# Automatically Identifying C++0x Concepts in Function Templates

Andrew Sutton, Jonathan I. Maletic
*Department of Computer Science*
*Kent State University*
*Kent Ohio 44242*
*asutton@cs.kent.edu, jmaletic@cs.kent.edu*

## Abstract

*An automated approach to the identification of C++0x concepts in function templates is described. Concepts are part of a new language feature appearing in the next standard for C++ (i.e., C++0x). Concept identification is the enumeration of constraints on the sets of types over which templates can be instantiated. The approach analyzes template source code and computes a set of viable concept instances describing the implied data abstraction of the template parameters. The approach is evaluated on generic algorithms defined in the C++ Standard Template Library (STL). The evaluation demonstrates the effectiveness of the approach. The approach can be used to assist in reengineering existing generic libraries to C++0x. Additionally, it has the potential to assist in the validation of concept hierarchies and interface definition in generic libraries.*

## 1. Introduction

The upcoming version of C++ will introduce a broad array of new features ranging from minor syntactic cleanups to support for additional programming paradigms. One such feature aims to augment and extend current support for generic programming, a programming paradigm well suited to the development of abstract datatypes and generic algorithms. In the current version of C++, generic programming is predicated upon language features such as templates, overloading, argument-dependent lookup, and various programming techniques such as template metaprogramming. The new features for generic programming, in C++0x, are centered on the introduction of the *concept*, a first-class linguistic entity for expressing requirements on sets of types. Concepts can be used to constrain the sets of types over which generic algorithms (i.e., function templates in C++) are instantiated.

With broader support for generic programming in C++, we expect it to become even more widely used to create generic libraries for any number of domains. It will also improve the quality and usability of generic libraries. However, the development and maintenance of generic libraries is not without its difficulties. With respect to generic algorithms, the correct identification of requirements (concepts) on the types that the algorithm operates on is critical. Such requirements dictate what sets of types can or cannot be used with the generic algorithm. For example, one can use pointers in the place of most iterators, but not integers since they cannot be dereferenced. This implies a significant effort is on the horizon to convert existing C++ generic libraries to use concepts.

As such, we feel that the process of migrating and reengineering existing C++ libraries to C++0x will greatly benefit from automated concept identification. Here, we present an approach to automatically identify concepts within function templates. Our approach, analogous to type inference, identifies the possible data abstractions represented by the template parameters of a function template. Analyzing the operators, functions, and types used on or by template parameters yields a set of concepts that describe their fundamental abstractions. Additional analysis is used to further refine this set of concepts and finally determine the set of concept instances that best represent the use of each template parameter within the function.

We prototyped the approach and ran it against a number of STL algorithms (e.g., `find`, `fill`, etc.) and compared the results against the requirements proposed for these algorithms in C++0x's STL description. We also ran the prototype against small variations of the same algorithm to determine the impact on the requirements based on differences in implementation. These evaluations show that the approach is not only a viable tool for assisting in the reengineering of existing generic libraries, but also provides useful insight into the construction and evolution of concept hierarchies and the design of new generic libraries.

This paper is organized as follow. The next section (2) gives a brief introduction to concepts in C++0x for completeness since many readers are not familiar with this new language feature. Section 3 presents our approach to automatically identifying concepts in existing template functions. Section 4 present the evaluation of our approach. This is followed by related work in Section 5 and lastly conclusions and future work.

## 2. Programming with Concepts in C++0x

This section provides a brief and incomplete overview of the concept model proposed for C++0x[1]. Initial definitions and discussion pertain to the most fundamental concepts present in the proposal of the new version. A more thorough treatment of the material is presented in [Gregor'06]. Note that the concepts definitions described in this paper may not match the proposals due to abbreviations, summaries, and omissions.

One of the primary goals of using concepts with generic algorithms is to constrain the set of types over which a template can be instantiated. Consider a simple implementation of the `std::fill` algorithm, shown in Figure 1.

```
template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x) {
  for( ; f != l; ++f) {
    *f = x;
  }
}
```

**Figure 1. A simple implementation of the generic `std::fill` algorithm.**

Experienced developers should immediately recognize the function template can be instantiated by most sequences providing an iterator (e.g., `std::vector`) and pointers into arrays (e.g., `int*`). Likewise, an experienced developer would be able to tell you that the template cannot be instantiated over an integer as the `Iter` template parameter because integers are not dereferenceable (i.e., `*f` is a compiler error if `Iter` is an `int`). However, there are a number of subtleties in this template that are not immediately obvious. For example, this algorithm cannot be instantiated over standard pair-associative containers (e.g., `std::map`) due to the expression `*f = x`. Here, we dereference the iterator `f` and assign that result to the value `x`. If `*f` results in a key-value pair of a `std::map`, the assignment will give a compiler error because the key is constant.

In C++0x, developers will be able specify these constraints using *concepts*. A concept is the specification of a set of requirements for a set of types. For example, the expression `f != l`, in our `fill` example, indicates that there is an inequality operator for the `Iter` template parameter. Likewise, the `++f` is requirement for a preincrement operator, and `*f` requires `Iter` to be dereferenceable. The expression `*f = x` is more complex in that it requires the result type of dereferencing operator to be assignable to an object of type `T`.

[1] See http://www.open-std.org/JTC1/SC22/WG21/ for the complete concepts proposal for C++0x.

