Complete Traversals and their Implementation Using the Standard Template Library

Eric Gamess(*)1 *David R. Musser*(†) *Arturo J. Sánchez-Ruíz*(*)2

(*)Centro de Ingeniería de Software y Sistemas (ISYS)
Escuela de Computación, Facultad de Ciencias
Universidad Central de Venezuela
CARACAS, VENEZUELA
egamess@kuaimare.ciens.ucv.ve, *arturo@acm.org*

(†)Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180
USA
musser@cs.rpi.edu

Abstract

A *complete traversal* of a container C (such as a set) is informally described by the iteration scheme

$$\textbf{for all } x \in C$$
$$\mathcal{F}(x, C)$$

where $\mathcal{F}$ is a function that might possibly modify C by inserting new elements into it. We assume that the order in which the elements are treated is not relevant, as long as the iteration continues until $\mathcal{F}$ has been applied to all elements currently in C, including those $\mathcal{F}$ has inserted. Standard iteration mechanisms, such as the iterators provided in the C++ Standard Template Library (STL), do not directly support complete traversals. In this paper we present two approaches to complete traversals, both extending the STL framework, one by means of generic algorithms and the other by means of a container adaptor.

Keywords: Generic Programming, Standard Template Library (STL), Iterators, Adaptors, Containers, Templates, C++.

1 Introduction

Consider the following problem:

> A manager wants to arrange a meeting of a certain set of people in her company. For each person in the original set she also wants to invite that person's boss, that boss's boss, and so on. (She has a database from which she can tell who a person's boss is.)

[1]Present address: Computer Science Department, Universidad del Valle, Cali, Colombia.
[2]Present address: University of Massachusetts, CIS Department, 285 Old Westport Road, North Dartmouth, MA 02747, USA.

The manager can solve this problem fairly simply by writing down the initial set of people's names in a list and iterating through the list from beginning to end, inserting new persons at the end where they become part of the iteration. To avoid duplicating names on the list, she should append a person's name to the list if and only if the name is not already present. In the computerized version of this problem, with a large list, an inefficient linear search is required, rather than the binary search that would be possible if the set of names could be kept in, say, alphabetic order. In that case, however, new names should be inserted in their proper place to maintain the order. But this in turn makes it difficult to tell when the iteration should stop, since names might have been inserted before the current iteration point. The kind of iteration required to gracefully solve this problem is called a *complete traversal*; we give a formal definition in Section 2. Problems requiring complete traversals are fairly common (we give another example in Section 2), and while there are various *ad hoc* ways of solving them, programmers should ideally have at their command an efficient packaged solution. In this paper we describe two such approaches to complete traversals, both of which fit into the framework defined by the Standard Template Library, STL (part of the ANSI/ISO standard for C++ [1]).

STL (see also [16, 11, 15]) provides a set of easily configurable software components of six major kinds: generic algorithms, containers, iterators, function objects, adaptors, and allocators. In each of these component categories, STL provides a relatively small set of fundamental components; it is through uniformity of interfaces and orthogonality of component structure that STL provides functionality far beyond the actual number of components included. But STL is not intended as a closed system; its structure is designed with extension in mind. The complete traversal components described in this paper may be of interest not only for the functionality they provide, but also as examples of, and measures of, how well the existing STL components support extensions.

In Section 3, we give two distinct ways of solving the complete traversal problem: a generic algorithms approach and a container adaptor approach. In both approaches, the complete traversal components are designed to work with the category of STL components called *associative containers*, which support fast retrieval of objects based on keys. The generic algorithms are restricted to *sorted* associative containers, in which keys are maintained according to a given ordering function, but the container adaptor we provide can also be used with *hashed* associative containers, which give up order properties in favor of faster retrieval. Hashed associative containers are not part of the C++ standard but are now provided as an STL extension by at least one compiler vendor [15]. Another classification of associative containers is *unique*, in which objects in a container cannot have equivalent keys, versus *multiple*, in which they can. Still another classification is *simple* containers, in which only the keys are stored, versus *pair* containers, in which pairs of keys and associated values are kept. The sorted associative containers provided in STL are shown in the following table:

Component	Classification
`set<Key, Compare, Allocator>`	unique, simple
`multiset<Key, Compare, Allocator>`	multiple, simple
`map<Key, T, Compare, Allocator>`	unique, pair
`multimap<Key, T, Compare, Allocator>`	multiple, pair

All of the associative containers have essentially the same interface; e.g., each provides `insert` and `erase` member functions for inserting and deleting objects, several kinds of search member functions, and several kinds of iterators for traversing through the current contents. None, however, provides for complete traversals in the sense discussed here. The specifics of these interfaces, and how our components are used for complete traversals, are illustrated at the end of Section 3, in terms of solving the manager's problem stated at the beginning of the paper.

