntax to indicate that in principle they could be written as functions rather than as relations. Grammar rules can be annotated with an M for metarules. Metarule annotation causes Ott not to generate constructors for the particular rule in a theorem prover output and not to consider this rule when parsing example input. The [I] annotations signify that a rule is not part of the user syntax and is only used on the metalevel in the semantic rules. Finally, the annotation S states that a rule is a metarule but it can be parsed in concrete syntax.

In the metatheory, we use *options*, indicated by adding a question mark to an identifier. For example, *cid*? can either be a *cid* or *None*. Given a list of options, the function *rm*? removes all the *None* options; for example, $rm?(\overline{cm?})$ gives a list $\overline{cm}$ with all *None* options removed. In some rules we use set comprehension, where $\{x \in y \mid \cdots\}$ gives all elements x in y that fulfill some condition. Set comprehension is assumed to be an operation on sequences that removes all elements that do not fulfill the comprehension criterion and all duplicates.

Errors are not handled explicitly. Semantic judgments capture the expected behavior of a compiler only for a correct program. An ill-formed program will simply cause one to get "stuck" when searching for a derivation tree of a semantic relation.

X++ Language

Figure 1 shows the syntax of X++ expressions and programs, while figure 2 lists the grammar for the three basic constructs of X++ (for the complete grammar, see [31]): concepts [7, clause 14.9.1], concept-constrained templates [7, clause 14.10], and concept maps [7, clause 14.9.2]. An X++ program P comprises definitions of concepts, templates, and concept maps. The syntax of a user program, which is omitted in figure 1, is denoted by

e ::=			expressions
|	var		variable
|	`obj` τ		object
|	$f\,(\overline{e_i}^i)$		application
|	v_{int}		integral
|	v_{bool}		boolean
|	(e)	S	parenthesis

P ::=			program
|	d	M	one
|	$\overline{P_i}^i$	M	flatten
d ::=			definitions
|	cp		concept
|	tl		template
|	cm		concepts map
|	icm	[I]	implicit map

Figure 1: Grammar of expressions and top-level program syntax

cp ::=			concept
|	con `refines` $\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$		def.
tl ::=			template
|	`template` $req\; tn\,\{\,e\,\}$		def.
req ::=			requirements clause
|	`requires` $\overline{cid_i}^i$		def.
cm ::=			concept map
|	`concept_map` $cid\,\{\,\overline{tydef}\;\overline{fdef}\,\}$		def.
|	$icm\,.\,cm$	M[I]	implicit concept map

Figure 2: Grammar of concepts, templates, and concept maps

P_{user} and is the same as the syntax of a program in figure 1 except that it does not include implicit concept map definitions (the *icm* production for *d*). Expressions are kept simple and the syntax is self-explanatory.

In none of the examples in section 2 did type parameters in templates and concepts play a role and accordingly X++ excludes those parameters. Although some parts of the semantics of concepts cannot be formulated without concept and template parameters, the name lookup and implementation binding semantics we consider is independent of a particular choice of template arguments.

A concept consists (figure 2) of a concept name *con* followed by a refinement clause, which contains a list of concept identifiers, $\overline{cid}$, followed by the concept body. A concept identifier, *cid*, represents a concept instantiated with some arguments, and in X++ is simply just a concept name *con*. For example, in the refinement "`B<typename T> : A<T>`," concept `B` refines a concept instance `A<T>`. In X++, the difference between a concept name *con* and a concept identifier *cid* is that a concept name is used in a definition of a concept and the concept identifier is used to refer to that concept. While in the concept wording [7], refinement is specified as a relation between concepts, in X++, refinement is a relation either between concepts and concept instances (in concept definitions) or between concept instances (in concept uses). Direct refinement ($\prec_1$) and its transitive closure ($\prec$) is described by the following 2 rules:

$$\frac{1.\, refined\,(C, cid) = \overline{cid}}{C \vdash cid \prec_1 \overline{cid}} \qquad \frac{1.\, C \vdash cid \prec_1 \overline{cid_i}^i \qquad 2.\, \overline{C \vdash cid_i \prec \overline{cid}_i}^i}{C \vdash cid \prec \overline{cid_i \overline{cid}_i}^i}$$

The rules should be interpreted as an inductive relation over abstract syntax, where, for each fraction, top and bottom are related. In the context of programming languages, the rules can be read top-to-bottom, considering only one direction of the relation: the simpler premises on the top give a semantic conclusion on the bottom. In our presentation we number premises for easy reference. The first rule determines the meaning of direct refinement denoted by $\prec_1$. The sole premise is that the *refined* function, defined elsewhere, returns a sequence of concept identifiers $\overline{cid}$ given a list of defined concepts C and a concept identifier cid. The conclusion states that cid directly refines $\overline{cid}$ in the context of C (indicated by the turnstile). This rule is the base case of refinement. The second rule defines the transitive closure $\prec$ of direct refinement $\prec_1$. The first premise extracts the list of directly refined concepts, containing i elements, and the second premise applies the refinement relation to each one of the directly refined concepts obtaining i new lists $\overline{cid}$. This is repeated recursively, since the second premise is the refinement ($\prec$) relation itself, until the leaves of the refinement tree, i.e., concepts with empty refinement clause for which $i = 0$, are reached. In the conclusion, all the results are concatenated together giving a list of all refined concepts. An important property of the refinement relation definition is that the syntactic order of concept identifiers in concept refinement clauses is preserved. This property is not apparent from the semantic rules we have listed because it depends on the function *refined*, which extracts a list of concept identifiers from a refinement clause preserving their syntactic order (see [31]). The list of refined concepts, $\overline{cid_i\,\overline{cid_i}}^{\,i}$, in the transitive closure rule, preserves the depth-first, syntactic order in which concepts were visited, since every concept identifier is followed directly by the ordered list of concept identifiers reached from it in the traversal.

A concept body contains (figure 2) concept requirements comprising associated types $\overline{ty^a}$ and associated functions $\overline{f^a}$. In ConceptC++, the requirements can be freely intermixed but for simplicity, X++ requires that associated types precede associated functions. A template (figure 2) consists of a requirements clause, a template name tn, and a template body with, for simplicity, a single expression. The requirements clause, next, consists of concept identifiers, each representing a requirement that must be fulfilled by a corresponding concept map. In X++, there is no distinction between function templates (e.g., `cplus` in listing 1) and class templates (e.g., `Test1` in listing 2). Lastly, a concept map provides type and function definitions necessary to satisfy the requirements corresponding to a particular concept id cid. In X++, there can be only one concept map per concept. The second syntax rule for concept maps, at the bottom of figure 2, is used in the metatheory. It states that given an *implicit* concept map icm, which, as explained in section 4, is a tuple of a concept map and a concept identifier, the concept map part of the tuple can be extracted using syntax akin to field access.

