Complete Traversals as General Iteration Patterns

William Klostermeyer
Department of Computer and Information Sciences
University of North Florida
Jacksonville, FL 32224, USA
klostermeyer@hotmail.com

David Musser
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
musser@cs.rpi.edu

A. J. Sánchez-Ruíz
Department of Computer and Information Sciences
University of North Florida
Jacksonville, FL 32224, USA
asanchez@unf.edu

Abstract Iterators are of central importance in the design of generic algorithms and collections, serving as intermediaries that enable generic algorithms to be written without concern for how collections are stored and collections to be written without having to code a large number of algorithms on them. A limitation of collection frameworks such as the C++ Standard Template Library (STL), the Java 2 platform, and the Java Generic Library is that they do not allow *complete traversals*, in which a collection might be modified by adding elements to it while it is being traversed by means of its associated iterators. Problems requiring complete traversals are fairly common, and while there are various *ad hoc* ways of solving them, programmers should ideally have at their command an efficient packaged solution. After reviewing prior work on extending generic algorithms and collections to support complete traversals, this paper describes a new generic component for complete traversals based on a design pattern extracted from a commonly used

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35672-3_13

J. Gibbons et al. (eds.), *Generic Programming*

implementation of STL sorted associative containers. Also presented are the results of experiments to assess the performance of complete traversal components by randomly generating abstract instances of complete traversal problems. Finally, several theoretical results relating to computability and undecidability are established. It is shown that complete traversals are general enough that any computable relation can be expressed as an instance of them. Even assuming termination, however, the problem of determining whether a given complete traversal pattern always terminates in the same collection is shown to be undecidable.

1. Introduction

Generic collection frameworks, such as the ones implemented by the C++ Standard Template Library (STL) [1, 9, 11], the Java 2 platform [15], and the Java Generic Library (JGL) [12], provide abstractions such as collections,[1] iterators, and algorithms. Generic algorithms operate on collections without resorting to knowledge associated with backing data structures by using iterators as proxies. The main kind of operations on collections that iterators enable is a *traversal*, namely the process of going through all the elements in a collection without repetition.[2] The Java 2 collection framework provides iterators that are a simple generalization of the enumeration types that are commonly defined in C-like languages. On the other hand, STL and JGL provide a hierarchy of iterator categories that provide progressively more capabilities, such as bidirectional traversal or random access to elements. Regardless of the kind of iterator used to implement a traversal, all these frameworks leave undefined what happens if one tries to add elements to the underlying collection while the iteration is in progress.

A *complete traversal* of a collection is informally defined as an iteration scheme that allows modification of the collection by adding new elements to it, while the iteration is in progress. In case the iteration always terminates with the same final collection, regardless of the traversal order, we call the iteration *determinate* and also refer to it as *the* complete traversal of the collection. In two previous articles [4, 10] we have formally characterized complete traversals by means of standard rewriting theory, presented generic components for complete traversals implemented on top of the C++ STL, and analyzed their computational complexity.

[1] In this paper we use the terms *collection* and *container* as synonyms.

[2] In STL and JGL there are "multiple collections" which allow multiple occurrences of equivalent objects. For this class of collections, a traversal must go through each occurrence exactly once.

These results are summarized in section 2. Section 3 presents a new generic component constructed on top of the STL by reusing a design pattern extracted from a commonly used implementation. Section 4 then reports results of experiments to assess the performance of this and the previously developed generic components by randomly generating instances of abstract complete traversal problems.

Some of the rewriting-related theory of complete traversals may be useful in characterizing solutions to database locking and mutual exclusion problems, an application that we briefly discuss in section 5.

In sections 6 and 7 we turn to more theoretical issues concerning complete traversals. We prove that any computable relation can be expressed as a complete traversal pattern, settling some of the questions posed in [10]. Such generality naturally raises questions of decidability, including whether a given complete traversal pattern is determinate. We show this problem is undecidable, even in restricted formulations.

We discuss related work in section 8 and conclude the paper in section 9 by posing new questions for future work.