We give more than one approach to the complete traversal problem because no single solution seems best in all cases. The presentation in Section 3 includes complexity analyses and discussion of other factors such as naturalness of interfaces. We are exploring still other approaches, which are discussed briefly in the concluding section.

2 Complete Traversals

In this section we give a precise definition of complete traversals and develop sufficient conditions for termination and uniqueness of complete traversals. We also describe another example application of the concept.

Let C be a container, $x \in C$, and $\mathcal{F}$ be a (program) function. Denote by $\hat{\mathcal{F}}(x, C)$ the container of elements to be inserted into C by the call $\mathcal{F}(x, C)$. We define complete traversals in terms of a rewriting relation, as follows:

Definition 2.1

1. *Given any containers C and D such that $D \subseteq C$ and a (program) function $\mathcal{F}$, let $x \in C - D$, $C' = C \cup \hat{\mathcal{F}}(x, C)$, and $D' = D \cup \{x\}$. We say that (C', D') is a* traversal successor *of (C, D) and denote this relation by $(C, D) \rightarrow (C', D')$.*
2. *(C, D) is said to be a* normal form, *or* irreducible, *if $C = D$.*
3. *A* traversal sequence *for a container C using a function $\mathcal{F}$ is any sequence $(C_0, D_0) \rightarrow (C_1, D_1) \rightarrow \ldots \rightarrow (C_n, D_n)$ starting from $C_0 = C$ and $D_0 = \phi$.*
4. *Such a traversal sequence is said to be* terminating *if (C_n, D_n) is irreducible (equivalently, if $C_n = D_n$).*
5. *A* complete traversal *of a container C using $\mathcal{F}$ is any terminating traversal sequence for C using $\mathcal{F}$.*

The container operations are appropriately interpreted depending on whether C is a unique or multiple container.

Can two complete traversals of the same container C and function $\mathcal{F}$ result in different final containers? Yes, as shown by the following example:

$$\begin{array}{rcl} C & = & \{1, 4\} \\ \mathcal{F}(x, X) & : & \textit{insert } (x + |X|) \textbf{ div } 2 \textit{ into } X \end{array} \tag{1}$$

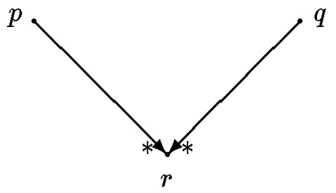
where C is assumed to be a unique container and $|X|$ denotes the size of X. In Fig. 1 we show all possible results depending on the order used to apply $\mathcal{F}$. We see that, for this C and $\mathcal{F}$, different orders lead to different results.

Order	Result
$\langle 1,4,3\rangle$	$\{1,3,4\}$
$\langle 4,1,2,3\rangle$	$\{1,2,3,4\}$
$\langle 4,1,3,2\rangle$	$\{1,2,3,4\}$
$\langle 4,3,1,2\rangle$	$\{1,2,3,4\}$

Figure 1: All possible results associated with instance in (1)

Since we have cast the traversal successor relation as a rewriting relation, we can draw upon standard rewriting theory (e.g., [7, 12]) for deriving conditions for uniqueness of complete traversal results. Given an arbitrary rewriting relation $\rightarrow$, the reflexive, transitive closure is denoted by $\rightarrow^*$ and is referred to as *reduction*.

Elements p and q are *joinable* if and only if there exists an r such that $p \rightarrow^* r$ and $q \rightarrow^* r$.



Joinability Relation

A rewriting relation $\rightarrow$ is said to be *uniformly terminating*,[3] if and only if there is no infinite sequence of the form $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$. Thus with a uniformly terminating relation every reduction path is finite and every element has a *normal form*; i.e., the last element of a path. An element might have several distinct normal forms, however.

A rewriting relation $\rightarrow$ is *confluent* if and only if for all elements p, q_1, q_2, if $p \rightarrow^* q_1$ and $p \rightarrow^* q_2$ then q_1 and q_2 are joinable.

It follows directly from the definitions that if a rewriting relation is confluent, an element cannot have two distinct normal forms; that is, confluence is a sufficient condition for every element to have a unique normal form.

It is possible in some cases to use a more localized test. A rewriting relation $\rightarrow$ is *locally confluent* if and only if for all elements p, q_1, q_2, if $p \rightarrow q_1$ and $p \rightarrow q_2$ then q_1 and q_2 are joinable. (Note the difference from confluence: only one step is taken from p rather than arbitrarily many.)

If a rewriting relation is uniformly terminating and locally confluent, then it is confluent. This is called the "Diamond Lemma," because of the structure of its proof; see [7].

Applying this theory, we have:

Theorem 2.2 *If a traversal successor relation is uniformly terminating and locally confluent, then every complete traversal of a container C using a function $\mathcal{F}$ results in a unique final container. We say that the complete traversal computation is* determinate.