4 SEMANTICS OF SEPARATE TYPE CHECKING

In section 2 we have informally outlined how separate type-checking safety is achieved. Here, we define separate type checking safety formally, in the form of a theorem:

Theorem 1 (Separate type-checking safety)**.**
Let C, M, and T represent an environment of the defined concepts, the defined concept maps, and the defined templates, respectively. Let P_{user} be a program, tl be a template, req be a requirements clause of a template. Assume

1. *$P_{user} \Downarrow C\,;M\,;T$, i.e., the program P_{user} is correct and produces the environment $C\,;M\,;T$ when processed,*
2. *$tl = \texttt{template}\, req\, tn\,\{\,e\,\} \in T$, i.e., tl is a specific template in the environment,*
3. *$req = \texttt{requires}\, \overline{cid_i}^{\,i}$, i.e., the requirements of the template tl are given by concept identifiers $\overline{cid_i}^{\,i}$, and*
4. *$\overline{cm\,(M, cid_i) = cm_i}^{\,i}$, i.e., for every concept identifier cid_i a concept map cm_i can be found in the list of defined concept maps.*

Then, instantiating the template tl is guaranteed to succeed. Template instantiation can be reduced to binding symbols from each requirement cid_i to definitions in the corresponding cm_i, and no additional checks are necessary.

The theorem states that given a correct program, i.e., one that instantiates ($\Downarrow$), a concept-constrained template can be *safely* instantiated without any additional type checking, which otherwise is necessary for unconstrained templates. The particular implementations of the types and functions used in the template are taken from the concept maps corresponding to the requirements rather than from the surrounding or argument-dependent namespaces as in the instantiation of unconstrained templates. The first premise states that the program P_{user} instantiates to an environment $C\,;M\,;T$. The premises 2–4 simply introduce names for a template and the concept maps that are referred to in the conclusion of the theorem.

In the remainder of this section we present in detail the crucial *program instantiation* relation that is the centerpiece of our semantic definition. The two major tasks of the program instantiation relation are processing of constrained templates and implementation binding in concept maps—the discussion is organized accordingly. In this section, we concentrate on the discussion of our semantics and in section 5 we discuss its correspondence to the wording.

Program Correctness

The program instantiation relation takes a context $C\,;M\,;T$ and a program to a new context $C'\,;M'\,;T'$. The relation consists of 5 rules, one for the empty program and one per d-production in figure 1. The rules for templates and implicit concept map definitions are shown in figures 5 and 7 and discussed in detail in the following subsections. The rules for processing concepts, concept maps, and empty programs are omitted here but can be found in the companion technical report [31]. The rule for processing concepts contains only a basic check that a concept is not already defined in the context. The rule for processing concept maps simply forwards all the work to the rule for processing implicit concept maps and the rule for processing empty programs is trivial.

It is important to note that while our semantic definition of program instantiation defines which programs are correct, there is no effort made to establish in what way an incorrect program is wrong. If a program is incorrect, it is simply impossible to show that it instantiates but no "error diagnostic" can be obtained as to why. This is a usual property of formal semantic definitions—error handling has to be added by compiler writers.

Both type checking of constrained templates and checking of concept map definitions depend on the formalization of name lookup in concepts. Accordingly, we preceed the discussion of the program instantiation rules with a presentation of the rules for name lookup.

Name Lookup

Name lookup for concepts, described in clause 14.9.3.1 (of [7]), is quite different from other name lookups in C++; we have given some examples in listings 2, 3, and 4, in section 2. The main difference is that functions and types may be declared multiple times throughout a refinement hierarchy but, in difference to other name lookups in C++, without causing ambiguity—if there are duplicates, a single representative is chosen. Clause 14.9.3.1 designates the unique representative to be the type or function found first in the depth-first traversal of the refinement structure, where refinement lists preserve the textual order in which refinements were given in a program.

$$\frac{\begin{array}{c} 1.\, C \vdash cid \prec \overline{cid_i}^{\,i} \qquad 2.\, \mathit{find\text{-}scope}\,(C, cid, f) = \overline{sig} \\ 3.\, \overline{\mathit{find\text{-}scope}\,(C, cid_i, f) = \overline{sig_i}}^{\,i} \qquad 4.\, \overline{sig}' = \overline{sig}\,\overline{\overline{sig_i}}^{\,i} \qquad 5.\, \mathit{rdup}\,(\overline{sig}') = \overline{sig}'' \end{array}}{\mathit{find\text{-}rec}\,(C, cid, f) = \overline{sig}''}$$

$$\frac{\begin{array}{c} 1.\, \mathit{con}\,(cid) = con \qquad 2.\, \mathit{find\text{-}concept}\,(C, con) = cp \qquad 3.\, \overline{f^a}\,(cp) = \overline{f^a} \\ 4.\, \mathit{overload\text{-}set}\,(f, \overline{f^a}) = \overline{sig} \qquad 5.\, \mathit{normalize}\,(C, cid, \overline{sig}) = \overline{sig}' \end{array}}{\mathit{find\text{-}scope}\,(C, cid, f) = \overline{sig}'}$$

Figure 3: Function name lookup in constrained context

Figure 3 lists the rules for function name lookup. The first rule defines recursive lookup, *find-rec*, and the second one defines local lookup in a concept, *find-scope*. The recursive rule first finds the sequence of refined concept identifiers that, as discussed in section 3, preserves the depth-first traversal order of the refinement hierarchy. Then, in the second premise, associated function signatures, $\overline{sig}$, are extracted from the concept *cid*, the root of the current lookup. In the third premise, a sequence of associated function signatures, $\overline{sig_i}$, is extracted from each of the concepts, cid_i, found in the first step. The sequences of function signatures are concatenated in the same order as their enclosing concepts in the sequence $\overline{cid_i}^{\,i}$, preserving the depth-first traversal order of discovery. In premise 4, the signatures $\overline{sig_i}$ from the refined concepts are appended to the sequence of signatures for the root of the lookup, $\overline{sig}$. Finally, in premise 5, the function *rdup* (defined elsewhere) removes duplicate signatures from the sequence $\overline{sig}'$, keeping the signatures encountered first and removing the subsequent duplicates, keeping only a single representative for each function. The second rule describes how an overload set is extracted from each concept locally. The first two premises extract the name *con* of a concept from a concept identifier and check that such concept indeed exists in the current environment. The third premise extracts a list of associated functions, $\overline{f^a}$, from the concept and in premise 4 an overload set is constructed, extracting signatures for functions named f from $\overline{f^a}$. Premise 5 is the most involved. For every signature in the overload set, the types in the signature are looked up in the current concept. This is done so that signatures can be distinguished later from other signatures with the same unqualified parameter types; for example, `void f(t)` may be `void f(X<T>::t)` in one concept and `void f(Y<T>::t)` in another. To distinguish these functions later, fully qualified type names must be used for parameter types. The definitions of the functions used in the premises, e.g., *normalize*, are listed in the report [31].