2. Complete Traversals

Let us consider the family of iterations of the form

$$\begin{array}{c}\textbf{for all } x \textbf{ in } C \\ \mathcal{F}(x, C)\end{array}$$

where C is a collection, and $\mathcal{F}$ is a function that can possibly modify C by adding new elements to it. Intuitively, the idea behind an iteration like this is that all elements are drawn from C exactly once and processed by $\mathcal{F}$, including those that are added to C while the iteration evolves in time. In the context of collection frameworks such as STL and JGL, one must deal with *multiple* and *unique* collections, where the former allow multiple occurrences objects (e.g., multisets and multimaps), and the latter do not (e.g., sets and maps).

Formally, if we denote by $\hat{\mathcal{F}}(x, C)$ the collection of elements to be inserted into C by the call $\mathcal{F}(x, C)$, we can define complete traversals in terms of a rewriting relation as follows [10].

Definition 1

1 *Given any finite collections C and D such that $D \subseteq C$ and a (program) function $\mathcal{F}$:*

(a) if $C = D$,(C, D) is said to be a normal form, or irreducible;

(b) *otherwise, 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 *A* traversal sequence *for a collection* 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* $D_0 = \emptyset$.

3 *Such a traversal sequence is said to be* terminating *if* (C_n, D_n) *is irreducible (equivalently, if* $C_n = D_n$*).*

4 *A* complete traversal *of a collection* C *using* $\mathcal{F}$ *is any terminating traversal sequence for* C *using* $\mathcal{F}$.

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

Definition 2

1 *Let* $\rightarrow^*$ *denote the reflexive, transitive closure of* $\rightarrow$. *We say that* p *and* q *are* joinable *if and only if there exists an* r *such that* $p \rightarrow^* r$ *and* $q \rightarrow^* r$.

2 *The relation* $\rightarrow$ *is said to be:*

(a) uniformly terminating, *if and only if there is no infinite sequence of the form* $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$.

(b) 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.*

(c) 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.*[3]

It is well known that the combination of uniform termination and local confluence of a rewriting relation imply confluence [7]. Thus an immediate application is the following theorem (from [10]):

Theorem 3 *If a traversal successor relation is uniformly terminating and locally confluent, then every complete traversal of a collection* C *using a function* $\mathcal{F}$ *results in the same final collection. We also say that the complete traversal computation is* determinate.

[3]Note the difference from confluence: only one step is taken from p rather than arbitrarily many.

For simple examples of both determinate and indeterminate iterations, define

$\mathcal{F}(x, C)$:

1: $count \leftarrow count + 1$
2: **if** $count = 1$ **then**
3: $\quad save \leftarrow x$
4: **else**
5: $\quad$ **if** $count = 2$ **then**
6: $\quad\quad$ insert $f(save, x)$ in C
7: $\quad$ **end if**
8: **end if**

where *count* and *save* are state variables of $\mathcal{F}$ and *count* is initialized to 0.[4]

If C is initially $\{1, 2\}$ and $f(x, y) = x + y$ (a symmetric function) then all traversals of C terminate in $\{1, 2, 3\}$, and thus the corresponding rewriting relation is confluent for this C (and for any other initial C). On the other hand, if $f(x, y) = x - y$ (a nonsymmetric function) then a traversal that begins with 1 inserts $1 - 2 = -1$ and terminates with $\{1, 2, -1\}$ whereas the one that begins with 2 inserts $2 - 1 = 1$ and terminates with $\{1, 2\}$. In this case the rewriting relation is not confluent for this initial C (or any other initial $C = \{a, b\}$ where a and b are distinct integers).

In the case of an f like $f(x, y) = \lfloor (x + 1)/y \rfloor + 1$, although it is not symmetric, all complete traversals of $C = \{2, 3\}$ terminate with the same collection (C unchanged), since $f(2, 3) = 2$ and $f(3, 2) = 3$.

