

# Change Impact Analysis for Generic Libraries

Marcin Zalewski and Sibylle Schupp  
Dept. of Computer Science and Engineering  
Chalmers University of Technology  
Göteborg, Sweden  
{zalewski,schupp}@cs.chalmers.se

## Abstract

Since the Standard Template Library (STL), generic libraries in C++ rely on concepts to precisely specify the requirements of generic algorithms (function templates) on their parameters (template arguments). Modifying the definition of a concept even slightly, can have a potentially large impact on the (interfaces of the) entire library. In particular the non-local effects of a change, however, make its impact difficult to determine by hand. In this paper we propose a conceptual change impact analysis (CCIA), which determines the impact of changes of the conceptual specification of a generic library. The analysis is organized in a pipe-and-filter manner, where the first stage finds any kind of impact, the second stage various specific kinds of impact. Both stages describe reachability algorithms, which operate on a conceptual dependence graph. In a case study, we apply CCIA to a new proposal for STL iterator concepts, which is under review by the C++ standardization committee. The analysis shows a number of unexpected incompatibilities and, for certain STL algorithms, a loss of genericity.

## 1. Introduction

Arguably one of the best known generic libraries is the Standard Template Library of C++ (STL) [17, 22]. From a practical point of view, STL provides a collection of generic containers and generic algorithms to operate on containers. However, it is the introduction of so-called *concepts*, underlying the organization of algorithms and containers, that is the outstanding contribution of STL. What characterizes “STL-style programming”, or generic programming, is therefore not just the exploitation of C++ templates but, more importantly, the development of concepts and conceptual specifications for a particular application domain. Although concepts can be formalized (type-) theoretically and in a language-independent way [15, 21, 26], most de-

velopers are familiar with them as a way to constrain C++ templates: instead of designing an algorithm merely in terms of (universally quantified) template parameters, the designer can use a concept to attach additional requirements on the types that instantiate a template. These requirements can be syntactic, semantic, or behavioral, but they always are abstractions from types. In the following, we denote by *conceptual specification* the concept taxonomy and the concept-constrained interfaces that define the specification of a generic library.

Although concepts on the whole ease the maintenance of a library, their very nature introduces some complications one has to be aware of when a particular concept is modified. For one, each concept describes a generic specification of a family of types, which is met by many concrete types at the implementation level of a library. A modification in a concept definition can therefore affect many types and algorithms with parameters that are constrained by the modified concept. Second, in contrast to abstract classes in object-oriented programming, a concept does not aim at capturing the complete interface of a type. Instead, it defines a coherent set of requirements, which a concrete type must meet at the minimum. As a consequence, modifications in a concept definition might go unnoticed for a long time just because all concrete types that instantiate the affected template, incidentally meet both the old and new concept specification. In fact, our initial interest in change-impact questions comes from a practical problem. When algorithms were changed to use STL allocators, the allocator concept introduced an additional requirement “Default-Constructible.” Yet, all types we used with the modified algorithms happened to be default-constructible; only later, and then unexpectedly, code broke when used with types that did not provide a default constructor. Finally, algorithms might be affected by a change in a concept even if none of their parameters is directly constrained by that concept. The non-local impact is propagated by the *refinement relation* between concepts, where one concept includes the requirements of one or more other concepts.

In this paper we propose a *conceptual* change impact analysis (CCIA), which determines the impact of changes of the conceptual specification of a generic library. The analysis is organized in a pipe-and-filter manner, where the first stage finds, for any part of a conceptual specification, all changes it is impacted by. Subsequent optional filters on the second stage further refine the output in various ways, to detect specific kinds of impact. Presently, we provide two filter algorithms: one to detect the impact of changes on the compatibility of different versions of concept specifications, and one to detect the impact of changes to the degree of genericity of an algorithm, that is, the number of types with which it can be instantiated. It is possible to extend the analysis by additional filters and stages.

We applied CCIA to one of the most fundamental changes to STL-like libraries that is currently under consideration by the C++ standardization committee: the proposed change of iterator concepts [19]. As our analysis can show, this proposal unintentionally introduces a number of incompatibilities (between old and new iterator concepts) and renders several STL algorithms less generic than before.

Recent discussions indicate that, “in some form or other” (see [4]), C++ will be extended by direct support for concepts [8, 20, 23]; indeed, our work is partly motivated by this discussion, as we are interested in providing good tools for concepts once they have become first-class citizens. Currently, however, concepts are described in an informal way, using structured natural language. For CCIA, the current state of affairs implies that the analysis cannot yet be completely automated: the task of transforming an informal conceptual specification into a machine-readable format must be done manually. The change impact analysis itself, though, is automated.

The outline of the paper is as follows: Section 2 compares CCIA with other change impact analyses, Sections 3 and 4 provide background on concepts in C++ and give a complete analysis example. The analysis itself is presented in Section 5 and demonstrated in a case study in Section 6. Sections 7 and 8 provide conclusions and an outlook on further work.