Each of these requirements corresponds to one or more concepts proposed for C++ standard. For example, Figure 2 shows the dereferenceable concept definition. The `typename` statement requires that `T` must provide a type name for the return type of its dereferencing operator (i.e., `reference`). The name `reference` is called an *associated type* because the actual type denoted by that name should be somehow derived from the type `T`. If the concept instance `Deref<Iter>` is valid (i.e,. `Iter` meets these requirements), then we can refer to the return type of the dereferencing operator as `Iter::reference`. The definitions for the other concepts can be found in the C++0x proposal.

```
concept Deref<typename T>
{
  typename reference;
  reference operator*(T);
};
```

**Figure 2. The proposed concept definition for dereferenceable types.**

Given these concepts above, one possible redefinition of the `fill` algorithm as shown in Figure 3. We can now modify the definition of `fill` to explicitly constrain the selection of types over which the template can be instantiated. This is done using a *requirements clause*. The requirements clause (see `requires` in the template body) is a conjunction of concept instances that completely express all requirements in the subsequent function body. The choice of concept instances being used is largely up to the programmer since there are many ways to express the same requirements.

```
template <typename Iter, typename T>
 requires
  Arith<Iter> &&
  Deref<Iter> &&
  Assign<Deref<Iter>::reference, T>>
void fill(Iter f, Iter l, typename T)
{ ... }
```

**Figure 3. A new definition for the fill algorithm that lists requires the satisfaction of concept instances in order to be instantiated.**

Although this sequence of concept instances does cover the requirements of the `fill` algorithm, it fails to capture the semantics of the types involved. As programmers we should implicitly understand that the `Iter` parameter represents an *iterator* and not simply some arithmetic-like object that also happens to be dereferenceable. While this requirements clause fails to capture the true semantics of the template parameters in this function, it is not technically incorrect. Any concept identification methods should attempt to distinguish technically and semantically correct specifications.

## 3. Identifying Concepts

Our approach to the automatic identification of concepts begins at the source level, extracting operations and types (requirements) associated with expressions that involve template parameters. Subsequent analyses match these requirements to concepts, refine the candidates, and finally compute the suggested data abstractions (concept instance) for each template parameter. The specific stages in this approach are:

- *requirement extraction* – The function template is analyzed for requirements, producing a list of requirements.
- *concept matching* – Extracted requirements are matched against a set of concept definitions to find those that express the given requirements.
- *concept instantiation* – Matched concepts are instantiated by correlating types in the extracted requirements to those in the concept definitions.
- *refinement search* – The refinement hierarchy is instantiated and searched to find additional candidate concept instances.
- *formal concept analysis* – A binary relation is constructed over the set of concept instances and their requirements and is used to generate a partial ordering of concept instances and their requirements.

The concept instances that represent the data abstractions of the template parameters to this function can be easily extracted from this lattice.

### 3.1. Requirement Extraction

The first phase of the concept identification algorithm starts with the bottom-up task of extracting requirements within the source algorithm. Like type inference engines, we traverse the AST of the function template. When the traversal encounters an expression involving operands whose types involve template parameters, we formulate a *type requirement* for the expression.

We define a type requirement as a function declaration formulated over the types or nested type names in the expression. The names of functions and types are encoded during this phase to reduce the possibility of ambiguities (e.g., unary/binary operators, free/member functions). Note that these function declarations only include a return type when the return is required by another expression. Consider, from the `fill` algorithm, the expression `f != l`. Because this expression is in the condition clause of the for loop, we can anticipate that its return type is `bool`. The entire expression results in the type requirement `neq(Iter, Iter)->bool`. Conversely the expression `++f` results in the requirement `preinc(Iter)`, with no return type.

In many cases, the types present in an expression depend on the result types of other expressions. Since we cannot generally deduce the return type of an expression, the types used in the formulation must be symbolically written. The expression `*f = x` is an example of type dependency between statements, consisting of dereferencing `f` and assigning the value `x` to that result. The resulting requirements are `operator*(Iter)->*Iter` and `assign(*Iter, T)`. Here, the left-hand type is depends on the result of the dereference operator, and is generated by prefixing the operator with the type involved, creating a unique type name for result. Other dependent requirements can be similarly derived.

A complete table of type requirements for the fill algorithm is given in Table 1.

**Table 1. Expressions and their associated requirements extracted from the `fill` algorithm.**

| *Expression* | *Requirement* |
|---|---|
| `Iter f, l` | `Iter::ctor(Iter)` |
| `f != l` | `neq(Iter, Iter)->bool` |
| `++f` | `preinc(Iter)` |
| `*f` | `deref(Iter)->*Iter` |
| `*f = x` | `assign(*Iter, T)` |

### 3.2. Concept Matching

Having extracted requirements, we search the *concept repository* to find a set of concept definitions that express the same requirements. The concept repository is a collection of concept definitions, their refinements, associated functions, types and requirements. The concept repository contains the concept definitions proposed for the STL, but can be expanded to contain concept definitions for any domain (e.g., graph data structures and algorithms).

We begin by preprocessing the repository. For each definition in the repository, we instantiate associated (nested) requirements and add any resulting function definitions as members of the nesting concept definition. This has the effect of inserting related operations on associated types into the scope of these concept definitions. Also, each operator, constructor and destructor is also rewritten using the same symbolic using the approach given above. The results of this preprocessing on the `OutIter` concept are shown in Figure 4.

```
concept OutIter<X> : IterBase<X> {
  X& preinc(X&);
  reference deref(X&);
  reference& assign(
    reference&, const value_type&);
};
```

**Figure 4. A partial listing of the preprocessed `OutIter` concept definition.**

Finally, a mapping is created that associates the name of each concept member (functions and types) with the set of concepts that include them. Given this mapping it is now a trivial task to match the type requirements in Table 1 to a corresponding set of concepts.