$$\frac{1.\,\mathit{find\text{-}scope}\,(C, \mathit{cid}, \mathit{id}_\tau) = \tau}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{id}_\tau) = \tau} \qquad \frac{\begin{array}{ll} 1.\,\mathit{find\text{-}scope}\,(C, \mathit{cid}, \mathit{id}_\tau) = \mathit{None} & 2.\,C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}_i}^{\,i} \\ 3.\,\overline{\mathit{find\text{-}rec}\,(C, \mathit{cid}_i, \mathit{id}_\tau) = \tau?_i}^{\,i} & 4.\,\mathit{rm}?\,(\overline{\tau?_i}^{\,i}) = \end{array}}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{id}_\tau) = \mathit{None}}$$

Figure 4: Selected rules for recursive type lookup

Type lookup is similar to function lookup except that only one type is returned rather than a set, as in the case of functions. On the other hand, there are more cases to handle since types can be hidden. All together, the recursive part of type lookup comprises 4 rules, of which 2 are shown in figure 4. The first rule is the base case of the recursive search: if the scope identified by *cid* contains an associated type τ named id_τ, the search is terminated and τ is the result of this part of the lookup. There may be other concepts that *cid* refines but hiding rules (clause 3.3.10 para. 1) require that they not be searched. The first premise of the second rule ensures that no type was found in the current concept. Then, the second premise extracts the directly refined concepts ($\prec_1$). If the transitive refinement ($\prec$) was applied instead, the base rule could not stop the search from progressing into scopes in which potentially matching types should be hidden by an earlier result. In the third premise, the refined scopes are searched recursively, until the first rule is matched or the list of refinements extracted in the second premise is empty. For each of the searched scopes, cid_i, a type option, $\tau?_i$, is returned. Finally, the fourth premise states that after removing all *None* results, which signify unsuccessful lookups in refined scopes, the sequence of results is empty. In such case, the result of the whole lookup is *None* since no type named id_τ was found.

Given the definition of name lookup, we next discuss the two parts of separate type checking: checking of constrained templates and of implementation binding.

Type Checking of Constrained Templates

Processing of constrained templates is the first step of separate type checking. The rule in figure 5 shows how a template definition *tl*, in a program *tlP*, is checked in the context $C; M; T$, producing a new context $C'; M'; T'$. Checking of constrained templates boils down to type checking of the expression e contained in the template *tl* in the context given

$$\frac{\begin{array}{c} 1.\, tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\ tn\,\{\,e\,\} \qquad 2.\, \overline{cid_i \text{ defined in } C}^{\,i} \\ 3.\, tl \text{ not defined in } T \qquad 4.\, C\,;\emptyset\,;\overline{cid_i}^{\,i} \vdash e :\ \tau \qquad 5.\, C\,;M\,;tl\,T \vdash P \Downarrow C'\,;M'\,;T' \end{array}}{C\,;M\,;T \vdash tl\,P \Downarrow C'\,;M'\,;T'}$$

Figure 5: Type checking of constrained templates

by the requirement clause of the template; premise 4 states that the expression e is of the type τ ($e :\ \tau$) in the context of all concepts C (necessary to traverse the refinement hierarchy), an empty variable typing environment $\emptyset$, and the requirements $\overline{cid_i}^{\,i}$ of the template tl. The other premises, in order, introduce the full definition of tl, ensure that every concept referred to in the requirements is defined, assert that a template with the same name is not already defined, and that processing the rest of the program P with the currently processed template inserted into context ($tl\,T$) produces the new context $C'\,;M'\,;T'$.

$$\frac{\begin{array}{c} 1.\, \overline{\mathit{find\text{-}rec}\,(C, cid_i, f) = \overline{sig}_i}^{\,i} \qquad 2.\, \mathit{distinct}\,(\overline{\overline{sig}_i}^{\,i}\,) \\ 3.\, \overline{C\,;\Gamma;\overline{cid_i}^{\,i} \vdash e_k :\ \tau_k}^{\,k} \qquad 4.\, \mathit{overload\text{-}res}\,(f\,\overline{\tau_k}^{\,k}, \overline{\overline{sig}_i}^{\,i}\,) = sig \qquad 5.\, \mathit{returns}\,(sig) = \tau \end{array}}{C\,;\Gamma;\overline{cid_i}^{\,i} \vdash f\,(\overline{e_k}^{\,k}) :\ \tau}$$

$$\frac{1.\, \overline{\mathit{find\text{-}rec}\,(C, cid_i, \tau) = \tau?_i}^{\,i} \qquad 2.\, \mathit{rm?}\,(\overline{\tau?_i}^{\,i}) = \overline{\tau} \qquad 3.\, \overline{\tau} \approx \tau}{C\,;\Gamma;\overline{cid_i}^{\,i} \vdash \texttt{obj}\,\tau' :\ \tau}$$

Figure 6: Selected rules for typing expressions in constrained contexts

An expression is typed in the context consisting of available concepts C, a variable typing environment Γ, and a list of concept identifiers $\overline{cid}$ that represents the requirements of a constrained template. Type checking of expressions includes some expected rules, for example, the type of a variable is looked up in a local environment. The difference to the usual type checking is in name lookup, which is performed in the concepts that the constrained template requires. Figure 6 shows typing of function application and of object creation, which require name lookup.

Type checking a function application $f\,(\overline{e_k}^{\,k})$ consists of 5 steps. First, *find-rec* looks up f in each of the requirements and the lookup gives a sequence of function signatures representing an overload set from each of the requirements. The second step ensures that function signatures found through name lookup are distinct and that there are no ambiguities in the overload set. In the third premise, the arguments with which the function is called are typed themselves. Then, in premise 4, overload resolution is invoked to choose one signature from the overload set given the function name and the types of the arguments with which the function is called. Finally, in the fifth premise, the return type of the function is extracted. Given all these premises, function application is typed with the return type of the function selected by overload resolution.