## 2. Related Work

An important classification criterion for change impact analyses are the sources for non-locality of change impact. In object-oriented programming, for example, non-locality of impact is introduced by inheritance, dynamic binding, aggregation, and polymorphism [16]. In imperative programs, the impact of changes may be propagated through side effects such as assignment to, and use of variables [25]. In the case of generic libraries, there are two major sources of non-locality: the separation between concepts at

the specification level and types and type parameters at the implementation level; and the refinement relation between concepts, through which changes propagate.

When the sources of non-locality correspond to programming constructs, the impact of changes is propagated by relations established at the source code level of a program. A large class of analyses therefore builds on top of program dependence graphs or system dependence graphs [13]. Analyses that operate not exclusively at source code level, on the other hand, but are concerned with models, ontologies, or specifications [3, 12], construct their own, tailored representation; in our case, the *concept dependence graph*, which represents the concept specification, the (concept-constrained) parameters of generic algorithms, and the changes in both.

Another axis of classification describes the kinds of impact that the analysis identifies. Most analyses are devised for a particular purpose, for example, to identify which regression tests to rerun or which documentation to update [2, 5]. Fewer analyses (most notably Han’s [11]) can be customized by the client. Our analysis falls in between: the analysis can be used to detect different kinds of impacts but is not yet organized in a freely customizable framework. The analysis itself is an instance of a reachability analysis, that is, it traverses the underlying dependence graph appropriately. Like in most change impact analyses, reachability, thus impact, is purely syntactically defined. One could envision a semantic analysis, which checks, e.g., whether two syntactically different sets of requirements are semantically equivalent. Yet, for the current prototype, we consider the gain too low compared to the effort that a semantic analysis incurs, in particular since most applications of CCIA are expected to be “processed” by a human client.

Finally, in some analyses, only the final impact is of interest, not the single propagation steps. In the delta debugging technique [27], for example, where program failure is of interest, the changes causing program failure are detected by means of an efficient search, without further consideration of the ways the change has been propagated. CCIA, in contrast, can be used by concept developers who want to understand how a change has caused a specific impact.

## 3. Concepts in C++

Presently, concepts in C++ are specified in a de-facto standardized documentation. In illustration, Figure 1 shows the description of the STL `BIDIRECTIONALITERATOR` concept adopted from the documentation of STL [1]. Of particular interest for the paper are the rows “Refinement of” and “Associated Types”, and the table “Valid Expressions.” The refinement relation specifies that for a type to model (i.e., fulfill all requirements of) the `BIDIRECTIONALITERATOR` concept, it must also model the `FORWARDITERATOR`

**Description** The `BIDIRECTIONALITERATOR` concept defines types that support traversal over linear sequences of values, including support for multi-pass algorithms, and stepping backwards through a sequence (unlike `FORWARDITERATORS`). A type that is a model of `BIDIRECTIONALITERATOR` may be either mutable or immutable, as defined in the `TRIVIALITERATOR` requirements.

**Refinement of** `FORWARDITERATOR`

**AssociatedTypes** The same as for `FORWARDITERATOR`

**Notation** Let  $X$  be a type that is a model of `BIDIRECTIONALITERATOR`;  $T$  be the value type of  $X$ ;  $i, j$  be objects of type  $X$ ; and  $t$  be an object of type  $T$ .

**Valid Expressions**

Name	Expr.	Return Type
Predecrement	<code>--i</code>	<code>X&amp;</code>
Postdecrement	<code>i--</code>	<code>X</code>

**Expression Semantics**

Expr.	Precond.	Semantics	Postcond.
<code>--i</code>	$i$ is dereferenceable or past-the-end. There exists a dereferenceable iterator $j$ such that $i == ++j$ .	$i$ is modified to point to the previous element.	$i$ is dereferenceable. $\&i == \&--i$ . If $i == j$ , then $--i == --j$ . If $j$ is dereferenceable and $i == ++j$ , then $--i == j$ .
<code>i--</code>	$i$ is dereferenceable or past-the-end. There exists a dereferenceable iterator $j$ such that $i == ++j$ .	Equivalent to <code>{X tmp = i; --i; return tmp; }</code> .	

**Complexity Guarantees** Operations on `BIDIRECTIONALITERATOR` are guaranteed to be amortized constant time.

**Figure 1. Definition of the `BIDIRECTIONALITERATOR` concept of STL [1]**

concept. Associated types are types that must be defined, for example, so that the signatures of the required expressions are well-defined. The table of valid expressions, finally, defines all operations a modeling type has to provide. Two additional requirements, the “Complexity Guarantees” and the “Expressions Semantics” tables, are not included in our impact analysis. The “Expressions Semantics” table, however, is used to precisely encode the informal concept specifications in a machine-usable form (see Section 6.2).

As we pointed out already, the informal nature of concept descriptions prevents full automation of the analysis: representing concepts in an effective format currently needs to be done by hand.

## 4. Example

Instead of plunging directly into technical details, we introduce our analysis by means of an example. Figure 2 shows the original and the modified conceptual specifications of a simple library. The original conceptual specification of the library consists of one concept and one al-

gorithm, `INPUT` and “equal”, respectively. The algorithm “equal” has two type (or template) parameters, “`It1`” and “`It2`”, both constrained by the concept `INPUT`. In the new version of the specification, the `INPUT` concept is modified, 3 new concepts along with a refinement hierarchy are introduced, and the constraints on the type parameters of “equal” are rewritten. For backward-compatibility, the `INPUT` concept is part of the new specification but it only refines the newly introduced concepts and does not have any requirements of its own.