One could always eliminate the use of state in $\mathcal{F}$, though with some loss of clarity, by encoding the state in C. Note that doing so makes $\mathcal{F}$ depend on C. The theorems in [10], which assume $\mathcal{F}$ neither depends on C[5] nor has its own state, are thus not applicable to proving determinacy in cases of functions $\mathcal{F}$ that have state. On the other hand, it seems useful to consider the case in which $\mathcal{F}$ does have state, since such functions are frequently used in generic programming (e.g., function objects, also called functors, as used for example in STL, can have state).

We have also previously built generic components that provide a complete traversal capability within the STL framework. They work on

[4] By a state variable of a function we mean a variable whose value is preserved between calls of the function. Ways in which state variables can be implemented include global variables or member variables of a class in which the function is a member function. For an example of an indeterminate iteration in which $\mathcal{F}$ does not have state, see [10]. Further discussion of the role of state in $\mathcal{F}$ follows the example.

[5] When we say $\mathcal{F}$ does not depend on C we mean that $\mathcal{F}(x, C_1) - C_1 = \mathcal{F}(x, C_2) - C_2$ for all C_1 and C_2.

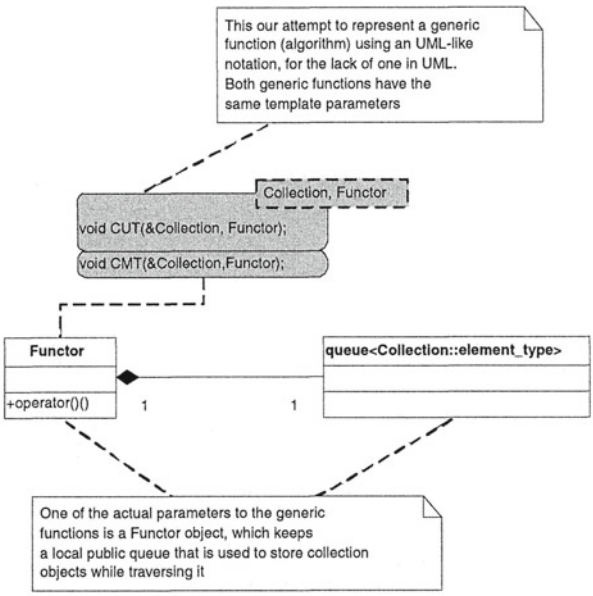

Figure 1. The generic algorithms as parameterized frameworks

STL *sorted associative collections* (SACs) using two approaches, namely generic algorithm components and a collection adapter [4].[6] These components differ both in their interfaces and in performance; the performance aspect is discussed further in section 4.

The generic algorithm component is built as two nested iterations. The outer iteration traverses the collection by using its native (bidirectional) iterators, calling F (the function that inserts elements into the collection), which is supposed to be constructed in such a way that it keeps a public queue containing the elements generated in a single call to it. The inner iteration treats the elements just stored in F's queue to determine whether they should be inserted and processed immediately (i.e., when the element is inserted before the current iteration point), or inserted and processed later (i.e., when the element is inserted after the current iteration point). Since the problem of determining where a newly inserted element stands with respect to the current iteration point varies from unique collections to multiple collections, we needed to implement two generic algorithms `complete_unique_traversal` (CUT), and `complete_multiple_traversal` (CMT). Figure 1 shows our attempt at

[6]The code is available from `http://www.cs.rpi.edu/~musser/gp/traversals/`.