Type checking of object creation starts with name lookup, in each of the requirements, for the type with which the object is created. Each lookup returns a type option, which is *None* if name lookup failed. In premise 2, the *None* results are removed, yielding a list of types that were successfully looked up. Premise 3 states that only one distinct type was found by requiring set equality ($\approx$) between the sequence of results $\overline{\tau}$ and a sequence of a single type τ. In such case, the created object can be typed with type τ.

In the following subsection, we discuss the next step of separate type checking as outlined in section 2, namely implementation binding.

Implementation Binding, Compatibility, and Propagation

$$\frac{\begin{array}{l} 1.\, icm = (cm, cid?) \qquad 2.\, cm = \texttt{concept\_map}\, cid\, \{\, \overline{tydef}\, \overline{fdef}\, \} \\ 3.\, cm \text{ not defined in } M \qquad 4.\, tydefs\text{-}check\, (C, cid, \overline{tydef}) \qquad 5.\, tydefs_{\eqcirc}\, (C, M, cid, \overline{tydef}) \\ 6.\, fdefs\text{-}check\, (C, cid, \overline{fdef}) \qquad 7.\, fdefs_{\eqcirc}\, (C, M, cid, cid?, \overline{fdef}) \\ 8.\, icms\, (C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm_i}^{\,i} \qquad 9.\, C\,;\, icm\, M\,;\, T \vdash \overline{icm_i}^{\,i}\, P \Downarrow C'\,;\, M'\,;\, T' \end{array}}{C\,;\, M\,;\, T \vdash icm\, P \Downarrow C'\,;\, M'\,;\, T'}$$

Figure 7: Checking and generation of implicit concept maps

The two main tasks when processing a concept map are to check whether the definitions that the map provides are compatible with the definitions in concept maps defined earlier and to implicitly define concept maps for refined concepts as necessary (clause 14.9.3.2). The program instantiation rule for processing concept maps is listed in figure 7. We first give an overview of the rule and then discuss it in detail in the remainder of the section.

The first premise simply creates an alias for the implicit concept map *icm* that is being processed, which is a tuple of a concept map *cm* and an optional concept identifier *cid*; the second premise creates an alias for the *cm* component of *icm*. The third premise establishes that a concept map for the same concept instance has not yet been defined. Premises 4 through 7 state that type and function definitions are well-formed, well-typed, and compatible ($\eqcirc$) with definitions provided in existing concept maps. Premise 8 gives the set of implicit concept maps that are generated by the concepts directly refining the currently processed concept map *cm*. And finally, premise 9 finds the new environment $C'\,;\, M'\,;\, T'$ resulting from processing the newly generated implicit concept maps and the rest of the program in the old environment extended by the current concept map. Each of the generated implicit concept maps is then processed in the same manner as the map that initiated their generation. This recursive processing ensures that implicit concept maps are propagated throughout the refinement hierarchy, traversed depth first.

$$\frac{1.\, \forall\, \tau f\, (\overline{var_i \colon \tau_i}^{\,i})\, \{\, e\, \} \in \overline{fdef} \bullet C\,;\, [\overline{var_i \colon \tau_i}^{\,i}]\,;\, cid \vdash e \colon \tau \qquad 2.\, distinct\, (\overline{sig}\, (\overline{fdef}))}{fdefs\text{-}check\, (C, cid, \overline{fdef})}$$

$$\frac{\begin{array}{l} 1.\, \overline{ty^a}\, (C, cid) = \overline{ty^{a}}^{\prec} \qquad 2.\, \overline{id_\tau}\, (\overline{tydef}) \approx \overline{id_\tau}\, (\overline{ty^{a}}^{\prec}) \\ 3.\, \forall\, tydef \in \overline{tydef} \bullet tydef \surd \qquad 4.\, distinct\, (\overline{id_\tau}\, (\overline{tydef})) \end{array}}{tydefs\text{-}check\, (C, cid, \overline{tydef})}$$

Figure 8: Function and type definitions check in concept maps

Figure 8 shows how function and type definitions are checked in the premises 4 and 6 in figure 7. For functions, the first condition is that the body of every function definition (• separates the quantifier from the condition) must type-check in the context of the concepts C in the environment, the function parameters $[\overline{var_i\colon \tau_i}^{i}]$, and the concept identifier *cid*, designating the concept map to which the function definition belongs. Type checking proceeds as described previously (figure 6).

Checking of type definitions differs from checking function definitions. The first premise extracts associated types, $\overline{ty^a}^{\prec}$, from all refined concepts (indicated by $\prec$ in $\overline{ty^a}^{\prec}$) and the second premise requires that type definitions in $\overline{tydef}$ provide a definition for every type in $\overline{ty^a}$. The equality in the second premise is a set equality because different concepts refined by *cid* may require an associated type with the same name. Premise 3 requires that every type definition is well-formed (see [31] for details) and premise 4 requires that there is only one definition per associated type name.

$$\frac{\begin{array}{ll} 1.\ \overline{f^a}(C, cid) = \overline{f^a}^{\prec} & 2.\ \overline{sig}(C, M, cid?, cid) = \overline{sig}^{\prec} \\ 3.\ distinct(\overline{sig}^{\prec}) & 4.\ \overline{sig}(\overline{fdef}) \approx \overline{f^a}^{\prec} \setminus \overline{sig}^{\prec} \end{array}}{fdefs_{\equiv}(C, M, cid, cid?, \overline{fdef})}$$

$$\frac{\begin{array}{ll} 1.\ \overline{ty^a}(C, cid) = \overline{ty^a} & 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\ \overline{tydef}(M, \overline{cid}^{\prec_1}) = \overline{tydef}^{\prec_1} & 4.\ \overline{ty^a} \vdash \overline{tydef} \equiv \overline{tydef}^{\prec_1} \end{array}}{tydefs_{\equiv}(C, M, cid, \overline{tydef})}$$