The first step of the analysis, performed manually, is to encode the original and the modified versions of the specification and to compare their differences. The comparison involves identifying entities and relations that exist in the old but not in the new version of the specification, and vice versa. As a result of the first step, for example, the concept `INCREMENTABLE` is marked as *added* since it exists in the new but not in the old version. All further steps execute automatically.

Next, a directed dependence graph is constructed, which represents the two versions of the specification. The vertices correspond to concepts, requirements, and type parameters, while the edges represent the direction of change impact propagation implied by three kinds of relations: concepts including requirements, concepts refining concepts, and concepts constraining type parameters. A dependence graph for the specification in Figure 2 is shown in Figure 3. The graph consists of two type-parameter vertices, four concept vertices of which three are marked as added, and five requirement vertices of which two are marked deleted, one is marked added, and the remaining three are unchanged between the two versions of the library. In the example, all the relations between concepts, requirements, and type parameters have been changed.

The impact analysis proceeds in stages, composed in a pipe-and-filter manner. In the first stage, the impact of the changes is propagated along the edges of the graph to identify the type parameters and concepts that may have been affected. The edges which propagate the impact are marked accordingly for use in subsequent analysis steps.

If one is interested merely in a list of concepts and type parameters impacted by changes, one can terminate the analysis at this point. Otherwise, one of the two currently available filters can be used to determine, for example, whether the concept `INPUT` represents the same set of requirements as before or whether rewriting the constraints on the parameters of the algorithm “equal” increases its genericity. Provided these filters are plugged-in, the analysis yields that the new concept `INPUT` is neither forward- nor backward-compatible with the old concept of the same name: two requirements are deleted (“`*r++`” and “(void) `r++`”), but also a requirement (“`r++`”) is added. Furthermore, the analysis detects that for both type parameters

INPUT	
operation	type
*a	T
++r	X&
(void)r++	
*r++	T
a = b	bool

```

1 template <class It1, class It2>
2 where {Input<It1>, Input<It2>}}
3 bool equal (It1 first1, It1 last1, It2
  first2);

```

(a) Old specification

INPUT	
operation	type
—	—

SINGLEPASS	
operation	type
a = b	bool

```

1 template <class It1, class It2>
2 where {SinglePass<It1>, Readable<It1>,
3       Incrementable<It2>, Readable<It2>}}
4 bool equal (It1 first1, It1 last1, It2 first2);

```

(b) New specification

INCREMENTABLE	
operation	type
++r	X&
r++	X

READABLE	
operation	type
*a	T

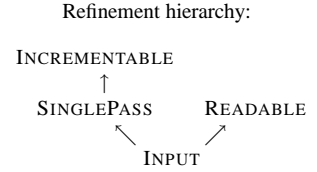


Figure 2. An example of a change in the conceptual specification of a library

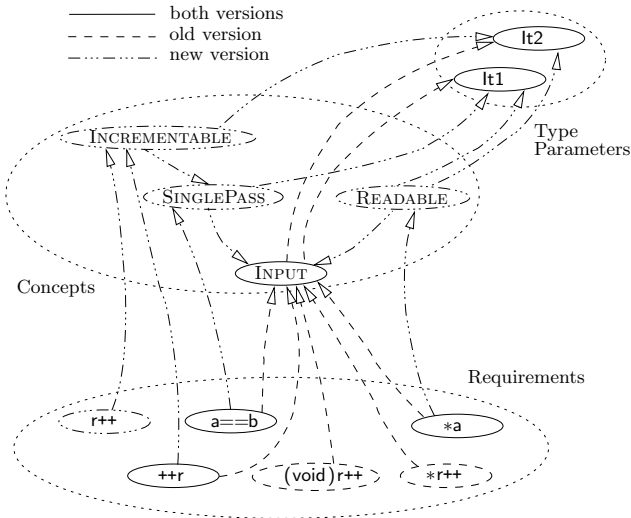


Figure 3. The dependence graph constructed for the specification from Figure 2

of the algorithm “equal” the requirements “(void)r++” and “\*r++” were removed and “r++” was added; for the parameter “It2” additionally the requirement “a == b” is removed. Since requirements on the type parameters are both added and dropped, the analysis can conclude that the algorithm “equal” became neither strictly more nor strictly less generic through the change. Both filters reuse the results of the first stage and only operate on relevant (impacted) parts of the dependence graph.

## 5. Conceptual Change Impact Analysis

As the example in the previous section shows, the three major steps of CCIA comprise the construction of the dependence graph, the first stage of general impact propa-

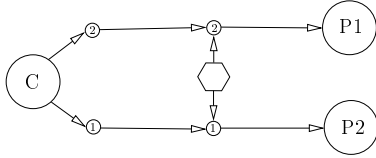
gation, and subsequent filtering for specific impact. The first stage is a forward-reachability problem that determines *any* impact of *any* changes. The second stage, a backward-reachability problem, varies between different applications. The current prototype provides two filters, which capture two frequent questions about conceptual changes: whether concepts are compatible between different versions of conceptual specification and whether the requirements on a type parameter of a generic algorithm changed. This section details each of the three steps.