**Table 2. The initial set of concepts that match the type requirements extracted from the `fill` algorithm.**

| *Requirement* | *Concepts* |
|---|---|
| `Iter::ctor(Iter)` | `CopyCtor` |
| `neq(Iter, Iter)->bool` | `Equal` |
| `preinc(Iter)` | `Arith, InIter, OutIter` |
| `deref(Iter)->*Iter` | `Deref, InIter, OutIter` |
| `assign(*Iter,T)` | `Assign, OutIter` |

The mapping from type requirements to concept definitions is then inverted such that each concept is now mapped to the set of type requirements that it describes and their corresponding set of concept members. This allows us to work with only the relevant members of each concept with respect to the original function. The results of this initial matching for the `fill` algorithm are given in Table 3.

**Table 3. An inversion of the mapping in Table 2, concepts are mapped to the set of requirements that they may express.**

| *Concept* | *Requirements* |
|---|---|
| `CopyCtor` | `Iter::ctor(Iter)` |
| `Equal` | `neq(Iter,Iter)->bool` |
| `Deref` | `deref(Iter)->*Iter` |
| `Arith` | `preinc(Iter)` |
| `InIter` | `preinc(Iter), deref(Iter)->*Iter` |
| `OutIter` | `preinc(Iter), deref(Iter)->*Iter, assign(*Iter, T)` |
| `Assign` | `assign(*Iter, T)` |

It is important to realize that the concepts in Table 3 can be mapped to requirements on different types. This problem will be resolved during concept instantiation.

## 3.3. Concept Instantiation

Concept instantiation is similar to template instantiation in that concrete types are substituted for concept parameters, resulting in a mechanism that can be used to validate or specialize the concrete types within the context of a function template. Unlike traditional instantiation processes, which might be described as "top-down", our approach requires us to implement "bottom-up" instantiation. We cannot be sure which template parameters are going to being used as arguments to candidate concepts, or whether or not the concept instance can be generated correctly.

The inputs to the bottom-up instantiation algorithm are a concept definition and a set of type requirements. From these inputs, we must deduce a) the number of distinct concept instances represented by the set of type requirements and b) the types that will act as arguments for concept parameters. A general outline of the algorithm follows:

Analyze each concept identified in the previous stage in order. For each concept, sort its associated set of requirements. This orders requirements with no computed types before those that return computed types before those that take computed types as arguments. For example, the order of requirements for the `OutIter` concept is `preinc`, `deref`, and `assign`. This ordering helps iteratively construct instances by reducing the probability that a computed type (e.g., `*Iter`) will not be associated with a type given in the concept member (e.g., `reference`).

We can now instantiate a set of concept instances with respect to the current concept definition and its set of type requirements. Create an empty set of pseudo-instances *X*. A pseudo-instance is like a scratch pad that stores additional information and can be used to instantiate a concept definition. Specifically these structures store mappings between types in the concept definition and those in the type requirements. For each type requirement, create a new pseudo-instance *x*. Identify the concept member that corresponds to the current requirement (i.e., has the same name). Perform a pairwise comparison of the types in the requirement (call these types set *S*) with those present in the concept member (call types these set *T*). If a type $s \in S$ is compatible with a type $t \in T$, record the mapping of *t* to *s* in the pseudo-instance. If the pseudo-instance is not in *X*, add it. If it is, merge the mapping of types *t* to *s* into the instance *x*.

If types are computed as the result of an operation (e.g., `*Iter`), identify the best candidate instance in *X* and try to identify a corresponding type. Because of the ordering of requirements, it is more likely that a mapping from the computed type to one in the concept definition will have been recorded (as a return type). If such a mapping exists, no more work need be done.

Type compatibility is essentially determined by the rules of type conversion in C++. For example, a `const` reference or pointer cannot be implicitly converted to a non-`const` reference or pointer. Objects and built-in types are not implicitly convertible, etc. If, at any point, a requirement has a type that is not type compatible with one in the concept definition and hence not mapped to anything, the current pseudo-instance is rejected.

After evaluating all requirements, the set *X* will contain a set of potential pseudo-instances for the original algorithm. A pseudo-instance can be instantiated if each concept parameter in its concept definition is mapped to a type present in one of the type requirements. If so, the pseudo-instance generates a concept instance over those mapped types.

Suppose, for example, that we are instantiating the `CopyCtor` concept over its requirement, `Iter::ctor(Iter)`. We start by creating a pseudo-instance for this requirement. We must now compare this requirement to its corresponding concept member `T::ctor(const T&)`. Here, `Iter` is compatible with `T`, so we add it to mapping for the pseudo-concept. The next type `Iter` is also compatible with the parameter `const T&`, but the mapping is already made so no action need be taken. By iterating over the concept parameters in the definition, we find that the only parameter `T` is mapped to `Iter` so we can create the instance `CopyCtor<Iter>`.