Figure 9: Function and type definitions compatibility in concept maps

The next step is to check that the newly introduced implementations are *compatible* with the existing ones (premises 5 and 7 in figure 7). The rules for compatibility checking are listed in figure 9. For functions, first all associated function requirements are extracted from refinements, including the associated functions from the currently considered concept. The second premise finds existing implementations, omitting concept maps that are implicitly defined by the same parent (based on the *cid*? argument to $\overline{sig}$). The omission is necessary to properly handle diamonds in the refinement hierarchy; an example of such hierarchy is shown in the following code snippet:

```
concept A<typename T>                { void f(); }
concept B1<typename T> : A<T>        { }
concept B2<typename T> : A<T>        { }
concept C<typename T>  : B1<T>, B2<T> { }

concept_map C<int> { void f() { return; } }
```

The definition of `void f()` is propagated from `C` to its refined concepts, and maps for `A`, `B1`, and `B2` are automatically created. The propagation proceeds from left to right in the refinement clauses, generating concept maps for `B1`, `A`, and `B2` in that order. When the map for `B2` is checked, the map for `A` is already in the environment and contains the definition for `void f()` propagated from `C`. The concept map for `B2` contains the same definition but since the two maps originate from `C` it is clear that they must be the same. The third premise checks for duplicate definitions. Finally, the fourth premise requires that the set of function signatures in the current concept map, $\overline{sig}(\overline{fdef})$, is equal to the difference between the set of required signatures $\overline{f^a}^{\prec}$ and the set of signatures of required functions $\overline{sig}^{\prec}$.

$$\frac{\begin{array}{l}1.\,\overline{tydef}' = \{\, tydef \in \overline{tydef} \mid \neg\, id_\tau(tydef) \in \overline{id_\tau}(\overline{ty^a})\,\}\\ 2.\,\forall\, tydef \in \overline{tydef}' \bullet \textit{find-defs}(id_\tau(tydef), \overline{tydef}^{\prec 1}) \approx tydef\end{array}}{\overline{ty^a} \vdash \overline{tydef} \doteq \overline{tydef}^{\prec 1}}$$

Figure 10: Type definitions one-level compatibility

Compatibility checking of associated types, given by the second rule in figure 9, is different because associated types can be defined many times (clause 14.9.3.2 para. 4), as long as each definition names the same type, and they can be hidden by other associated types while functions cannot. First, a sequence of associated type requirements is extracted from the current concept. Next, in premise 2, the sequence of directly refined concepts is obtained and then, in premise 3, all the type definitions contained in the corresponding concept maps are extracted. Finally, in the fourth premise, the definitions are checked for compatibility, according to the rule in figure 10. Compatibility between type definitions $\overline{tydef}$, from the concept map designated by cid, and type definitions $\overline{tydef}^{\prec 1}$, from the concept maps for concepts directly refining cid, are compared in the context of associated types $\overline{ty^a}$ that are explicitly required in the concept cid. These associated types hide others with the same names in the refined concepts and, consequently, their definitions do not need to be compared with the ones in the refined concept maps; premise 1 in figure 10 accordingly eliminates hidden definitions from the compatibility check. All remaining definitions, $\overline{tydef}'$, must be identical to those in the directly refined concept maps. Accordingly (premise 2), the set of definitions in $\overline{tydef}^{\prec 1}$, that define a type with the same name, $id_\tau(tydef)$, as that defined by a $tydef$ in $\overline{tydef}'$, must consist of only one element, $tydef$ itself. Because this compatibility check is performed at every step of refinement and because every concept map must contain definitions for associated types from its own and from all concepts it refines, type definitions for associated types with the same name are guaranteed to be the same unless one definition hides another.

$$\frac{\begin{array}{l}1.\, cid? = cid_{impl} \qquad 2.\, C \vdash cid \prec_1 \overline{cid}^{\prec 1} \qquad 3.\, \overline{cid_i}^{\,i} = \{\, cid \in \overline{cid}^{\prec 1} \mid cm(M, cid) = None \,\}\\ 4.\, \overline{fdef}(C, M, cid_{impl}, cid) = \overline{fdef}^{\prec} \qquad 5.\, \overline{\overline{ty^a}(C, cid_i) = \overline{ty^a}^{\prec}_{i}}^{\,i}\\ 6.\, \overline{\overline{f^a}(C, cid_i) = \overline{f^a}^{\prec}_{i}}^{\,i} \qquad 7.\, \overline{\overline{tydef}_i = \{\, tydef \in \overline{tydef} \mid id_\tau(tydef) \in \overline{id_\tau}(\overline{ty^a}^{\prec}{}_i)\,\}}^{\,i}\\ 8.\, \overline{\overline{fdef}_i = \{ fdef \in \overline{fdef}\,\overline{fdef}^{\prec} \mid sig(fdef) \in \overline{f^a}^{\prec}_{i} \}}^{\,i}\\ 9.\, \overline{cm_i = \texttt{concept\_map}\, cid_i \{\overline{tydef}_i \overline{fdef}_i\}}^{\,i}\end{array}}{icms(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{(cm_i, cid_{impl})}^{\,i}}$$

Figure 11: Generation of implicit concept maps

After the current concept map is checked for compatibility, concept maps for refined concepts are generated if they do not already exist (premise 8 in figure 7). Figure 11 shows how implicit concept maps are generated from a concept map that itself has been implicitly generated. Another rule, not listed in the figure, defines how concept maps are generated from an explicit concept map; these two rules differ only in the first premise. The rule in the figure handles concept maps that are implicitly generated by requiring the optional concept identifier of the implicit "parent" to be set to cid_{impl}. The identifier of the parent is then passed on to further generated concept maps, marking all generated maps with the explicit map that triggered the generation. For an explicit concept map, the first premise reads $cid? = None$, meaning that the concept map currently considered was not generated from another concept map but that it was explicitly defined. Then, the concept identifier of the current map, *cid*, is passed to the generated concept maps as the "parent." In premise 2, the sequence of directly refined concepts is extracted and then, in premise 3, restricted to only the concepts for which no concept map is found in the environment M. Next, a sequence of all associated function implementations is extracted from all concept maps that are defined for any of the concepts refined by the current concept represented by *cid*. In premises 5 and 6, the sequences of required associated types and functions are extracted for each of the directly refined concepts in $\overline{cid_i}^{\,i}$. In premise 7, a sequence of type definitions $\overline{tydef}_i$ is constructed for each of the implicit concept maps from the definitions $\overline{tydef}$ in the parent concept map. This is possible because, as we described earlier, the parent concept map contains type definitions for all associated types from all concepts it refines. Premise 8 defines how associated function definitions are generated. Premise 9 puts all the parts together in a sequence of implicitly defined concept maps to be returned.