### 5.1. Conceptual Dependence Graph

An intermediate representation of six constructs suffices to capture the conceptual specification of a library: 3 *entities* and 3 *relations*, directly corresponding to the relations that concepts establish (see Section 3):

- *Type Parameters*: Type parameters of generic algorithms
- *Concepts*: Sets of requirements
- *Requirements*: Operations, associated types, and any other valid expressions (see Fig. 1)
- *Constrains-relations*: Relations between type parameters and concepts
- *Refines-relations*: Relations between concepts
- *Requires-relations*: Relations between concepts and requirements

For example, the concept SINGLEPASS (see Figure 3) constitutes one requires-relations to the requirement “a == b” and one refines-relation to the concept INCREMENTABLE. The type parameter “It1” (see Figure 3) constitutes one constrains-relation to the INPUT iterator concept in the original library and two constrains-relations to the READABLE and SINGLEPASS concepts in the modified library.



**Figure 4. A detailed view of a constrains-relation edge.**

The six constructs of the intermediate representation naturally map to vertices in a graph, where edges capture the dependencies between them. In this graph, any entity or relation that exists in the new version but not in the old one is marked as *added*, and any entity or relation that exists in the old version but not in the new one is marked as *deleted*; any entity or relation that exists in the old version and is modified in the new version is represented by a deletion followed by an addition. In the current prototype, we perform these annotations manually; if concepts were first-class citizens, the annotation could be easily automated by applying a diff-like algorithm to a compiler-provided representation.

We note that the graph in Figure 3 is a simplified view of a dependence graph insofar it represents only single-parameter concepts and requirements. Since concepts and requirements can have multiple parameters, every relation and entity, except for type-parameter entities, explicitly represents its parameters. Figure 4 shows an example of a constrains-relation edge between a two-parameter concept C and two type parameters “P1” and “P2”. The relations and entities in Figure 3 are represented similarly but always happen to have only one parameter vertex.

## 5.2. Impact Propagation

After the graph is constructed, the impact of changes is propagated: any vertex marked as added or deleted has an impact on any of the vertices reachable from it. It is sufficient, however, to only consider relation vertices: a deletion of an entity must be accompanied by deletion of the corresponding relations and addition of an entity does not have a non-local impact unless some relations are added. Depending on the particular kind of impact sought after, addition or deletion of a relation may not be significant. Yet, in general, all vertices reachable from a relation marked as added or deleted are impacted in *some* way.

The algorithm for propagating the impact of changes is rather straightforward. In short, it is a depth-first search where the root vertex of the search is a relation vertex marked as either added or deleted and all the traversed edges are marked as change-propagating. The search stops on deleted edges if the root vertex of the search is marked

added, and on added edges if the root vertex is marked deleted. This condition is necessary to avoid false impact propagation; for example, if a path from an added vertex  $v$  to a vertex  $u$  contains a deleted edge  $e$ ,  $u$  should not be affected by the added vertex. An algorithm that finds all impacting edges for a given vertex  $y$  is similar, but the direction of the search is reversed. In the example from section 4 every relation is marked as added or deleted and thus all edges reachable from relation vertices are marked as propagating. In practice, however, such all-encompassing change of a library will rarely occur and only some edges will be marked as change-propagating.

## 5.3. Filtering for Specific Impact

The impact propagation algorithm identifies *any* impact. Yet, a more specific kind of impact is often of interest. In this subsection, we present two algorithms, *Constraints Change* and *Concept Compatibility*, to detect specific impact. The first algorithm checks whether requirements on algorithm parameters are changed, the second one determines the compatibility of concepts between different versions of a conceptual specification. Both algorithms operate on the level of syntactic requirements. As said earlier, these filter algorithms are based on the result of the preceding impact propagation algorithm and further refine its results. Separating the general impact algorithm from the filters for specific impact, makes the analysis both extensible and scalable: a potentially unlimited number of specific kinds of impact and accordingly specializing filters can be admitted. At the same time, each of these filtering algorithms can restrict itself to impacted vertices. Where only general impact is of interest, filters may be omitted altogether. We now turn to the discussion of the two algorithms.

Informally speaking, algorithm *Constraints Change* computes the change in syntactic requirements on type parameters. The algorithm consists of 2 main steps. First, for all type parameters that are reached by some change (that is, targets of change-propagating edges), the set of reaching changes has to be found. This is done by a search through the reversed dependence graph, traversing only change-propagating edges. Second, for every change that reaches a type parameter, the specific impact on the type parameter has to be computed. For example, if the change reaching some type parameter is a deleted constrains relation, all syntactic requirements implied by the constraining concept have to be marked as removed for that type parameter. A high-level definition of the algorithm follows:

**Algorithm *Constraints Change*.**

*Input:*  $\mathcal{G}$ , a dependence graph where all change-propagating edges are marked;  $T$ , a type parameter vertex.