As this algorithm operates on the intricacies of the C++ program model, it goes without saying there are a number of corner cases that can best be solved heuristically. For example, instances are not rejected simply because they have a computed type as an argument. Also, arguments to concepts with default parameters are telescoped when applicable. This is to say that if the name of the argument is the same as the argument referred to by the default parameter, then the argument list is truncated on the first such occurrence. But, this does not happen if the type of the default argument is computed, etc. The final set of concept instances for the `fill` algorithm is:

```
CopyCtor<Iter>, Equal<Iter>, Deref<Iter>,
Arith<Iter>, InIter<Iter>, OutIter<Iter>,
Assign<*Iter, T>.
```

## 3.4. Refinement Search

Unfortunately, the best description of a template parameter's data abstraction may not be identified during concept matching and instantiation. Instead, it may be a refinement of one of the previously identified concept instances. This stage of the concept identification process is to search the refinement hierarchy for additional candidates. The refinement hierarchy is taken directly from the refinement relations in the concept repository referenced earlier. We want to constrain the inclusion of refinements in our search to exclude those that represent "redundant" expressions of required functions and types.

To do so, we consider the reverse graph of the hierarchy defined by refinements in the concept repository such that each edge is directed from the refined concept to the refining concept. Each of the concept instances in the previous stage is used to generate an instantiation of this graph by propagating instantiations along these edges. The result is a set of disjoint directed acyclic graphs that represent the possible refinements of all concept instances identified in the previous stage. Two new vertices, *top* and *bottom*, are added to the graph such that *top* is connected to the roots of each disjoint subgraph, and the terminal nodes in each subgraph are connected to *bottom*. This creates a lattice over the set of concept instances. Figure 5 shows the subset of this lattice containing the iterator concept instantiations for the fill algorithm.

Concept instances that contain non-redundant expressions of requirements are found at the meet (least upper bound) of combinations of the instances identified in the previous stage. All such concept instances are found by computing the meets for the powerset of the original instances. The bottom element is disregarded since it does not actually represent a concept instance.

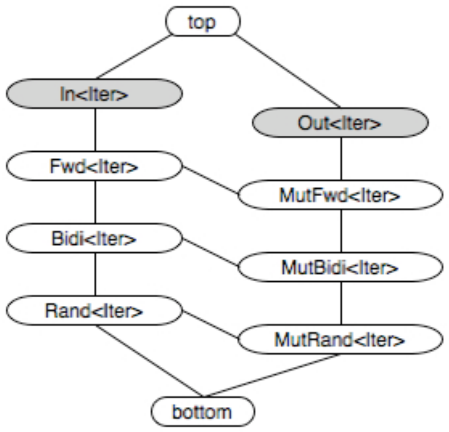


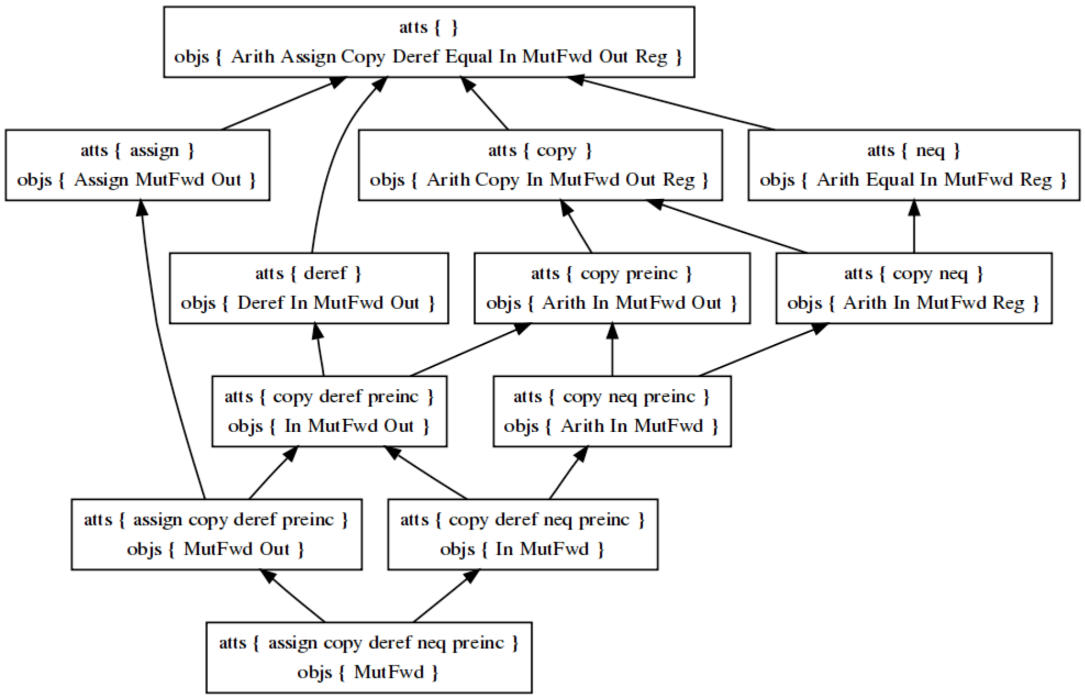
**Figure 5. The refinement lattice for the `Iter`-based concept instances. Concept names are abbreviated, and initially matched concepts are shown in gray.**

For example, the `MutFwdIter<Iter>` concept instance will be included as a candidate because it is a meet of a refinement of the `InIter` and `OuIter` concepts. However, the `MutBidiIter` concept, also a refinement of `InIter` and `OutIter`, is not a suitable candidate because it is not the least upper bound of any initial instances. Computing the meets of the powerset of inputs yields the final set of candidates:

```
CopyCtor<Iter>, Equal<Iter>,
Regular<Iter>, Deref<Iter>, Arith<Iter>,
InIter<Iter>, OutIter<Iter>,
MutFwdIter<Iter>, Assign<*Iter, T>.
```

This algorithm has added two new candidates: `Regular<Iter>` and `MutFwdIter<Iter>`. The `Regular<Iter>` instance is added because it is the meet of the `CopyCtor<Iter>` and `Equal<Iter>` instances. `MutFwdIter<Iter>` is added for the reasons described above.