5 INTERPRETATION OF THE INFORMAL DEFINITION

While our semantic definition is as faithful as possible to the concept wording [7], we have encountered informal definitions that were either ambiguous or seemed to be inconsistent with the understood design goal. In this section we outline the important differences between our definition and the wording.

Name Lookup

Clause 3.3.10 para. 1 states that a name can be hidden by a declaration in a refining concept. While it is not stated explicitly which names can be hidden, the previous publications on concepts and the authors' experience suggest that associated type names can be hidden by associated type names in refining concepts. On the other hand, the formulation of name lookup in concepts, specifically clause 14.9.3.1 para. 4, 4th bullet, states that associated type names are collected from the scopes of *all* refining concepts and that for the name lookup to succeed they all must be the same type, as if there was no hiding. Our semantics assumes hiding, as prescribed in clause 3.3.10 para. 1, and terminates the search through the refined scopes once a matching type is found: the first rule in figure 4 does not continue if a type was found while the second rule continues the search since the first premise assures that no matching type was found in the currently searched scope.

Furthermore, clause 14.9.3.1 para. 4 states that if there is more than one associated type with the same name in a refinement hierarchy, these types must be explicitly declared to be the same—through `SameType` constraints (a built-in, compiler-supported concept)—for name lookup to succeed; if these constraints are missing then name lookup fails with ambiguity. Our semantics implicitly assumes these constraints; the motivation for that assumption is given in the next subsection.

Implementation Binding

Compatibility and definition propagation. Clause 14.9.3.2 para. 4 states that a definition in a concept map for a refining concept is compatible with a definition in a refined concept only in three cases:

- the definition in the refining concept map is explicit and the definition in the refined map is implicit,
- the definition in the refined concept map is explicit and the definition in the refining concept map is implicit,
- the definitions satisfy an associated type requirement and both definitions explicitly name the same type.

When formalizing this definition we have discovered that the definition in the wording may allow cases that are not compatible and that it makes defining certain concept maps impossible. Consequently, our semantic rules differ from the wording.

The first compatibility difference is that we restrict associated types definition to the third case given above. That is, the second rule in figure 9 requires, in the third premise, that every concept map provide definitions for *every* associated type requirement in the concept hierarchy. Since every concept map explicitly defines all required associated types, the compatibility check boils down to the third rule, which requires that the definitions for the same associated type requirement name the same type. Note that while the wording does not take hiding into account, our definition in figure 10 eliminates hidden types from compatibility check.

The compatibility rules in the wording, specifically the second rule, allow the situation where concept maps for refined concepts provide differing definitions for associated functions with identical signatures. This compatibility breach, coupled with the name lookup rules given in clause 14.9.3.1 and reflected in our semantic rules listed in figure 3, means that one definition is chosen out of potentially conflicting associated function definitions, according to the depth-first traversal discovery rules discussed in section 4. Our semantic definition allows *only one* explicit definition for an associated function in refined concept maps, as seen in figure 9, first rule, premise 3. Consequently, there cannot be conflicting definitions. In addition, while the wording allows only one level of definition propagation, from an explicit definition to an implicit one, our rules allow definitions to be propagated further. In figure 11, associated function definitions are taken from the sequence $\overline{fdef}\,\overline{fdef}^{\prec}$ meaning that definitions will be propagated downwards but also sideways in the refinement hierarchy. The wording allows propagating downwards, from an explicit definition in a refining concept map to an implicit definition in a refined concept map, while our definition allows taking both implicit and explicit definitions in refined concept maps and propagating them to other refined concept maps. If, in figure 11, we changed the sequence of available definitions to $\overline{fdef}$, our semantics would match the semantics in the wording. However, the semantics in the wording may be too restrictive; in the following code listing we give an example of a situation where concept maps cannot be defined in the semantics given by the wording.

```
concept A<typename T>              { void f(); }
concept B<typename T>              { void f(); }
concept C<typename T>              { void f(); }
concept E<typename T> : A<T>, B<T> { }
concept F<typename T> : B<T>, C<T> { }

concept_map A<int> { void f() {} }
concept_map B<int> { void f() {} }
concept_map C<int> { void f() {} }
concept_map E<int> { }                // error
concept_map F<int> { }                // error

concept_map E<float> { void f() {} }
concept_map F<float> { void f() {} }  // error
```

Given the concept hierarchy in the example there is no way to define a concept map for all of the concepts. The concept maps for `int` are defined starting with the refined concepts `A`, `B`, and `C`. The concept maps for the refining concepts have no explicit definitions of `void f()` and must "pull" the implementations from the refined concept maps. Yet, for both concept maps `E<int>` and `F<int>` there are conflicting definitions in the refined maps. In the maps for `float`, the first concept map is defined and it implicitly generates maps for `A<float>` and `B<float>`. When the map for `F<float>` is defined, an explicit definition of `void f()` must be given so that the missing map for `C<float>` can be generated but that definition conflicts with the one in `B<float>` generated previously. If our "sibling-to-sibling" semantics was allowed, the map for `F<float>` could be defined without an explicit definition of `void f()`, first pulling the definition from `B<float>` and then pushing it to `C<float>`.

Our choice of compatibility semantics is not meant as a final one. Instead, we indicate that there may be a problem and we show one example solution.

Substitution of associated type implementations. According to the wording in clause 14.9.2.2 para. 2, associated types are substituted with their assigned implementations in the signatures of associated functions within concept maps; for example, `void f(A<T>::t)` becomes `void f(int)` in contexts where `A<T>::t` has been implemented as `int`. In our semantics, this substitution does not take place; this can be seen in premise 4 of the first rule in figure 9 where signatures of function definitions must be exactly the same as the signatures of associated functions, without substituting the actual type implementations, and in premise 1, where signatures of the associated functions are normalized, i.e., parameter types are fully qualified (the extraction of associated function signatures is not shown in any of the figures but is included in the companion technical report [31]). These function definitions are directly stored in concept maps, which can be seen in premise 2 of the rule in figure 7 for explicit concept maps, and premise 9 of the rule in figure 11 for implicit concept maps. The following code listing shows an example of a situation where our semantics allows code that is incorrect according to the wording and rejects code that the wording deems correct.

```
 1 concept A<typename T>        { typename t;  void f(t); }
 2 concept B<typename T> : A<T> { typename t;  void f(t); }
 3
 4 concept_map B<int> {
 5   typedef int t;
 6   void f(int) { return; }
 7 }
 8 concept_map B<float> {
 9   typedef int t;
10   void f(A<int>::t) { return; }
11   void f(B<int>::t) { return; }
12 }
```

The concept map for `B<int>` is allowed by the wording but not by our semantics. The difference between two associated functions is lost and they are "munged" into a single `void f(int)` function. The second map for `B<float>` is allowed by our semantics but not by the wording. According to the wording, both `A<int>::t` and `B<int>::t` become aliases to `int` due to the definition on line 9 and the two function signatures on lines 10–11 are identical, causing an error. In our semantics, the substitution of associated type implementations does not take place and, although both associated types have been assigned the same implementation, the functions remain different, insulating templates constrained with the concept `B` from the fact that both associated types are assigned the same implementation.