*Output:*  $R$ , a set of tuples  $(p, S)$  such that  $p$  is a path in  $\mathcal{G}$  from  $T$  to a modified vertex  $q$  and  $S$  is a set of paths from  $q$  to the added or deleted requirements on  $T$  that result from the change in  $q$ .

*Local:* *reaching\_paths*, a container of the results of [Find changes].

*Notation and subroutines:*

*path:* A path in a reversed dependence graph.

*forward or cross edge:* An edge  $(u, v)$  where  $v$  is colored black and not an ancestor of  $u$  in a search tree.

*last():* Given a path, extracts the last vertex.

*significant\_vertex():* Given a parameter vertex, finds the main vertex of the corresponding relation or entity.

**A1.** [Filter and revert  $\mathcal{G}$ .]  $\mathcal{G}'$  is  $\mathcal{G}$  with all edges reversed and non-propagating edges removed.

**A2.** [Find changes.] Run depth-first search on  $\mathcal{G}'$  with  $T$  as the root vertex:

**A2.1.** If a vertex marked as added or deleted is discovered, record current path in *reaching\_paths*, mark vertex black, and backtrack depth-first search.

**A2.2.** If a forward or a cross edge is detected in the depth-first search, recursively run [Find changes] for the target of that edge. Merge all detected paths with the current path and record it in *reaching\_paths*.

**A3.** [Process Changes.] For each path  $p$  in *reaching\_paths* where  $s$  is *significant\_vertex(last(p))*:

**A3.1.** If  $s$  is a constrains-relation, flatten the requirements of the constraining concept. Record the tuple  $(p, \{\textit{paths from the flattened concept to requirements}\})$  in  $R$ .

**A3.2.** If  $s$  is a refines-relation, flatten the requirements of the refined concept. Record the tuple  $(p, \{\textit{paths from the flattened concept to requirements}\})$  in  $R$ .

**A3.3.** If  $s$  is a requires-relation, record the tuple  $(p, \{\textit{one-vertex path of last(p)}\})$  in  $R$ .

■

In the example from section 4, every reaching change identified in step A2 is an added or deleted constrains-relation. Consequently, in step A3, only A3.1 applies.

The second algorithm, *Concept Compatibility*, is an extension of the first algorithm, *Constraints Change*. For every concept for which compatibility is tested, temporary type-parameter vertices are created. Specifically, a type-parameter vertex and a corresponding constrains-relation edge are created for every parameter of the tested concept that is a target of a change-propagating edge. The *Constraints Change* algorithm is then performed on each of the temporary type-parameter vertices. To decide whether compatibility holds, finally, the requirements that hold for a type parameter in the new and the old versions of the library need

to be compared. In the current prototype, we simply check whether every requirement removed for a type parameter in one way was then added in another way (backward-compatibility) and whether every added requirement existed previously (forward-compatibility). Such simple comparison could find false positives, for example, if a deleted requirement continues to be associated with a type parameter through another, unchanged path in the graph. False positives can be eliminated in a third, forward pass that checks for any added or deleted requirement whether other paths exist that neutralize the effect of addition or deletion, respectively. We have not yet implemented this pass, but the algorithm *Constraints Change* is prepared insofar it already associates changes and paths.

## 6. Case Study

For a case study, we apply CCIA to a proposed change of the iterator concept taxonomy of STL. The background of this proposal is a discussion that started in 2001, when Siek pointed out that the currently standardized iterator concepts entangle the concerns of range traversal and data access [18]. Because of such unnecessarily strong coupling between value-access and traversal requirements, some generic algorithms are under-generalized and some iterator types incorrectly categorized with respect to their traversal protocol (see, e.g., Sutter's paper [24]). Since 2001, altogether 5 new concept specifications for iterators have been submitted for consideration by the C++ standardization committee, and the discussion still continues.

Since iterators are in the very core of STL, any new specification will impact virtually every user of STL and many other libraries. Since these changes are subtle and their impact, because of non-local effects, non-trivial to detect, the proposed changes make for a good case study of a change impact analysis. In this study, we concentrate on the two kinds of intended impact that the proposal sets out to make, namely to reduce conceptual requirements on the parameters of STL algorithms, while at the same time ensuring backward- and forward-compatibility between old and new iterator concept taxonomies. In the following, we denote by *new* the latest version of the proposed concepts [19] and by *old* their current specification from the C++ standard [14].

### 6.1. Iterator Concept Taxonomies

In the old iterator taxonomy, each concept includes requirements related to range traversal as well as value access. In the new taxonomy, in contrast, these requirements are divided into two groups: traversal concepts on the one hand and value-access concepts on the other hand. New and old iterator taxonomies are depicted in Figure 5 (the ITERATOR suffix has been omitted in all concept names); concept

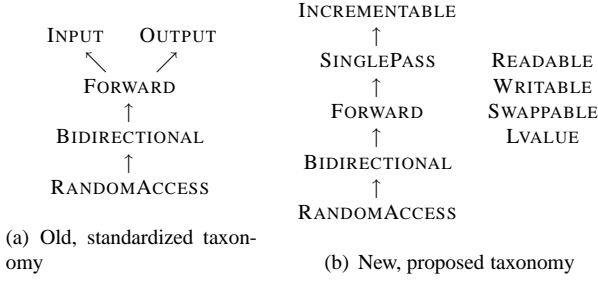


Figure 5. Old and new iterator taxonomies

refinement is represented by the usual arrows. As the figure shows, the old taxonomy consists of five concepts total, while the new taxonomy has nine concepts: five traversal concepts, corresponding to the traversal requirements of the old taxonomy, and, separately, four value-access concepts that have been factored out from the old concepts.