[3] Other terms for this property are *finitely terminating* and *Noetherian*.

Thus, in the determinate case, the order in which elements are chosen is not relevant, and we can speak of *the* complete traversal $CT(C, \mathcal{F})$ of C by $\mathcal{F}$. Uniform termination alone does not guarantee determinacy, as the instance in (1) shows.

Theorem 2.3 *If $\hat{\mathcal{F}}(x, C)$ does not depend on C, then the traversal successor relation is locally confluent.*

Proof: At any reduction step i, if $x_i, x'_i \in C_i - D_i$ and $x_i \neq x'_i$, let

$$\begin{aligned}(C_{i+1}, D_{i+1}) &= (C_i \cup \hat{\mathcal{F}}(x_i, C_i), D_i \cup \{x_i\}) \\ (C'_{i+1}, D'_{i+1}) &= (C_i \cup \hat{\mathcal{F}}(x'_i, C_i), D_i \cup \{x'_i\})\end{aligned}$$

Then (C_{i+1}, D_{i+1}) and (C'_{i+1}, D'_{i+1}) are immediately joinable to a common result: continue one step from (C_{i+1}, D_{i+1}) using x'_i to obtain

$$(C_{i+1} \cup \hat{\mathcal{F}}(x'_i, C_{i+1}), D_i \cup \{x_i\} \cup \{x'_i\})$$

and one step from (C'_{i+1}, D'_{i+1}) using x_i to obtain

$$(C'_{i+1} \cup \hat{\mathcal{F}}(x_i, C'_{i+1}), D_i \cup \{x'_i\} \cup \{x_i\})$$

By the assumption that $\hat{\mathcal{F}}$ does not depend on the container, these two results are identical. □

Corollary 2.4 *If $\hat{\mathcal{F}}(x, C)$ does not depend on C and the traversal successor relation is uniformly terminating, it is also determinate.*

Proof: Follows directly from Theorems 2.2 and 2.3. □

We now consider some sufficient conditions for uniform termination.

Theorem 2.5 *Let C and $\mathcal{F}$ be such that for any reduction sequence $(C, \phi) \to (C_1, D_1) \to \ldots$ there is some i such that $(C, \phi) \to^* (C, D_i)$ and for all $j > i$ in the sequence,*

1. *$\hat{\mathcal{F}}(x_j, C_j) \subseteq C_j$, if C is unique; or,*
2. *$\hat{\mathcal{F}}(x_j, C_j) = \phi$, if C is multiple.*

Then the traversal successor relation is uniformly terminating.

Proof: In both cases, we have $C_j = C_i$ for all $j \geq i$. Since $D_j \subseteq C_j$ we have $|D_j| \leq |C_i|$. Since $|D_j|$ is strictly increasing, eventually we must have $D_j = C_j$. □

Corollary 2.6 *If C is a unique container and X is a finite set such that $C \subseteq X$ and range$(\mathcal{F}) \subseteq X$, then the traversal successor relation is uniformly terminating.*

Proof: The length of any reduction sequence is bounded by $|X|$. $\square$

As described in Section 1, the manager's invitation list is one problem that can be solved by instantiating a complete traversal scheme. We show in detail such a solution using our complete traversal components in Section 3.4. We can easily prove uniform termination and determinacy for this example using the theory developed above. To prove uniform termination, we can apply Corollary 2.6 letting X be the set of all employees. Determinacy then follows from Corollary 2.4, since we have

$$\hat{\mathcal{F}}(x, C) = \text{the boss of } x$$

which is independent of C.

As another example, the one which inspired this study of complete traversals, suppose we are given a specification of types written in a certain formalism called CTS.[4] CTS types are classified into *atomic types* (which have no structure), *concrete types* (which are used to define data structures), and *abstract types* (which are used to define classes). The representations associated with classes are, in turn, concrete CTS types. With each CTS expression we can associate a *syntax graph*, which captures the relationships among involved types. We also assume that each CTS expression defining a type is associated with a name. Given a map C of elements (n, g), where n is the name of an abstract CTS type and g is the syntax graph associated with its representation, we want to iterate over C in such a way that, at each iteration, we take an element (n, g) and traverse g in a certain order, making sure that we insert into C elements (n', g'), for all type names n' we come across, where g' is the syntax graph associated with n'. Iteration stops when all elements in C have been processed. This instance of the complete traversal problem is summarized in Fig. 2, where C is shown with its initial value. Notice that, since the conditions of Corollary 2.4, and Corollary 2.6 hold in this case, this complete traversal problem is also uniformly terminating and determinate.

$$C = \{(n, g) \mid n \textit{ is the name of an abstract } \text{CTS} \textit{ type}$$
$$\textit{and } g \textit{ is the syntax graph of its representation}\}$$

$$\mathcal{F}(x, C) = \textit{traverse } x.g \textit{ and insert in } C \textit{ any type name found}$$
$$\textit{with its corresponding syntax graph}$$

Figure 2: An instance of the complete traversal problem

3 Complete Traversals Implemented as STL Extensions

In this section we present two different approaches to complete traversals, one using generic algorithms and the other a container adaptor. In both cases, we assume the complete traversal problem to be defined by a function such that makes the traversal successor relation both uniformly terminating and confluent. We also compare the complexity of these components, and show how they can be used in a simple application.