Type Checking of Concept-Constrained Templates

According to the wording, clause 14.10.2 para. 18, in a situation where different requirements introduce conflicting associated functions, no effort should be made to check whether the conflicting functions are actually the same. The following listing gives an example of a situation in which name lookup finds the same function through each of the requirements but the call to the function results in an error:

```
concept Child<typename T> { void f(); }
concept Parent1<typename T> : L<T> { }
concept Parent2<typename T> : L<T> { }

template<typename T> requires Parent1<T>, Parent2<T>
void test1() { f(); }  // error
```

As a consequence of performing type lookup in each of the requirements separately and choosing the type found first in depth-first search, sometimes intuitively correct code may cause errors; a small example is shown in the following code listing[1]:

```
concept A1<typename T>                  { typename t; }
```

[1]The example would require same-type constraints in the concepts P1, P2, and P3 but in our semantics the constraints are implied.

```
concept A2<typename T>                { typename t; }
concept P1<typename T> : A1<T>, A2<T> { }
concept P2<typename T> : A2<T>, A1<T> { }
concept P3<typename T> : A2<T>, A1<T> { }

template<typename T>  requires P1<T>, P2<T>  struct Test1 {
  typedef t test;  // error: A1<T>::t or A2<T>::t?
};
template<typename T>  requires P2<T>, P3<T>  struct Test2 {
  typedef t test;  // ok: A2<T>::t
};
```

In `Test1`, name lookup for `t` finds `A1<T>::t` in `P1<T>::t` and `A2<T>::t` in `P2<T>::t`. Since the two types are different, there is ambiguity. In `Test2`, lookup of `t` in both of the requirements results in `A1<T>::t`, and the sequence of final results contains two types that are the same. The first rule in figure 6 makes it clear that lookup is performed in each requirement separately and that the knowledge about identity of associated types is lost between requirements. In this case our semantics agrees with the wording but a potential problem is clearly exposed.

6 RELATED WORK

Linguistic support for concepts in C++ has a long history in the C++ standardization process, dating back to Stroustrup [26] and Siek et al. [23]. More recently, concepts have been presented to a broader forum by Dos Reis and Stroustrup [18] and soon after by Gregor et al. [6]. All concept proposals share the goal of separate type checking safety and Dos Reis and Stroustrup indeed provide a sketch of a type checking safety theorem as a future work item. Our theorem 1 is a development of this idea, supported by our semantic definition—in that sense, our contributions are a direct continuation of the previous work.

The tradition of C++ concepts is rooted in the formal setting of algebraic specification [11] but, nevertheless, the true utility of concepts was first manifested in the design documentation of the Standard Template Library [1, 24]. As prominence of concepts rose in C++ generic libraries, there was renewed interest in formalizing concepts. Willcock et al. [29] proposed a formal definition of concepts based on their algebraic specification roots, on various counterparts in other languages, mostly in ML [14], and on the ongoing experience of concept use in the C++ world. Their formalization had the ambitious goal of capturing the very notion of concepts in a programming language. Siek and Lumsdaine [21], on the other hand, incorporated concepts into System F [3], providing strong formal foundation to separate type checking with concepts and outlining the important features that a concept system should include. Finally, in our own work [34], we bridged the gap between concepts as proposed for C++ and their roots in algebraic specification—we have described how concepts may be interpreted as specifications in the context of institutions [4] (Haveraaen [8] also considers concepts as institutions, specifically in the context of testing). In the current paper, we complement the existing body of work with a C++ specific formalization that formalizes the design of concepts as proposed for the next version of C++. Our formalization has the practical goals of enabling precise analysis of the current design and providing a foundation for the future compiler implementations, but also the more theoretically inclined goal of initiating the work on formal verification of crucial parts of C++ concepts design.

Concepts are the basis of practical analysis and programming techniques. Currently, these applications of concepts are developed in terms of concrete C++ syntax and tools—our analysis provides a foundation to describe and develop such applications. In our previous work [33], we have demonstrated a change impact analysis for conceptual specification of generic libraries; Tang and Järvi [27] propose concept-based optimizations. Concepts have also been used to compose generic libraries. Järvi et al. [10] use concepts to compose GUI controls from different libraries and to compose graph and imaging libraries; Breuer et al. [2] compose a parallel graph and eigensolver libraries to obtain an efficient parallel graph eigensolver. Furthermore, Gregor and Lumsdaine [5] develop principles and patterns for using generic libraries—their development is based on concepts.

Our formalization shares the motivation and some techniques with formalizations of other parts of C++. Wasserrab et al. [28] have formalized type safety of multiple inheritance and provided a machine-checked proof in support of their hypothesis. Many issues covered by their formalization are related to lookup of names; in particular, they rely on a slightly modified sub-object model introduced by Rossie and Friedman [19]. Siek and Taha [22] have formalized the template instantiation process and operational semantics for a subset of the C++ language, providing a proof of type safety. Their formulation of the template instantiation process served as the basis for our program instantiation relation. Dos Reis and Stroustrup [17] have provided a formal framework for a large part of the C++ language, concentrating on the features necessary to concept checking. Their formalization is broad, covering a large part of the C++ language, but is not as well developed as others and, to our knowledge, has not been used in the current process of specifying concepts. A final note of caution: Järvi et al. [9] have identified cases where separate type-checking safety cannot hold, yet those violations stem from the interaction of the (safe) concept language with unsafe features *outside* (e.g., partial ordering of function templates) and cannot be prevented without changing, and breaking, legacy C++.

Finally, our formalization has been strongly inspired by the Ott tool [20]. Ott lowers the cost of formally defining a language and enables a smooth progression in formality, from LaTeX typeset mathematics to a theorem prover definition. Our semantic definition borrows many techniques from the formalizations of a fragment of OCaml by Owens [16] and of Java modules semantics by Strniša et al. [25], which are defined using Ott.