The new concepts were specified with the intention of avoiding compatibility problems. In illustration, Figure 6 shows the old `INPUTITERATOR` concept, which is in the new proposal refactored into two concepts, `SINGLEPASSITERATOR` and `READABLEITERATOR`. As CCIA will reveal, however, the new and old `INPUTITERATOR` concept are, in fact, not fully compatible. The proposal also aims at increasing the genericity of a number of STL algorithms and, to that end, provides rules for rewriting the current constraints on type parameters by the appropriate constraints through the new concepts. For instance, one of the rules specifies that for the STL algorithms `equal` and `mismatch`, all parameters constrained by (the second occurrence of) the concept `INPUT` are now to be constrained by the concepts `INCREMENTABLE` and `READABLE`. CCIA will confirm that (under additional conditions, see Section 6.4) the application of this rewrite rule will dispose of 4 requirements, thus makes both algorithms more generic.

## 6.2. Experiment Setup

Before the analysis can be conducted, the conceptual specifications have to be encoded in a form from which the dependence graph (see Section 5.1) can be constructed. Unfortunately, this encoding cannot be automated. As explained, the iterator concepts are specified informally, using natural language, with parts of the requirements formatted as tables (see, e.g., Figure 1). Further syntactic and semantic requirements are dispersed throughout the documentation—manual encoding is thus unavoidable.

The conditional specifications that are often used in valid expression tables represent an additional complication; they have to be encoded as two different requirements in two different concepts, each corresponding to one branch of the

Old concept	Corresponding new concepts
INPUT	READABLE, SINGLEPASS
OUTPUT	WRITABLE, INCREMENTABLE
FORWARD	READABLE, READABLELVALUE, FORWARD
MUTABLEFORWARD	READABLE, READABLELVALUE, FORWARD, BASICWRITABLE, WRITABLELVALUE
BIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL
MUTABLEBIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL, BASICWRITABLE, WRITABLELVALUE
RANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS
MUTABLERANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS, WRITABLERANDOMACCESS, WRITABLE, WRITABLELVALUE

Table 1. Correspondences between the old and the new iterator concepts

condition [10, App. A]. For example, the return type of the dereference operator “\*a” in `FORWARDITERATOR` is stated as “T if X is mutable, otherwise const T&” (where T denotes the value type, X the type of the modeling iterator) [14, Table 75]. To represent this requirement we had to introduce the `MUTABLEFORWARDITERATOR` concept corresponding to the “is mutable” condition. Other conditional specifications in disguise have the form of optional (type) qualification or different return types of overloaded expressions. We forgo the further discussion of the details of the encoding process as they are not crucial to the case study. Table 1 shows the intended compatibility between old and new concepts including the concepts added during the encoding process. Using this table, compatibility can be decided row-wise: a concept in the old taxonomy is forward-compatible if any type modeling the concept also models the new concepts in the same row. Conversely, a concept of the new taxonomy is backward-compatible if every modeling type also models the old concept in the same row.

The setup of the case study is now easy to explain. To check the compatibility between the old and the new concepts, we proceed essentially as already illustrated in Section 5.3: based on the expected compatibilities defined in Table 1, we redefine all old concepts in terms of their counterparts in the new proposal, that is, we mark all requires- and refines-relations from the old specification as *deleted* and then *add* refines-relations from every old concept to the corresponding new one(s). From there, we build the conceptual dependence graph and then run algorithm *Concept Compatibility*. To calculate the changes in the requirements of STL algorithms, we encode the rewrite rules from the iterator proposal (see Section 6.1 for an example) and apply *Constraints Change* to the type parameters of the STL algorithms. The following two subsections discuss the results of the two CCIA algorithms. An example of the command-line output of CCIA can be found in Figure 7.

INPUTITERATOR	
operation	type
X u(a);	X
u = a;	X&
a == b	convertible to bool
a != b	convertible to bool
*a	convertible to T
a→m	
++r	X&
(void)r++	
*r++	convertible to T

SINGLEPASSITERATOR		READABLEITERATOR	
operation	type	operation	type
++r	X&	X u(a);	X
r++	X	u = a;	X&
a == b	convertible to bool	*a	convertible to T
a != b	convertible to bool	a→m	

$T$  denotes the value type,  $X$  the type of the modeling iterator.