[4] CTS stands for Common Type System. The application problem and the definition of CTS were taken from [14].

3.1 Generic Algorithms

A generic algorithm is an algorithm designed to work with a variety of data structures, the specialization to a particular structure being realized by the programming language processor (compiler, interpreter, or run-time system) rather than by manual editing of the source text. In the STL framework, generic algorithms are expressed as C++ function templates. An algorithm can be made generic over a category of containers if the way it accesses a container can be limited to a fixed set of operations, all of which are provided, with the same interfaces, by every container in the category. As a simple example, consider the following function template for performing an (ordinary) iteration over the elements of a container, applying a function `f` to each element.

```
template <class Container, class Function>
void for_each(Container& container, Function f) {
  typename Container::iterator i;
  for (i = container.begin(); i != container.end(); ++i)
    f(*i);
}
```

This function can be applied to any of the STL containers, because they all provide iterator types (`Container::iterator`) and member functions `begin` and `end` that return iterators defining the range of positions of elements currently within the container. To use `for_each` with a list of integers, for example, one could write

```
  list<int> list1;
  // ... code to insert some elements in list1
  for_each(list1, f);
```

where `f` is some function object that does not modify the list.

The main algorithmic idea behind our first approach is to set up an iteration through the container with the ordinary iterators provided, applying a function `f` that may generate new elements for insertion into the container. But instead of allowing `f` to do the insertions, we require it to enter the new elements in a queue. After each call of `f`, we take elements from the queue and insert them into the container, checking whether they are inserted before or after the current iteration position. If an element is inserted after the current position, it will be taken care of in the course of the remaining iterations, but if it is inserted before the current position, we apply `f` to it immediately (which in turn may generate new elements and add them to the queue).

STL defines several different categories of iterators according to the set of operations they support, the most powerful being random access iterators. While random access iterators support relational operators, the bidirectional iterators provided by associative containers do not. However, we can still easily tell whether an element x appears before another element y in the iteration sequence of a sorted associative container, just by comparing x with y using the order relation $\prec$ the container uses to maintain sorted order.

There are then two distinct cases, depending on whether the container is unique or multiple. A unique container does not allow two equivalent elements to be in the container. Equivalence of elements is defined as x is equivalent to y if both $x \prec y$ and $y \prec x$ are false. With a unique container, if `f` generates an element equivalent to one already present (either before or after the insertion point) we do not apply `f` to it again, but with a multiple container we do.

For the unique container case, we define the generic algorithm `complete_unique_traversal`, which traverses a unique sorted associative container applying a function object `f` to each element. It is assumed that `f` creates and maintains a queue of elements to be inserted (but does not insert them itself), and that it makes this queue available as a public member named `Q`. When inserting an element `v` taken from this queue into the container, we use an insertion function that returns a pair `p` consisting of an iterator that tells where the element was found or inserted, and a boolean value that is true if and only if the element was actually inserted (was not already present). The full definition of `complete_unique_traversal` is shown in Fig. 3.[5]

```
template <class UniqueSortedAssociativeContainer, class Function>
void complete_unique_traversal(UniqueSortedAssociativeContainer& container,
                               Function f)
{ typename UniqueSortedAssociativeContainer::value_type v;
  typename UniqueSortedAssociativeContainer::iterator i;
  typedef typename UniqueSortedAssociativeContainer::iterator iterator;
  for (i = container.begin(); i != container.end(); ++i) {

    f(*i, container);
    while (!f.Q.empty()) {

      v = f.Q.front();
      f.Q.pop();
      pair<iterator, bool> p = container.insert(v);

      if (p.second && container.value_comp()(v, *i))
        // v has been inserted in container (it wasn't already there)
        // and it occurs before the current traversal
        // position, i, so process it now with f:
        f(v, container);

    }//end while ...

  }//end for ...

}//end complete_unique_traversal
```

Figure 3: Generic algorithm for unique containers

For the multiple container case, the generic algorithm `complete_multiple_traversal` is defined similarly, as shown in Fig 4, but the implementation is complicated by the fact that when an element taken from the queue is equivalent to the one at the current position, testing for whether it is inserted before or after that position is no longer simply a matter of comparing it with the current element. The C++ standard specification of the `insert` operation on multiple containers leaves unspecified where within a range of equivalent elements a new equivalent element is inserted, so we must conduct a linear search within the range.