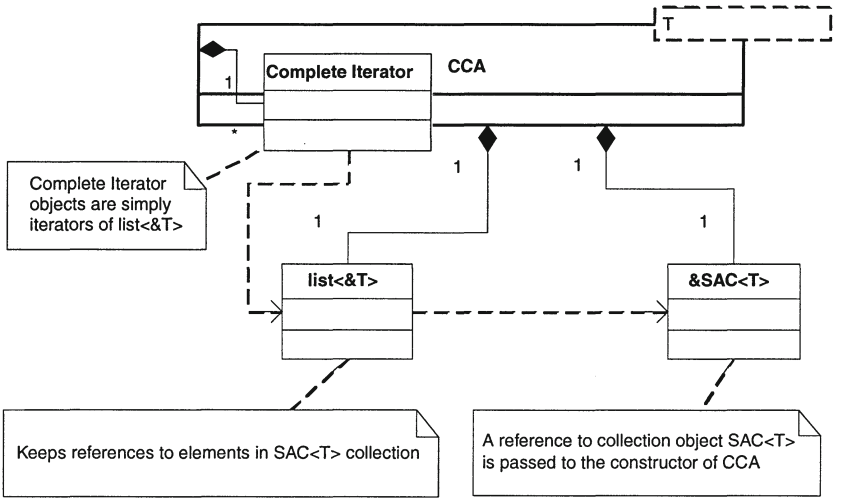

Figure 2. The collection adapter as parameterized framework

describing the approach as parameterized frameworks using a UML-like notation [13].

The second generic component was implemented as a collection adapter (called a complete collection adapter), which wraps the original collection in order to provide its users with a new class of bidirectional iterators that transparently perform a complete traversal. Internally, an STL list is kept so that each time an element is inserted into the collection, a reference to it is appended to the list. By using this list, the adapter can implement what we have called complete iterators. While an iteration with one of these iterators is in progress, the following invariant holds: "the element referenced by the current iterator value is the element to be processed, all the elements to the left of it have been processed, and all elements to the right of it will be processed next." The adapter also implements an insertion operation such that F can directly insert the elements into the collection without the aid of an auxiliary queue. In summary, this approach splits the iteration into two orthogonal components, the insertion component and the traversal component. Figure 2 shows the adapter as a parameterized framework.

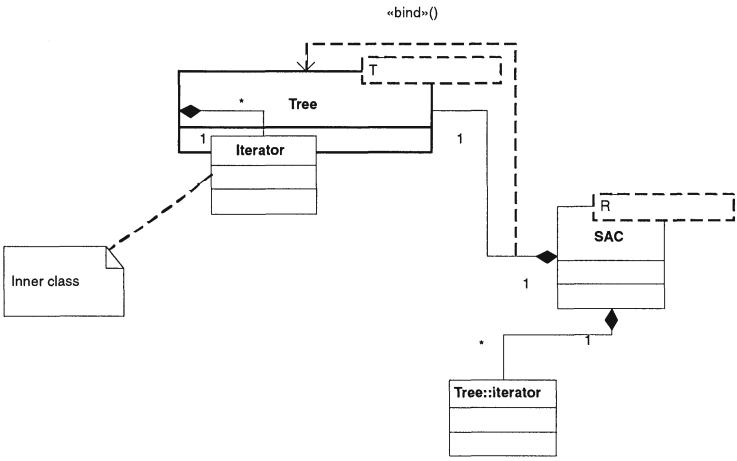

Figure 3. Design pattern associated with a commonly used implementation of SACs

3. A New Complete Traversal Generic Component

A commonly used implementation of STL SACs follows the design pattern shown in figure 3. Briefly, a red-black tree [2] is used as the backing data structure to achieve efficient look-up, insertion, and deletion operations.[7]

By reusing this design, we have implemented a new class of collections, which we call Complete SACs (CSACs). We modified the red-black tree data structure in such a way that it is always possible to bidirectionally traverse the elements in the collection in the order of their insertion. We simply added a couple of links to the structure that supports the nodes in the tree and made sure that every time an element is inserted, it is threaded according to both the order of insertion and the reverse order of insertion. Clearly, the dual set of actions must be performed when an element is deleted from the tree. Since the associated red-black tree template class offers several member functions for insertion and deletion, it might seem to be a complicated and error-prone task to modify them all appropriately. However, this turned out not to be necessary in view of one of the key design criteria of STL, that all container classes are

[7]This design pattern is used, for example, in SGI STL [14], which is derived from the original Hewlett-Packard implementation and from which the GNU project's implementation of STL is derived. A somewhat different design pattern, based on inheritance instead of composition, is used in at least one STL implementation [11].