Note that when the refinement hierarchy is instantiated, we also propagate the requirements expressed by each instance to their refinements. At the end of this stage of the algorithm, each resulting concept instance will contain all of the requirements that it represents within the original function template.



atts { }
objs { Arith Assign Copy Deref Equal In MutFwd Out Reg }
atts { assign }
objs { Assign MutFwd Out }
atts { copy }
objs { Arith Copy In MutFwd Out Reg }
atts { neq }
objs { Arith Equal In MutFwd Reg }
atts { deref }
objs { Deref In MutFwd Out }
atts { copy preinc }
objs { Arith In MutFwd Out }
atts { copy neq }
objs { Arith In MutFwd Reg }
atts { copy deref preinc }
objs { In MutFwd Out }
atts { copy neq preinc }
objs { Arith In MutFwd }
atts { assign copy deref preinc }
objs { MutFwd Out }
atts { copy deref neq preinc }
objs { In MutFwd }
atts { assign copy deref neq preinc }
objs { MutFwd }

**Figure 6. The concept lattice generated for the concept instances and type requirements in the `fill` algorithm. In this representation, concept names are abbreviated, function names represented symbolically, and same-type constraints are hidden.**

## 3.5. Concept Analysis

We now have a set of concept instances that can be used to generate requirement clauses by combining them in various ways. However, we would like to identify the *best possible* requirement clause that includes concept instances that are neither overly specific nor generic. Ironically (because the word "concept" has not been sufficiently overloaded), we rely on formal concept analysis (FCA) to compute a lattice that a) provides a mechanism for constructing requirement clauses and b) orders them in such a way that the best requirement clauses are ordered before less appropriate ones.

Formal concept analysis is an information analysis technique that is used to show relations between a set of objects and the attributes they may share in common [Eisenbarth'01, Snelting'96, '05]. Formally defined, a formal context $C = (O, A, R)$ is defined as a triple, consisting of a set of objects $O$, a set of attributes $A$, and a binary relation $R$ that associates objects in $O$ with their attributes in $A$. A formal concept $c = (\{a_i\}, \{o_i\})$ is defined as the maximal matching between a set of objects $\{o_i\}$ and attributes $\{a_i\}$ such that all objects $o_i$ possess only and all attributes $a_i$. Equivalently, only attributes $a_i$ are possessed by exactly the objects $o_i$. The set $\{o_i\}$ is the *extent* of the concept and the set $\{a_i\}$ is its *intent.*

Formal concept analysis computes a partial ordering over formal concepts based on the subset relation of objects and attribuets. The top of the lattice (derived from the partial order) contains the set of all objects that possess no attributes, and the bottom formal concept contains all objects that possess all attributes.

In this context, we are using formal concept analysis to provide an ordering of concept instances (requirements), the best of which will express maximally disjoint subsets of type requirements in the original function template. To do so, we let the previously computed sets of concept instances be the set of objects $O$, and the set of all requirements originally extracted from the function template be the set of attributes $A$. The relation $R$ is constructed such that $(o, a) \in R$ iff instance $o$ expresses requirement $a$. The relation $R$, computed for the fill algorithm is shown in Table 4. This is used to construct the lattice shown in Figure 6.

The resulting concept lattice provides the means for computing the best requirements for the original function template. Fortunately, the methods that we have employed as inputs to the construction of this lattice result in lattices with some interesting properties.

Because FCA orders lattices with respect to the maximal subsets of object attribution, the extents of formal concepts toward the bottom of the lattice will contain concept instances that express the most requirements within the context of the original function. The net result is that concept instances in these extents will accurately represent the data abstractions implied by the template parameters of the template function.

Consider the lattice in Figure 6, which represents the final analysis of the `fill` algorithm. Recall from Figure 1 that `fill` has two template parameters, but only one with any concrete type requirements, `Iter`. The minimum (bottom) concept in this lattice is the concept containing the `MutFwd<Iter>` instance in its extent. This is the best possible expression of requirements for the `Iter` template parameter given the requirements present in the algorithm, and is, in fact, the concept required by the `std::fill` algorithm in the C++0x proposal.

**Table 4. The relation mapping concept instances to requirements of the `fill` algorithm. Concept arguments and types are omitted for brevity.**

| | `copy` | `neq` | `preinc` | `deref` | `assign` |
|---|---|---|---|---|---|
| `Copy` | X | | | | |
| `Equal` | | X | | | |
| `Deref` | | | | X | |
| `Reg` | X | X | | | |
| `Arith` | X | X | X | | |
| `In` | X | X | X | X | |
| `Out` | X | | X | X | X |
| `MutFwd` | X | X | X | X | X |
| `Assign` | | | | | X |

The solution to this problem will not always manifest in the extent of the bottom concept. In fact, this will not be the case for most generic algorithms. In the presence of multiple, distinct data types, the bottom concept will have an empty extent. The immediate superconcepts of the bottom will, however, contain the concept instances that best describe the abstract data types of the different template parameters, and the union of their intents will be the set of all requirements. Said more simply, the solution to this problem will always be found in extent of the bottom concept or those just above it.