7 CONCLUSIONS AND FUTURE WORK

While concepts are considered one of the most important features to be introduced in the next version of C++, the current design process relies on informal design documentation supported by only a partial implementation and little experience. In this paper we have provided a formal semantic definition of the crucial parts of concept design, covering name lookup in concepts, implementation binding in concept maps, and type checking in concept-constrained templates.

We first give, in section 2, an informal introduction to concepts. We illustrate separate type checking with several examples, such as listing 1 that shows the difference between concept-constrained and unconstrained templates, and with more specific examples, such as listing 5 that shows implementation binding for associated functions.

Next, in section 3, we introduce the abstract syntax of a small language, X++, that contains the crucial features of the C++ concept system: concepts, concept maps, and concept-constrained templates. X++ includes language features necessary to discuss concepts, such as overload resolution, but excludes all other features that are orthogonal to concepts.

The semantic definition of concepts, which is the crux of our contribution, is presented in section 4. There, we formulate the separate type-checking safety theorem (theorem 1). In the rest of the section we present the semantic judgments that form the basis of the theorem. The presentation of the semantics is divided into three parts: name lookup, type checking of constrained templates, and implementation binding.

Finally, in section 5, we discuss how our formal definition corresponds to the informal definition in the wording [7]. Two differences stem from our attempt to fix apparent errors in the wording. First, type lookup rules in the wording do not respect hiding rules established elsewhere in the wording; in our semantic definition the hiding rules are honored. Second, the wording admits a situation where concept maps for refined concepts can provide conflicting implementations for associated entities; our semantics does not allow such situations. Furthermore, our semantics differs from the wording where intuitively valid code is not admitted by the wording. In summary, the differences are that our semantics allows "sibling-to-sibling" propagation of associated function implementations and it does not perform substitution of associated types with their implementations in concept maps. Furthermore, our semantics exposes a possible problem with type checking of concept-constrained templates where types and functions that are intuitively the same are considered different.

Our formalization is useful in the analysis of the current concept design put in front of the C++ standardization committee. A formal definition of concept semantics, such as ours, is a great aid in understanding and discussing the design, and can serve as the specification for implementations. Also, a formal semantics makes it easier to design program analyses involving concepts; instead of having to understand hundreds of pages of C++ standard and complex compiler implementations, an analysis can be prototyped at the abstract level of semantic entities and relations.

To facilitate such design and prototyping, we plan to make our semantics "executable" through a context-sensitive term-rewriting system. In particular we plan to investigate the PLT Redex rewriting system [12]. Using a context-sensitive rewriting system may require restatement of our semantic definition with evaluation contexts, in the style of Wright and Felleisen [30]. Ott currently supports evaluation context semantics but it does not generate output for a rewriting system.

To further enhance usefulness of our semantics for the development of tools we plan to extend the coverage of C++ with concepts. First, we plan to extend our semantics with concept parameters. Directly related features that we plan to include in the next version of semantics are template concept maps and same type constraints on template parameters. We anticipate that adding parameters and the related features may as much as double the size of the definition.

References

[1] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[2] A. Breuer, P. Gottschling, D. Gregor, and A. Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Proc. 20th Internat. Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2006.

[3] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, France, 1972.

[4] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[5] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proc. 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 423–437. ACM, 2005.

[6] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, 2006.

[7] D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

[8] M. Haveraaen. Institutions, property-aware programming and testing. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.

[9] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 272–282. ACM, 2006.

[10] J. Järvi, M. A. Marcus, and J. N. Smith. Library composition and adaptation using C++ concepts. In *Proc. 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 73–82. ACM, Oct. 2007.

[11] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI-92-20, Department of Computer Science, Rensselaer Polytechnic Institute, NY, USA, 1992.

[12] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. 15th Internat. Conf. on Rewriting Techniques and Applications*, volume 3091 of *LCNS*, pages 310–311. Springer, June 2004.

[13] A. Meredith. State of C++ evolution (pre-antipolis 2008 mailing). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

[14] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[15] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 2nd edition, 2001.

[16] S. Owens. A Sound Semantics for OCaml$_{light}$. In *Proc. 17th European Symp. on Programming (ESOP)*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008.

[17] G. Dos Reis and B. Stroustrup. A formalism for C++. Technical Report N1885=05-0145, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2005.

[18] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 295–308. ACM, 2006.

[19] J. G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. *SIGPLAN Not.*, 30 (10):187–199, Oct. 1995.

[20] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *Proc. ACM SIGPLAN Internat. Conf. on Functional Programming (ICFP)*, pages 1–12. ACM, 2007.

[21] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 73–84. ACM, 2005.

[22] J. G. Siek and W. Taha. A semantic analysis of C++ templates. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, LNCS, pages 304–327. Springer, 2006.

[23] J. G. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.

[24] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-95-11(R.1), Hewlett Packard Laboratories, Palo Alto, CA, USA, Nov. 1995.

[25] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proc. 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, and Applications (OOPSLA)*, pages 499–514. ACM, 2007.

[26] B. Stroustrup. Concepts—a more abstract complement to type checking. Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2003.

[27] X. Tang and J. Järvi. Concept-based optimization. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.

[28] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 345–362. ACM, 2006.

[29] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, Apr. 2004.

[30] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115 (1):38–94, 1994.

[31] M. Zalewski. A semantic definition of separate type checking in C++ with concepts—abstract syntax and complete semantic definition. Technical Report 2008:12, Department of Computer Science and Engineering, Chalmers University, 2008. URL http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=72572.

[32] M. Zalewski. *Generic Programming with Concepts*. PhD thesis, Chalmers University, Nov. 2008. URL http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=76351.

[33] M. Zalewski and S. Schupp. Change impact analysis for generic libraries. *Proc. 22nd IEEE Internat. Conf. on Software Maintenance (ICSM)*, pages 35–44, Sept. 2006.

[34] M. Zalewski and S. Schupp. C++ concepts as institutions. a specification view on concepts. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.

ABOUT THE AUTHORS

Marcin Zalewski received his PhD degree in Computer Science from Chalmers University in 2009. Currently, he is a postdoctoral researcher at Open Systems Lab, at Indiana University. His current research interests include generic programming, metaprogramming, and application of algebraic specification methods to software construction. He can be reached at zalewski@osl.iu.edu.

Sibylle Schupp is Professor and Head of the Institute for Software Systems at Hamburg University of Technology. See http://www.sts.tu-harburg.de/~schupp for contact details.