[5] All the C++ code presented in this paper, plus additional examples, is available from `http://www.cs.rpi.edu/gpg`.

```
template <class MultipleSortedAssociativeContainer, class Function>
void
  complete_multiple_traversal(MultipleSortedAssociativeContainer& container,
                              Function f)

{
  typename MultipleSortedAssociativeContainer::value_type v;
  typename MultipleSortedAssociativeContainer::iterator i, j, k;

  for (i = container.begin(); i != container.end(); ++i) {

    f(*i, container);
    while (!f.Q.empty()) {

      v = f.Q.front();
      f.Q.pop();
      j = container.insert(v);
      // If v's position, j, in container is before the current
      // traversal position, i, then process v now with f

      if (container.value_comp()(v, *i))
        f(v, container);

      else if (!container.value_comp()(*i, v)) {
        // *i and v are equivalent, so we must search forward from
        // v's position, j, through all of the following equivalent
        // elements, if any, looking for position i:

        for (k = ++j; k != container.end() &&
               !container.value_comp()(v, *k); ++k)
          if (k == i)  {         // v was inserted before position i,
            f(v, container);    // so we process v now with f
            break;
          } //end search

      }//end equivalent case

    }//end while

  }//end for

}//complete_multiple_traversal
```

Figure 4: Generic algorithm for multiple containers

With some existing implementations of STL, we could omit the linear search since an element is always inserted at the end of the range of elements equivalent to it. We cannot assume this to be the case with all implementations since the standard does not require it. This situation is a simple illustration of the tension faced by the library (or language) specifier, between the goal of allowing implementors as much freedom as possible by leaving some details unspecified, and the goal of enabling programmers to optimize their code while retaining portability.

3.2 A Container Adaptor

The implementation based on generic algorithms, shown in Section 3.1, requires function `f` (implemented by the programmer) to put the elements generated in each activation in a queue, which is less natural than having it insert the elements directly into the container. In order to relax this requirement, we propose another approach based on a container adaptor, whose usage for implementing complete traversals is depicted in Fig. 5. The `complete_container` adaptor provides (see

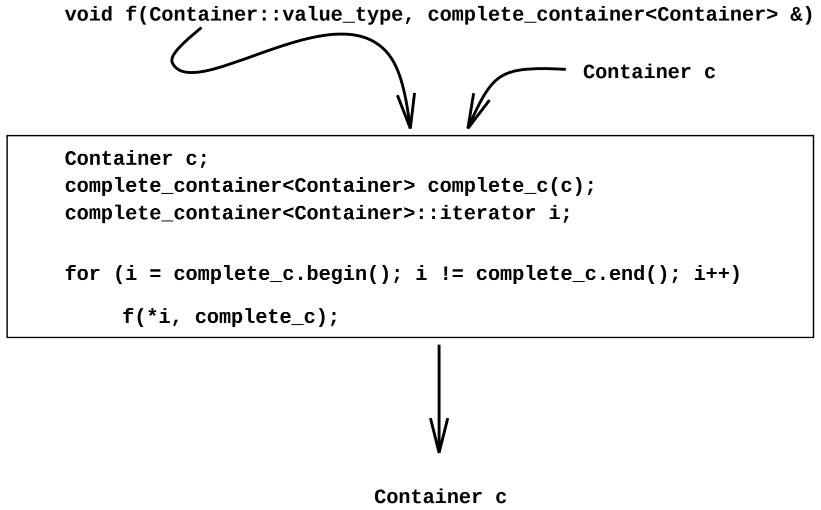


Figure 5: Implementing complete traversals by using a container adaptor approach

Fig. 6):

- Types `size_type` and `value_type` taken from the corresponding `Container`.
- A constructor that takes a `Container` as a parameter and stores a reference to its argument, and also creates an iteration list (see below).
- Types `iterator` and `const_iterator`, which implement complete traversal iterators on non-constant and constant containers, respectively.
- Member function `size`, which returns the size of the `Container` component of a `complete_container`.
- Member function `insert` which takes a `value_type` value and inserts it into the underlying `Container`.

```
#ifndef COMPLETE_CONTAINER_H
#define COMPLETE_CONTAINER_H

#include <list>
using namespace std;

template <class SortedAssociativeContainer>
class complete_container {

public:
   typedef typename SortedAssociativeContainer::size_type size_type;
   typedef typename SortedAssociativeContainer::value_type value_type;
   //iterators are simply list-based iterators
   typedef list<value_type>::iterator iterator;
   typedef list<value_type>::const_iterator const_iterator;

protected: //representation
   list<value_type> iteration_list;       //iteration list
   SortedAssociativeContainer& container; //reference to container

public:
   complete_container(SortedAssociativeContainer& c) : container(c) { //constructor
     copy(c.begin(), c.end(), back_inserter(iteration_list));
   }

   //to manipulate the associated container:
   SortedAssociativeContainer& base() { return container; }

   //iterator-related member functions:
   iterator begin() { return iteration_list.begin(); }
   const_iterator begin() const { return iteration_list.begin(); }
   iterator end() { return iteration_list.end(); }
   const_iterator end() const { return iteration_list.end(); }

   size_type size() const { return container.size(); } //size

   void insert(const value_type& v) { //direct insertion
     typename SortedAssociativeContainer::size_type n = container.size();
     container.insert(v);
     if (container.size() != n) //this allows uniform handling of both
                                //unique and multiple containers
       iteration_list.push_back(v);
   }
};

#endif
```