More formally, we claim that the solution – the set of concept instances that best describe the template parameters within the scope of a given function – are found in the extent of the least concept of each disjoint, bounded sublattice whose greatest and least elements are immediately less than the top and greater than the bottom, respectively. The proof of this follows from the definition of formal concepts and the means by which they are ordered.

Note that the truly "best" possible solution would be the extent of the least formal concept containing a single concept instance. This will unambiguously identify the abstract data type of a template parameter within the context of its usage and the definition of the concept repository. Unfortunately, we cannot guarantee the uniqueness of such solutions. For example, consider the trivial standard function `advance`, which moves an iterator to the next element (i.e., whose trivial implementation is `++i`). Given the standard concept hierarchy, our approach will be unable to distinguish between the concept instances `Arith<T>`, `MutFwdIter<Iter>`. The former concept is identified since it explicitly lists the preincrement operator as a requirement, while the latter is identified because it is a refinement of both `InIter` and `OutIter`.

## 4. Evaluation

We have implemented a prototype of our concept identification algorithm as a pipe-and-filter style tool with two components. The first component, based on srcML [Collard'03, Sutton'07b], implements the requirement extraction process and enumerates a listing of type requirements for a function template. The output of this program is piped to the second, which implements the analyses responsible for concept identification. The evaluation of our approach targets its applicability to different algorithms, and its stability with respect to variations of the same algorithm.

### 4.1. Applicability

To test applicability, we ran the algorithm against a selection of generic algorithms from the STL, including sequence and numeric algorithms. We implemented these algorithms ourselves in order to eliminate any "interference" from the realities of C++ programming (e.g., the preprocessor, external dependencies, unreadable naming conventions, etc). We compared the results of these tests against the concept requirements proposed for their respective algorithms. A selection of results is shown in Table 5. This selection of algorithms includes a sampling of many of the different forms of STL algorithms: those that operate on a single sequence, those that operate on multiple sequences, and those that employ functors on elements of the sequence.

For the most part, excepting a fair amount of ambiguity, the requirements identifies for these algorithms align well with their proposed requirements. Unfortunately, the results were not as unambiguous as we had originally hoped. For example, our prototype only identifies unambiguous concepts for iterator types that are dereferenced and assigned values (e.g., the `fill` algorithm). The reasons for this are found in the proposed concept hierarchy. The common iterator operations (increment, dereference, etc.) are multiply specified within this hierarchy: in the input and output iterator concepts. The unfortunate result of this duality is the inevitable inclusion of both as well as their mutual refinement, mutable forward iterator, in the candidate set. Without the requirement on assignment through the reference type (as seen in the `fill` algorithm), our approach cannot disambiguate the actual type of iterator.

Moreover, many of the algorithms for which we have ambiguously identified either input iterators or mutable forward iterators, are actually proposed to require (non-mutable) forward iterators – which our implementation unfortunately fails to list. The cause of this problem is also rooted in definition of the standard concepts. As proposed, the forward iterator is a refinement of an input iterator that admits a simple convertibility requirement (`const X& = x++`). Because it defines no new features as associated functions or types, it is never considered as a candidate for inclusion. However, we might observe that it sits on the path between the input and mutable forward iterator in the graph and could be included as a viable candidate.

**Table 5. A selected listing of STL algorithms, their computed concept requirements (with abbreviated names). The requirements of each algorithm is described by a conjunction of concept instances with ambiguities written as disjunctions.**

| *Algorithm* | *Requirement* |
|---|---|
| `fill` | `MutFwdIter<Iter>` |
| `find` | `InIter<Iter>` <br> `|| MutFwdIter<Iter>` <br> `Equal<*Iter,T>` |
| `find_if` | `InIter<Iter>` <br> `|| MutFwdIter<Iter>` <br> `Ctor<Pred,Pred>` <br> `|| CopyCtor<Pred>` <br> `Callable<Pred,*Iter>` |
| `find_end` | `InIter<Iter1>` <br> `|| MutFwdIter<Iter1>` <br> `InIter<Iter2>` <br> `|| MutFwdIter<Iter2>` <br> `Equal<*Iter1,*Iter2>` |
| `for_each` | `InIter<Iter>` <br> `|| MutFwdIter<Iter>` <br> `Ctor<Func,Func>` <br> `|| CopyCtor<Func>` <br> `Callable<Func,*Iter>` |
| `accumulate` | `InIter<Iter>` <br> `|| MutFwdIter<Iter>` <br> `ArithmeticLike<T>` <br> `|| RandAccessIter<T>` |

The misidentification of the predicate functors in the `find_if` function is a result of the same artifact. The predicate concept is defined as a refinement of the callable concept whose return type is convertible to `bool`. Unfortunately, this definition is insufficient for our algorithm to incorporate the refined predicate definition into the set of candidates.

However, some of the more interesting requirements being generated are the constructibility requirements for the functors' types. Because functors are passed (and often returned) by value, they must be copy constructible. The current proposals for the STL do not list any constructibility requirements on generic algorithms taking a functor as an argument. We believe that this is an error of omission in the proposal.