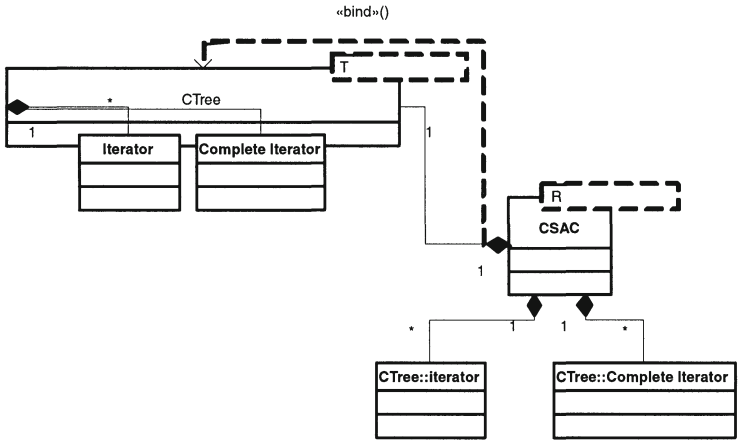

Figure 4. Design pattern of the new generic SAC components

parameterized by allocator types whose objects have the sole responsibility for storage allocation and deallocation. In implementing container classes, further isolation of storage management is possible through restricting calls of the allocators to a few intermediary member functions. In the case of the red-black tree implementation, we found that we only needed to modify exactly two member functions, namely the ones that do allocation and deallocation of tree nodes. The problem was then reduced to a simple instance of inserting/deleting an element into/from a doubly linked list. However, extra care needed to be taken to preserve the invariants associated with special iterator values such as `begin()` and `end()`. In summary, all operations associated with this modification are constant time operations, which implies that the new class has the same asymptotic behavior as the original one. Figure 4 shows the resulting design pattern that describes the new generic components.[8]

4. Generic Components Performance Assessment

Our approach to assessing the performance of these generic components is based on random generation of complete traversal problems (CTPs). There are two elements that define a CTP, an initial collection C, and a function $\mathcal{F}(x, C)$. In [10] we proved that if $\mathcal{F}$ does not depend on C

[8]The code is available from `http://www.unf.edu/~asanchez/CSAC`, including the code that generates the performance assessment results presented in section 4.

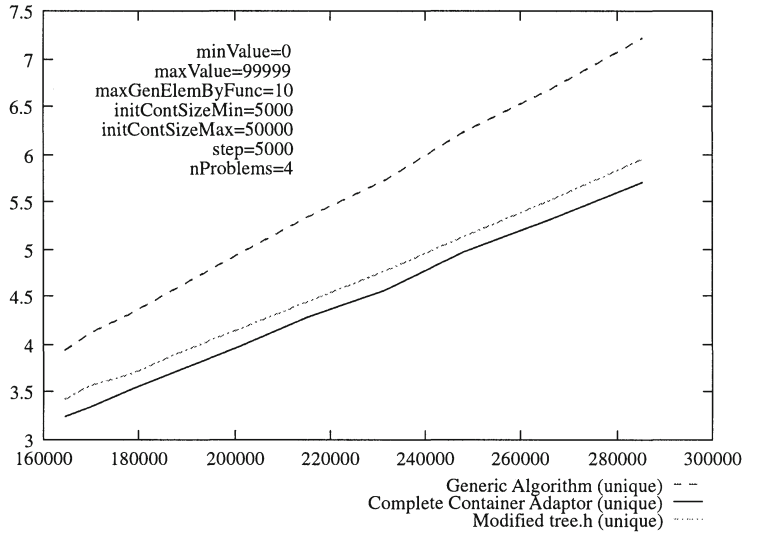

Figure 5. Performance assessment results for unique collections. The horizontal axis measures the number of insertion attempts and the vertical axis measures time in seconds.

(see footnote 5), then the associated CTP is determinate. Therefore, to avoid nondeterministic behavior, we restrict the random generation of CTPs to this family of functions $\mathcal{F}$. The following parameters were used to randomly generate pairs (C, F).