**Figure 6. “Valid Expression” tables of the INPUTITERATOR concept [14, Table 73] and its (incompatible) refactoring into the concepts SINGLEPASSITERATOR and READABLEITERATOR of the new taxonomy [19]**

Full:

```
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines --> (DELETED)
  T (of CopyConstructible) --> requires --> T (of T t; T(t));
  T (of CopyConstructible) --> requires --> T (of const T u; T(u));
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines --> (ADDED)
  Iter (of ReadableIterator) --> requires --> T (of typename T::value_type;)
  Iter (of ReadableIterator) --> requires --> T (of T::value_type v; T p; v = *p;)
```

Compatibility-summary:

```
InputIteratorModel NOT COMPATIBLE
ForwardIteratorModel NOT COMPATIBLE
```

Compatibility-incompatible:

```
InputIteratorModel
  Requirement "T t; T q = t++;" added 1 times, deleted 0 times. (FORWARD INCOMPATIBLE)
  Requirement "typename T::difference_type;" added 0 times, deleted 1 times. (BACKWARD INCOMPATIBLE)
```

Compatibility-short:

```
OutputIteratorModelIter
  Requirement "T t; T u; T& q = (t = u);" added 1 times, deleted 1 times.
  Requirement "T t; const T v; T& q = (t = v);" added 1 times, deleted 1 times.
```

Genericity-change:

```
find_first_of::ForwardIterator2 --- Genericity not increased.
```

**Figure 7. Examples of five different kinds of output from the analysis**

### 6.3. Compatibility

Surprisingly, the analysis yields that new and old iterators are not compatible. More specifically, none of the 8 old concepts and their corresponding new concepts (in the sense of Table 1) is backward- or forward-compatible. Even if we ignore incompatibilities propagated through the refinement hierarchy, there are only 3 concepts that introduce no incompatibilities on their own: FORWARDITERATOR and BIDIRECTIONALITERATOR (yet, see the discussion below), and the MUTABLEBIDIRECTIONALITERATOR concept that we had to introduce (Section 6.2). These 3 concepts will be automatically both backward- and forward-compatible provided their refined concepts have been made compatible.

Table 2 details the incompatibilities. Following the refinement hierarchy, the table lists for each concept exactly the incompatibilities this concept introduces, that is, omits those incompatibilities that are only propagated through refinement. Each row, thus, corresponds to one incompatibility; the kind of incompatibility is indicated in the last

column. For instance, line 1 of the table indicates that the associated type “value\_type” of the old specification of OUTPUTITERATOR is missing in the specification of the corresponding new concepts (WRITABLEITERATOR and INCREMENTABLEITERATOR), which breaks backward-compatibility. A further 6 incompatibilities of the OUTPUTITERATOR concept are given in lines 2-7. They all propagate to all refining concepts—that is, all other concepts except the INPUTITERATOR concept—but are not listed again in the table.

It is important to note that some incompatibilities detected by our analysis are in fact wrong, albeit in a subtle way that shows a general limitation of our approach. We indicate all faults on part of the analysis by stricken text in the last column of Table 2. In all cases, the reason of the false positive is that the analysis does not take the semantics of the requirements into account. As a purely syntactical analysis, it cannot recognize mere splitting or merging of C++ expressions. For example, the requirement “\*r++” of the FORWARDITERATOR concept is decomposed into two requirements, “r++” and “\*r”, which are then associated to

	Con.	Requirement	
1	O	typename lter::value_type;	b
2	O	lter r; lter q = r++;	f
3	O	typename lter::difference_type;	b
4	O	typename lter::pointer;	b
5	O	typename lter::reference;	b
6	O	lter r; const lter& q = r++;	b
7	O	lter r; V o; *r++ = o;	<del>b</del>
8	I	lter r; lter q = r++;	f
9	I	typename lter::difference_type;	b
10	I	typename lter::pointer;	b
11	I	typename lter::reference;	b
12	I	lter r; r++;	<del>b</del>
13	I	lter r; lter ::value_type q = *r++;	<del>b</del>
14	F	lter r; const lter ::value_type& q = *r++;	<del>b</del>
15	MF	lter r; lter ::value_type& o = *r;	f
16	MF	lter r; lter ::value_type o; *r++ = o;	<del>b</del>
17	B	lter r; lter ::value_type q = *r--;	<del>b</del>
18	RA	lter r; lter ::value_type q = r[n];	f
19	RA	lter r; const lter ::value_type& q = r[n];	b
20	MRA	lter r; lter ::value_type v; r[n] = v;	f

where O=OUTPUT, I=INPUT, F=FORWARD, MF=MUTABLEFORWARD,  
B=BIDIRECTIONAL, RA=RANDOMACCESS, MRA=MUTABLERANDOMACCESS

**Table 2. The requirements that cause forward-incompatibility (f) or backward-incompatibility (b). False positives are indicated by striken text.**

algorithms	Del.
reverse_copy, find_end, adjacent_find, search, search_n, rotate_copy, lower_bound, upper_bound, equal_range, binary_search, min_element, max_element	1
find_first_of	3, 4
copy_backwards	1, 0
equal, mismatch, transform	4

**Table 3. STL algorithms with increased genericity, grouped by the number of requirements removed per parameter (second column); backward-compatibility is provided.**

different new concepts (false positive on line 14). False positives were identified manually after the analysis has completed.