The final ambiguity is that found in the `accumulate` algorithm (generalized summation). Here, the template parameter `T` is described as requiring either an arithmetic-like type or a random access iterator. However, an instantiation of this algorithm with `T` as a random access iterator is almost certainly a logic error and can only work if the dereferenced type of the `Iter` parameter is the reference type of the random access iterator. This error demonstrates an insufficiency in our approach: the filtering of candidates based on allowable or required conversions between their associated types.

## 4.2. Stability

In order to test stability, we ran the algorithm against small variations of the same algorithms. These variations involved either the expansion of single statements into several, or the compression of many statements into fewer. Two variations of the `fill` implementation are shown in Figure 7. The first compresses the increment and dereference operations into a single statement. The second expands the assignment to use an explicitly named associated type.

```
template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x) {
  while(f != l) {
      *f++ = x;
  }
}

template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x) {
  for( ; f != l; ++f) {
      typename Iter::reference r = *f;
      r = x;
  }
}
```

**Figure 7. Two minor variations in the implementation of the `fill` algorithm. The first compresses a number of expressions in the original into a single statement, and the second expands them into multiple statements.**

Because these two variations are semantically equivalent to the original (shown in Figure 1), we expect the concept requirements to remain roughly the same. A small sampling of results is shown in Table 6.

Overall, the stability of our approach across these minor variations is fairly good. Most of the differences that we see in the results have more to do with implicit conversions between types rather than differences in the primary abstractions of the template parameters. In the cases shown here, the analysis fails to correctly associate the associated `reference` type with the return value of the dereference iterator and can be attributed to the explicit declaration of the associated type. Correcting this flaw will cause algorithm to determine that the `Iter` parameter must model a mutable forward iterator.

Evaluation of other algorithms yields similar results; we often find that minor variations in the implementation result in minor differences in the final results. While some of these variations are the result of implementation issues (as seen before), others occur because the variations restrict or change the requirements in subtle ways. For example our analysis of find variants caused the `Equal<*Iter, T>` requirement to be reduced to `Equal<T>`.

**Table 6. Concept requirements for minor variations on the same algorithm.**

| *Algorithm* | *Requirement* |
|---|---|
| `fill` original | `MutFwdIter<Iter>` |
| `fill` compressed | `MutFwdIter<Iter>` |
| `fill` expanded | `InIter<Iter>`<br>`|| MutFwdIter<Iter>`<br>`Ctor<reference,reference>`<br>`|| CopyCtor<reference>`<br>`Assign<reference,T>` |

### 4.3. Discussion

We had originally hoped that our approach would be capable of extracting unambiguous requirements for these algorithms. Because these algorithms are part of a standard library, we expected their interfaces (and hence their requirements) to be unambiguous and immutable. Instead, we have shown that the results of automatic identification of concepts is dependent upon the unambiguous definition of concept hierarchies, and that minor variations in the implementation of algorithms can cause differences in the resulting abstractions. Far from disappointing, these results show that automated concept identification has tremendous value to developers building or refactoring code into generic libraries.

The existence of ambiguous results during concept identification may indicate problems in the specification of the concept hierarchy. While we make no claims about the incorrectness of the iterator hierarchy as proposed for the STL, the ambiguities found during evaluation certainly cause us to seek rationale for some of its design choices. We believe that tools such as this can be used to help developers better understand their concept design choices and ultimately build better generic libraries.

The generation of different requirements based on minor variations in the implementation of algorithms is also rooted in processes of generic programming. The process of generic lifting – iteratively rewriting concrete algorithms into generic algorithms – requires the study of different concrete algorithms in order to understand the commonalities that can be exploited to produce a generic version. Our evaluation shows that sampling a variety of implementations is vital to the correct specification of requirements for the algorithm. Failing to sample a sufficient variety of implementations can result in overly strict or loose constraints on the implementation and on the types over which they can be instantiated.

## 5. Related Work

The goals of this work are, in some ways, quite similar to those of type inference algorithms [Milner'78]. If we consider concept instances to be the "types" of template parameters, then our algorithm could be seen as a variation of classical type inference. There are, however, fundamental differences in the goals and the problems in these applications. We have not based our analysis on a rigorously defined type system (such as that in [Siek'05]), a crucial component to common inference approaches. Our approach is designed to correlate template parameters with flexibly defined and changing type systems (concept hierarchy) rather than assigning known and fixed types to variables. Unlike type inference algorithms, we are not concerned with the validation of generic programs. The inability of our algorithm to unambiguously infer the type (concept instance) of template parameters is not seen as a failure in validation, but rather an opportunity to discover the means and rationale for disambiguation.

This is not to say that type inference does not have applications for C++ or other statically typed languages. In [Siff'96], the authors describe the application of a type inference engine to rewrite C code into generic (template) C++. A similar approach is applied for Java in [Donovan'04]. While related, these projects use type inference to assist the lifting of concrete source code. Our approach is targeted solely on the identification of concepts within existing generic algorithms.

With regard to reverse engineering, there is comparatively little research on the analysis of C++ templates, much less C++ concepts. The primary reason for this is that C++ poses difficultly for reverse engineering tools because of preprocessing [Sutton'07a] and templates [Kraft'07]. Taken at face value, template parameters have no type, which makes it difficult to connect them to concepts in the application domain. In fact, templates only become "real" code when they are instantiated over concrete types, and many applications are only capable of seeing the instance but not the template [Dean'01, Kraft'07] [Ferenc'02]. This is also true of tools such as Doxygen and GCC-XML.