- `minValue` and `maxValue`: lower and upper bounds for elements generated by $\mathcal{F}$, respectively.
- `maxGenElemByFunc`: maximum number of elements generated by $\mathcal{F}$ in a single call.
- `initContSizeMin` and `initContSizeMax`: minimum and maximum size of initial collection.
- `step`: size of step from one problem to the next, in collection size units.
- `nProblems`: number of problems used to compute the average processing time associated with a given number of insertion attempts.

Using these parameters, the random generation of CTPs can be decomposed into two subproblems.

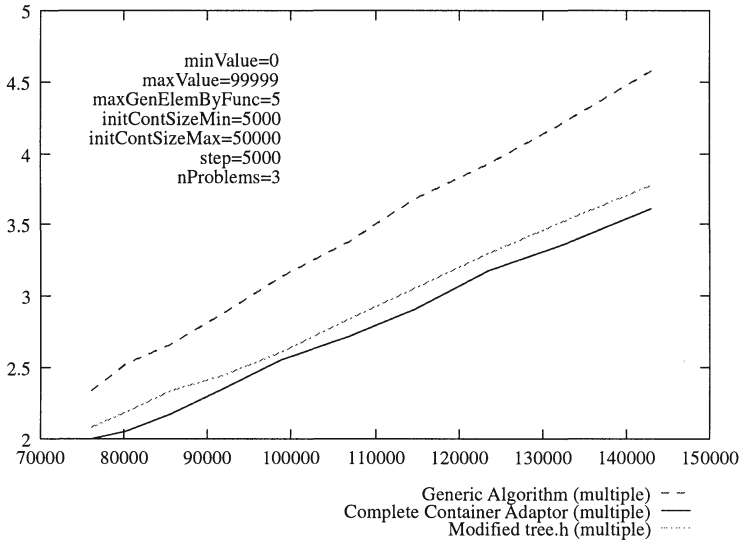

Figure 6. Performance assessment results for multiple collections. The horizontal axis measures the number of insertion attempts and the vertical axis measures time in seconds.

- Subproblem A: randomly generate `nProblems` functions with the given profile, which can be tabulated because they do not depend on the collections. The value of a function at a particular point in the range `[minValue,maxValue]` is computed by first randomly deciding how many elements the function will generate, then randomly generating that many elements.

- Subproblem B: use the functions generated by the previous step to randomly generate initial collections with the given profile, and then solve the `nProblems` CTPs. We keep track of the number of insertion attempts associated with each problem, and the time needed to perform the complete traversal, in order to compute the average to be plotted.

Figures 5 and 6 show the results of this experiment for unique and multiple collections, respectively. We used GNU g++ 2.8.1, on a SUN workstation Ultra 1, running Solaris 2.5.1. For an asymptotic analysis, see [4]. From the results we see that the performance of the new generic component is close to that of the complete container adaptor (and significantly better than that of the generic algorithms). Since the new component has an interface that better conforms to containers in the original STL

framework, it appears to have the overall advantage among the alternatives.

5. An Application to Database Problems

We give another example of the importance of confluence. Consider a database (i.e., the collection) that is accessed by multiple users. One of the fundamental problems in databases is that of locking and mutual exclusion. In particular, at what level of granularity should the items in the database be locked in order that each transaction be successfully, and correctly, completed. If we assume that each transaction operates on a single "atomic" element from the collection, then if the underlying function that is being computed on the database is confluent, then we may safely lock at the atomic level, and allow concurrent reads of items in the database. In this case, the final resulting collection will be the same, no matter what order the operations (the "writes" to the database) are performed. However, if the underlying function is not confluent, it may be necessary to disallow concurrent reads of the database, e.g., to lock the entire collection, so that the operations are forcibly ordered in a first-come first-serve order. More generally, it is desirable in the situation of nonconfluence to assess what final collection values are "correct" and to impose additional constraints on access to ensure correctness. In database theory, any sequence of operations is deemed to be correct if they are *serializable*, i.e., if an interleaved sequence of (concurrent) operations results in the same final state as *some* serial, i.e., sequential, sequence of the same operations. In other words, in a serializable sequence of operations any of the possible resulting states are correct. If this is one's chosen definition of correctness, then, for example, one may use a generic programming solution (e.g., an extension of the STL, as described in [4]) to implement a complete traversal in the case on nonconfluence. However, if the user wishes to constrain the order in which elements from the collection are accessed, then the implementation must be flexible enough to accommodate this.