#### 6.4. Constraints Change of Parameters

Every backward-incompatibility from Table 2 introduces an additional requirement on an algorithm parameter, which it originally did not have to meet. Since it is of interest to understand nevertheless how much the genericity *could* increase, we enforced total compatibility for the purpose of the experiment. For every backward- and forward-incompatibility in Table 3 we accordingly added and removed the corresponding requirement in the old concepts.

Table 3 shows for which algorithms their genericity increases provided the intended compatibility between the new and the old iterators holds. The algorithms are grouped by the number of requirements removed for each of their

type parameters. From the 42 STL algorithms that are affected by the changes to the iterator concepts, 17 became more generic.

## 7. Conclusions

Generic libraries in C++ use so-called *concepts* to constrain type parameters in templates; conversely, concepts control the degree of genericity of an algorithm or a class. The task of a generic library designer, therefore, includes the development of a concept taxonomy for the particular application domain of the library. Despite their importance for specification and implementation, however, very little mechanical support exists for concepts. In this paper, we introduce the first conceptual change impact analysis, CCIA. The analysis is a reachability analysis, comprising a forward-reachability stage and a backward-reachability stage. For extensibility and scalability, the two stages are organized in a pipe-and-filter manner.

We applied CCIA to an officially submitted proposal to change iterator concepts, one of the most fundamental constructs in STL and other generic libraries. The analysis shows that, unexpectedly, the two iterator concept taxonomies are neither forward- nor backward-compatible and lists the parts of the specification that cause incompatibility. Its results can help library designers to avoid unintended effects of a change and, in general, provides a base for assessing its impact.

## 8. Future Work

As the two next steps, we want to fully automate our analysis and make it more customizable. To develop CCIA into a full-fledged tool, we plan to integrate it in the ConceptGCC compiler [7, 9], which is an experimental implementation of concepts for C++ [10]. Full integration in a real compiler requires extending our analysis with support for all linguistic features of concepts, like nested requirements, for example.

To make our analysis more customizable, we want to identify basic “queries”, similar to the queries in PathInspector<sup>TM</sup> [6]. These queries should allow constructing new filters by simple combining basic queries appropriately. We expect already the existing filters, *Constraints Change* and *Concept Compatibility*, to be decomposable into smaller, reusable computation units.

## Acknowledgments

We thank the reviewers of ICSM that greatly improved the presentation of the paper. Further thanks go to the participants of the Library-Centric Software Design (LCSD)

workshop at OOPSLA, where an early version of the case study was presented; Frank Tip deserves a special mention. We also thank the authors of the Boost Graph Library (BGL), which saved us substantial implementation work. Douglas Gregor provided input for the necessary mapping from the semi-formal specifications in the C++ standard to a machine-usable format.

## References

- [1] Standard Template Library Programmer's Guide. <http://www.sgi.com/tech/stl/>.
- [2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 384–396. ACM Press, 1993.
- [3] L. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *Proc. 19th International Conference on Software Maintenance (ICSM)*, pages 256–265. IEEE Computer Society, 2003.
- [4] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.
- [5] M. J. Fyson and C. Boldyreff. Using application understanding to support impact analysis. *Software Maintenance: Research and Practice*, 10(2):93–110, 1998.
- [6] GrammaTech. Pathinspector. <http://www.grammatech.com/products/codesurfer>.
- [7] D. Gregor. ConceptGCC: Concept extensions for C++. <http://www.generic-programming.org/software/ConceptGCC/>.
- [8] D. Gregor, J. Järvi, J. Siek, A. Lumsdaine, G. D. Reis, and B. Stroustrup. Concepts: First-class language support for generic programming. In *Proc. 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006.
- [9] D. Gregor and J. Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.
- [10] D. Gregor, J. Siek, J. Willcock, J. Jarvi, R. Garcia, and A. Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.
- [11] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proc. 8th IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, pages 172–182. IEEE Computer Society, 1997.
- [12] J. Heflin and J. A. Hendler. Dynamic ontologies on the web. In *Proc. of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 443–449. AAAI Press / The MIT Press, 2000.
- [13] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proc. 14th International Conference on Software Engineering (ICSE)*, pages 392–411. ACM Press, 1992.
- [14] ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++. *C++ Standard Draft, N1804=05-0064*, ANSI standards for information technology edition, 2003.
- [15] 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, 1992.
- [16] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 202–211. IEEE Computer Society, 1994.
- [17] 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.
- [18] J. Siek. Improved iterator categories and requirements. Technical Report J16/01-0011 = WG21 N1297, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2001.
- [19] J. Siek, D. Abrahams, and T. Witt. New iterator concepts. Technical Report N1640=04-0080, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Apr. 2004.
- [20] J. 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++, Jan. 2005.
- [21] J. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 73–84. ACM Press, 2005.
- [22] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-95-11, Hewlett Packard, Nov. 1995.
- [23] B. Stroustrup and G. Dos Reis. A concept design (rev.1). Technical Report N1782=05-0042(rev.1), ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Apr. 2005.
- [24] H. Sutter. vector(bool) is nonconforming, and forces optimization choice. Technical Report J16/99-0008 = WG21 N1185, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Feb. 1999.
- [25] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Software Eng.*, 29(6):495–509, June 2003.
- [26] 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, 2004.
- [27] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 1–10. ACM Press, 2002.