However, some tools such as Microsoft's Intellisense and other code completion tools are capable of indexing nested members of templates. Similarly, TUAnalyzer, is capable of extracting template specifications from GCC translation unit dumps in order to identify and associate them with function invocations [Gshwind'04]. STLlint employs a high-level static analysis to detect various usage errors in user code [Gregor'05]. Another work describes a conceptual change impact analysis model [Zalewski'05, '06]. This model determines the impact of changes to a concept taxonomy on the generic algorithms in its domain.

## 6. Conclusions and Future Work

In this paper, we have described an automated approach to the identification of concepts in existing C++ function templates. We have built a prototype implementation and ran it against a number of generic algorithms found in the STL. The results validate the usefulness of the approach for assisting in the reengineering and migration of current C++ template libraries to use the new concept features of C++0x. Such tools will help alleviate large amounts of manual work determining and validating concepts in function templates, which can be error-prone and tedious.

Clearly, there is a significant amount of research still to be done in the domain of reverse engineering of generic libraries. One such goal is the further investigation of and harmonization with work in type inference since one can conceivably consider concepts analogous to types in template functions. Another is the extension of our analyses to a better model with implicit conversions and other subtle relationships between variables in function templates. These augmented models will help increase the accuracy of our analyses.

## 7. References

[Collard'03] Collard, M. L., Kagdi, H., and Maletic, J. I., (2003), "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), OR, May 10-11, pp. 134-143.

[Dean'01] Dean, T. R., Malton, A. J., and Holt, R. C., (2001), "Union Schemas as a Basis for a C++ Extractor", in Proceedings of Eighth Working Conference on Reverse Engineering (WCRE'01), Germany, Oct 2-5, pp. 59-70.

[Donovan'04] Donovan, A., Kiežun, A., Tschantz, M. S., and Ernst, M. D., (2004), "Converting Java Programs to Use Generic Libraries", *SIGPLAN Notices*, vol. 39, no. 10, Oct, 2004, pp. 15-34.

[Eisenbarth'01] Eisenbarth, T., Koschke, R., and Simon, D., (2001), "Derivation of Feature Component Maps by means of Concept Analysis", in Proceedings of Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, March 14 - 16, pp. 176-179.

[Ferenc'02] Ferenc, R., Magyar, F., Beszedes, A., Kiss, A., and Tarkiainen, M., (2002), "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems", in Proceedings of SPLST 2001, June 2001, pp. 16-27.

[Gregor'05] Gregor, D. and Schupp, S., (2005), "STLlint: Lifting Static Checking from Languages to Libraries", *Software: Practice and Experience*, vol. 36, no. 3, Mar 2006, pp. 225-254.

[Gregor'06] Gregor, D., J ärvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A., (2006), "Concepts: Linguistic Support for Generic Programming in C++", in Proceedings of Object-Oriented Programming, Systemls, Languages, and Applications (OOPSLA'06), OR, Oct 22-26, pp. 291-310.

[Gshwind'04] Gshwind, T., Pinzger, M., and Gall, H., (2004), "TUAnalyzer - Analyzing Templates in C++ Code", in Proceedings of 11th Working Conference on Reverse Engineering, The Netherlands, Nov 8-12, 2004, pp. 48-57.

[Kraft'07] Kraft, N., Malloy, B., and Powers, J., (2007), "A Tool Chain for Reverse Engineering C++ Applications", *Science of Comp Prog*, vol. 69, no. 1-3, Dec, 2007, pp. 3-13.

[Milner'78] Milner, R., (1978), "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348-375.

[Siek'05] Siek, J. and Lumsdaine, A., (2005), "Language Requirements for Large-scale Generic Libraries", in Proceedings of 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, Sep 29-Oct 1, 2005, pp. 405-421.

[Siff'96] Siff, M. and Reps, T., (1996), "Program Generalization for Software Reuse", in Proceedings of 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'96), San Francisco, California, Oct 16-18, 1996, pp. 135-146.

[Snelting'96] Snelting, G., (1996), "Reengineering of Configurations based on Mathematical Concept Analysis", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, April, 1996, pp. 146-189.

[Snelting'05] Snelting, G., (2005), "Concept Lattices in Software Analysis", in *Formal Concept Analysis: Foundations and Applications*, B. Gantner, G. Stumme and R. Wille, Eds., pp. 272-287.

[Sutton'07a] Sutton, A. and Maletic, J. I., (2007a), "How We Manage Portability and Configuration with the C Preprocessor", in Proceedings of 23rd International Conference on Software Maintenance (ICSM'07), Paris, France, Oct 2-5, 2007, pp. 275-284.

[Sutton'07b] Sutton, A. and Maletic, J. I., (2007b), "Recovering UML Class Models from C++: A Detailed Explanation", *Information and Software Technology*, vol. 49, no. 3, Jan 2007, pp. 212-229.

[Zalewski'05] Zalewski, M. and Schupp, S., (2005), "Changing Iterators with Confidence: A Case Study of Change Impact Analysis Applied to Conceptual Specifications", in Proceedings of Workshop on Library-centric Software Design (LCSD'05), San Diego, California, Oct 16, 2005.

[Zalewski'06] Zalewski, M. and Schupp, S., (2006), "Change Impact Analysis for Generic Libraries", in Proceedings of 22nd Conference on Software Maintenance (ICSM'06), Philadelphia, PA, Sep 24-27, 2006, pp. 35-44.