6. Complete Traversals as General Iteration Patterns

In this section we settle one of the questions posed by our previous work, namely whether the computation of the closure of a set under a set of relations can be expressed as a complete traversal, by proving an even more general result, namely that any computable relation can be expressed as a complete traversal.

We shall need the following result from the text by Papadimitriou and Lewis [8], pages 38–39.

Theorem 4 *Let P be a closure property defined by relations on a set D and let A be a subset of D. Then there is a unique minimal set B that contains A and has property P.*

Theorem 5 *Any computable relation can be modeled as a complete traversal pattern.*

Proof: Assume the computable relation R is represented as a nondeterministic Turing machine, denoted TM. A TM has a transition function operating on a set of *states*, where a state is a tuple consisting of one of the finite states of the TM paired with a "snapshot" of the (finite) contents of the TM's tape. Hence each computation, given some input x, terminates in a final state y such that $R(x, y)$. To model this computation as a complete traversal, we let the collection C consist of a set of these TM states, and initially we place in it only the "start" state, which is the tuple of the start state plus the initial tape contents, representing the input to the TM. The complete traversal proceeds by processing the unique state in C that has yet to be processed and in the course of so doing, adding one state to C, or in the case of termination of the TM, not adding a state to C, in which case the complete traversal terminates. □

This enables us to resolve a question from [10]. The **Closure of a Set under a Relation** problem, see for example [8], is the problem of determining the closure of a given finite set, under a set of relations.

Corollary 6 *The* **Closure of a Set under a Relation** *can be modeled as a complete traversal pattern, and, moreover, one that is determinate.*

Proof: The closure can be determined by a computable function—in fact, by a polynomial-time algorithm (e.g., [8, p. 40]). By treating this function as a computable relation, we conclude from the theorem that it can be modeled by a complete traversal pattern. The uniqueness result stated in Theorem 4 implies that no matter what permutation of the elements of the set is used as input, the computation will terminate in the same state. Hence the rewriting relation for the complete traversal formulation is confluent. □

There obviously exist problems that are not complete traversals (more generally, which do not terminate). Consider an appropriately couched version of the Halting Problem, for example. Additionally, since it can be proved that "*any polynomial-time algorithm can be rendered as the*

computation of the closure of a set under some relations of fixed arity" [8], and it is well-known that there exist computable functions that cannot be computed in polynomial time [6], we can say that the family of problems expressible as complete traversals is larger than the family of problems expressed as the closure of a set under some relations of fixed arity.

7. Deciding Confluence

Suppose we have a computable relation R. Thus, we can model R as a complete traversal pattern. But is it confluent? Some iterations are confluent, and some are not, as shown by the examples given in section 2. The key to those examples is that the result depends on the order in which elements of the collection are processed. Note also that in the example of the $\mathcal{F}$ based on the nonsymmetric function $f(x, y) = x - y$, we have confluence for some initial input collections and not for others. We define the **General Complete Traversal Confluence Problem** to be to determine whether a complete traversal pattern is confluent for *all* input collections. Is this problem decidable? We prove below that it is undecidable; in fact we give two distinct proofs, first with a formulation of the problem in terms of Turing machines and again with a formulation directly in terms of confluence of the rewriting relation of the complete traversal pattern.