Figure 6: Implementation via a container adaptor

The representation of a `complete_container` consists of a container reference and an iteration list, implemented as an STL `list<value_type>`. The implementation maintains the following invariant:

> *At the beginning of each iteration with an* `iterator`*, the element associated with the iterator on the iteration list is the one to be processed, the elements to its right are those that will be processed next, and elements to its left are those which were already processed.*

This invariant implies that, initially, the iteration list should be a copy of the original `Container`, and insertion of an element with the adaptor's `insert` member should insert the element into the underlying container and onto the end the iteration list.

3.3 Complexity

To compare the time complexity of our two approaches, let C be a sorted associative container, let n be the number of elements in C initially and let m be the total number of insertion attempts done by the complete traversal of C using a function f. There might be fewer than $N = n + m$ elements in C after the traversal because some of the elements being inserted might have already been present. But N is a bound on the final size of C, so $O(\log N)$ bounds the time for any one insertion, and $O(m \log N)$ bounds the time for all insertions. Let $T(f, j, k)$ be a bound on the total amount of time for j evaluations of f on a container of maximum size k, where we exclude (because we have already counted it) any time f spends doing insertions. So the total time for evaluating f is $T(f, N, N)$.

1. In `complete_unique_traversal`, the time for all of the queue processing is $O(m)$, so the total time is

$$\begin{array}{lccccc} & \text{queue processing time} & + & \text{insertion time} & + & \text{function evaluation time} \\ = & O(m) & + & O(m \log N) & + & T(f, N, N) \end{array}$$

   The extra linear searches required by `complete_multiple_traversal` add to these times an extra $O(mN)$ term in the worst case, but in practice the extra time is likely to be $O(N)$.

2. For the `complete_container` adaptor, the time for all of the list processing is $O(N)$, so the total time is

$$\begin{array}{lccccc} & \text{list processing time} & + & \text{insertion time} & + & \text{function evaluation time} \\ = & O(N) & + & O(m \log N) & + & T(f, N, N) \end{array}$$

Since $T(f, N, N)$ is $\Omega(N)$, the bound in 1 cannot be asymptotically better than the bound in 2. It is clear, however, that the list processing time associated with complete containers is more than the queue processing associated with the complete traversal algorithms. It is also clear that the complete traversal algorithms require less extra space than the complete containers. On the other hand, complete containers offer a more natural interface and can be used with hashed associative containers, while the same cannot be said of our complete traversal algorithms. In summary, these two approaches offer a good spectrum of possibilities to tackle the complete traversal of containers.

3.4 An Example

As a simple example of use of the generic components described in this section, we program the solution of the manager's invitation list problem described in Section 1. First we present a solution using one of the generic algorithms of Section 3.1.

```
// Read a bosses database file, another file containing an initial
// set of persons, and compute a complete traversal of the set, inserting
// as a new member the boss of anyone already present.
#include <iostream>
// ... other #includes, for STL and string class headers
using namespace std;
#include "complete_traversal.h"  // contains complete_unique_traversal algorithm

// A type of map from strings to strings, alphabetically ordered:
typedef map<string, string> name_association;

// A type of set of strings, ordered by alphabetic ordering of the keys:
typedef set<string> name_set;

// A class of function objects for generating names using a name_association
// directory, meeting requirements of the complete_unique_traversal algorithm:
class name_function {
private:
  const name_association& directory;
public:
  name_function(const name_association& d) : directory(d) { }
  queue<string, list<string> > Q;
  void operator()(const string& name, name_set& s) {
    cout << name << endl;
    typename name_association::const_iterator i = directory.find(name);
    if (i != directory.end())
      Q.push((*i).second);
  }
};

// Function to scan the database file and build an internal directory
void get_database(istream& is, name_association& directory); // details omitted

// Function to scan the names file and build a names set
void get_names(istream& is, name_set& names); // details omitted

int main()
{
  // Create the bosses database:
  name_association bosses;
  ifstream ifs("bosses.txt");
  get_database(ifs, bosses);

  // Create the initial set of names:
  name_set invitees;
  ifstream ifs1("initial.txt");
  get_names(ifs1, invitees);
```

```
  cout << "Original set of invitees:" << endl;

  name_set::iterator i;

  for (i = invitees.begin(); i != invitees.end(); ++i)
    cout << *i << "  ";
  cout << endl << endl;

  cout << "Output during complete traversal:" << endl;
  name_function get_boss(bosses);
  complete_unique_traversal(invitees, get_boss);
  cout << endl << endl;

  cout << "Final set of invitees:" << endl;
  for (i = invitees.begin(); i != invitees.end(); ++i)
    cout << *i << "  ";
  cout << endl << endl;
  return 0;
}
```

Finally, a solution using the `complete_container` adaptor of Section 3.2 needs a different definition of the function to be applied to each person, one that actually does the insertions. The relevant part of the code follows.

```
template <class AssociativeContainer>
class name_function { //Class to which the traversing object function belongs
private:
  const name_association& directory;
public:
  name_function(const name_association& d) : directory(d) { }

  void operator()(const string& name, AssociativeContainer& c) {
    cout << name << "  ";
    name_association::const_iterator i = directory.find(name);
    if (i != directory.end() && (*i).second != string("---"))
      c.insert((*i).second);
  }
};

  typedef complete_container<name_set> cc_type;
  cc_type cc(invitees);
  name_function<cc_type> insert_boss(bosses);

  for (cc_type::iterator k = cc.begin(); k != cc.end(); ++k)
    insert_boss(*k, cc);
```

4 Related Work

CLU [8] is one of the earliest contributions which offers language support for defining iterators as operations on programmer-defined container types. Since the programmer has total control over how iteration is defined, supporting complete traversal would be possible, perhaps by adapting one of the approaches discussed here. In [8], the authors mention the potential usefulness of such iterators but develop neither a formal definition nor any examples.

More recently, the work reported in [9] on list iterators in C++ covers issues associated with iterator integrity; i.e., problems which may arise when the object to which an iterator is pointing is deleted. Even though this work does not deal with complete traversals, the iterator integrity problem would come into play if we tried to do complete traversals on STL sequence containers (e.g., vectors or deques), because insertion in vectors and deques might require memory reallocation which invalidates all iterators pointing to the container in question. Except for the case of such iterator invalidation, complete traversals of STL sequence containers can be trivially programmed.

The programming language Sather, which was designed and implemented at the International Computer Science Institute (UC-Berkeley), is object–oriented with "...parameterized classes, object–oriented dispatch (late binding), multiple inheritance, strong typing, and garbage collection" [10], and supports containers and iterators. However they mention that their iterators "... will fail if the underlying data structure is modified." [10], which seems to imply that they do not tackle complete traversals automatically. Moreover, their suggestion to handle them is by trying to detect situations under which modifications to the container can happen while the iteration is in progress, which they mention could be problematic. From our point of view we do not have to detect those situations simply because this is a property which naturally arises in our definition of complete traversals.

Karla, which stands for the Karlsruhe Library of Data Structures and Algorithms, developed at the Faculty of Informatics (Karlsruhe University), also supports the concepts of containers and iterators [3, 5, 4]. This library was implemented by using Sather-K, which is "... a statically typed, type-safe variant of Sather" [3, 6]. Sather-K iterators are called *streams*, which, as is mentioned in [10], are a generalization of Sather's iterators; however as far as we know this generalization does not address the issue of an iterator being able to handle modifications to their container(s) while the iteration is in progress.

Very recent related work are the Java Generic Library (JGL) [13] and the so-called *collection framework* in the `java.util` package, part of the Java 2 platform [17]. The first is strongly based on the STL design, the second can be seen as a subset of STL. In both cases, modification of the underlying container through insertions while performing an iteration is discouraged; thus complete traversals were not considered in the design of these components.

In summary, a broad sampling of the literature perceives complete traversals as something negative that must be avoided. A good example of this perception is illustrated by the following *programming principle* [2]:

"**Principle 14** *Never modify a data structure while an associated* `Enumeration` *is alive*"

In this context, an `Enumeration` is the simplest container traversal that is directly supported by Java. Our definition of complete traversals captures its non-deterministic nature and establishes conditions under which such traversals are determinate, and terminate. To the best of our knowledge, no such formal framework for treatment of complete traversal problems has been previously developed.

5 Summary and Future Work

We have defined the complete traversal of a given container C by a function $\mathcal{F}$ as an iteration scheme which consists of iterating over C applying, at each iteration, $\mathcal{F}(x, C)$ which might possibly modify C by inserting new elements into it. The iteration should stop when all elements currently in C have been processed. By using standard rewrite rule techniques we have expressed the non-deterministic nature of this class of iterations, and have found conditions under which they are determinate. We have also given sufficient conditions for their termination.

In order to offer packaged solutions to programmers who need to use this class of iteration schemes, we have presented two approaches to perform complete traversals implemented on the platform provided by STL. Our first approach is in terms of generic algorithms which require that the iterated function create a queue to hold the generated elements. One algorithm handles containers with unique keys, while the other handles containers with multiple keys. By taking special precautions in the case of multiple keys, we remain independent of the details of particular implementations of sorted associative containers.

Our second approach is based on a complete container adaptor. The main features of this adaptor are the special iterators and insert operation it provides, by which the programmer can obtain complete traversals using a function that directly inserts new elements in the container.

The time complexities of both our approaches are asymptotically equivalent. However, the approach based on generic algorithms stores just the elements which are generated at each iteration in the function's queue, while the container adaptor stores all container elements in its iteration list. On the other hand, our complete container can be used with any STL associative container (including existing extensions such as hashed containers and any future extensions meeting the requirements of associative containers), while the generic algorithms can be used with sorted associative containers only.

Besides the examples given in this paper, there are two general arguments we can make for the usefulness of our design of these components as templates. First, one indication of the need for iteration schemes that can handle modifications to the underlying data structure is the frequent mention of the idea in the literature, albeit to warn against it as something not supported. Second, we did not design our templates in a vacuum; we took care to make them fit into an existing, widely-used framework, STL. The many programmers who are familiar with this framework, either by direct experience working in C++ or indirectly through the influence it has had on Java libraries, will be able to use or adapt our components with no significant change in the way they think about component-based programming, beyond having the freedom to use STL-like iteration schemes in even more situations than before.

There are still other approaches we are exploring, such as the result of melding the complete container idea with the approach of keeping just the generated elements. We are also interested in trying iteration schemes which do deletions as well as insertions. The connection between complete traversals and iterator integrity is also on our list of future work. We are also exploring the relationship between complete traversals and what we call *iterator trajectory functions*; i.e., function objects which describe a specific way of traversing a container. Lastly, we plan to use the components presented in this paper to solve problems that can be seen as instances of complete traversals, and to measure the performance of both approaches with randomly-generated instances.

Acknowledgments We thank Jesús Yépez for his assistance in the early stages of this work, and Ricardo Baeza-Yates and two anonymous referees for their valuable comments and suggestions.

References

[1] ISO/IEC FDIS 14882. *International Standard for the C++ Programming Language.* American National Standards Institute (ANSI), X3 Secretariat, 1250 Eye Street NW, Suite 200, Washington, DC 20005, 1998.

[2] D. Bailey. *Java Structures—Data Structures in Java for the Principled Programmer.* WCB/McGraw-Hill, 1999.

[3] A. Frick, W. Zimmer, and W. Zimmermann. *Karla: An Extensible Library of Data Structures and Algorithms. Part I: Design Rationale.* Karlsruhe University, Faculty of Computer Science, August 1994.

[4] A. Frick, W. Zimmer, and W. Zimmermann. On the design of reliable libraries. In *Technology of Object-Oriented Programming (TOOLS'17)*, pages 12–23. Prentice-Hall, 1995. Available from `http://i44www.info.uni-karlsruhe.de/~zimmer/karla/index.html`.

[5] A. Frick and W. Zimmermann. *Karla: An Extensible Library of Data Structures and Algorithms. Part II: Usage for Beginners.* Karlsruhe University, Faculty of Computer Science, January 1995.

[6] G. Goos. Sather-K, the language. Technical report, Karlsruhe University, April 1996.

[7] G. Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems.* Academic Press, New York, 1980.

[8] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[9] H. J. Messerschmidt. List iterators in C++. *Software—Practice and Experience*, 26(11):1197–1203, 1996.

[10] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in sather. *ACM TOPLAS*, 18(1):1–15, Jan. 1996. Available from `http://www.icsi.berkeley.edu/~sather/`.

[11] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[12] David R. Musser. Automated theorem proving for analysis and synthesis of computations. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, New York, 1989.

[13] ObjectSpace. Objectspace—JGL, the Generic Collection Library for Java. `http://www.objectspace.com/jgl`, 1997.

[14] A. J. Sánchez-Ruíz. *On Automatic Approaches to Multi-Language Programming via Code Reusability.* Phd dissertation, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, USA, 1995.

[15] Silicon Graphics. Standard Template Library Programmer's Guide. `http://www.sgi.com/Technology/STL`, 1997.

[16] A. A. Stepanov and M. Lee. The standard template library. Technical Report HP-94-93, Hewlett-Packard, 1995.

[17] Sun Microsystems. Java 2 Platform API Specification. `http://java.sun.com/products/jdk/1.2/docs/api/index.html`, 1999.