Let G_h be a set of nondeterministic, two-tape Turing Machines $\{M_1, M_2, \ldots\}$ such that M_i halts on all inputs. That is, each computation path of M_i halts for each input x. Let us denote that on input x, M_i halts in state $M_i(x)$. Furthermore, we assume that each M_i treats its input x as an n-tuple, with appropriate separators, e.g., each element of an n-tuple is given in unary (using the appropriate number of 1 bits), with a 0 bit delimiting each element of the n-tuple and 00 signifying the end of the n-tuple. The input (i.e., the initial values in the collection) and the subsequent contents of the "collection" are stored on tape one, and tape two is used as a work tape. Each TM in G_h uses its nondeterminism in a specific fashion: at each step, an element of tape one is chosen nondeterministically to be used in the next computation step. Each TM in this set G_h can be simulated by a deterministic two-tape TM that simulates each possible computation path of the nondeterministic TM and writes the result of each such computation path (i.e., the number of the final state), sequentially on a special output region of tape one. We then wish to know if all the values written on this output region are equal. If so, then the problem, for input x anyway, is confluent. The General Complete Traversal Confluence Problem (GCT-CP) asks

whether a machine $M_i \in G_h$ is confluent for all x. To prove that GCT-CP undecidable, we consider a restricted problem.

Let L_h be the set of deterministic two-tape Turing Machines $\{M_1, M_2, \ldots\}$ such that M_i halts on all inputs. As before, we assume that M_i treats its input x as an n-tuple encoded in unary with appropriate separators. We may assume that the input is recorded on its own read-only tape. Now let L_p be the subset of L_h such that M_i is in L_p if and only if for all $n > 0$, all n-tuples $< x >$, and all permutations $\Pi(x)$ of $< x >$, we have $M_i(x) = M_i(\Pi(x))$. Of course, we require (and can easily verify) that $\Pi(x)$ is a "legal" n-tuple, in the format described above. Our question is: Can we decide whether M, a machine known to be in L_h, is also in L_p? We refer to this as the **Restricted Complete Traversal Confluence Problem** (RCT-CP).

The important distinction between the restricted and general problems is that in the restricted problem, any modifications made to the collection during the course of the computation do not affect the outcome of the computation. That is, it is a purely deterministic computation, depending only on the input (and, importantly the *order* of the input), whereas in the general problem, two seemingly identical TM's could produce different outputs given the same input collection, if they "probed" their collections differently, thereby modifying their collections in different ways. The restricted problem, in essence, allows no modification of the collection. In some sense, this is an issue of how the data structure used in a computation can affect the output of the computation (see section 9), as the data structure chosen can impose an order to an input collection.

An important point to observe is that we cannot use Rice's Theorem (see [8]) to answer this question, as we are making the assumption that all Turing machines M_i halt on all inputs.

For the proof of undecidability of RCT-CP we rely on the following definition and theorem.

Definition 7 *A total recursive function f is symmetric if and only if $f(x, y) = f(y, x)$ for all x, y in its domain.*

Theorem 8 *There is no decision procedure to determine whether any total recursive function f is symmetric.*

Proof: The proof is by reduction of the Halting Problem to the stated problem. Let $f_{M,t}(x, y) = 1$ if Turing machine M halts on input tape t in less than x steps but does not halt on t in less than $y + 1$ steps. This can be true only if $y < x$ and M halts in time z for some z such that $y < z < x$. There is an simple algorithm that produces the function $f_{M,t}$

given M and t, by simply simulating M for y steps. It is easy to see that the function $f_{M,t}$ is symmetric if and only if M does not halt on tape t. But note that for each M and t, $f_{M,t}$ is a total recursive function. So if we had a decision procedure to determine whether any total recursive function was symmetric, then we would have a decision procedure for the Halting Problem. □

Corollary 9 *The Restricted Complete Traversal Confluence Problem is undecidable.*

Proof: The formulation of RCT-CP is in terms of Turing machines that halt on all inputs, but of course, by the Church-Turing thesis, we could equivalently formulate it in terms of total recursive functions. The problem of deciding symmetry of total recursive functions is then easily reduced to RCT-CP. Thus if we had a decision procedure for RCT-CP, we would have one for symmetry of total recursive functions. □

Theorem 10 *The General Complete Traversal Confluence Problem is undecidable.*

Proof: Suppose to the contrary that the GCT confluence problem were decidable and let M be a Turing Machine that decides it. We show that we can use M to decide the RCT Confluence Problem. Let M_q be an input to the RCT-CP; i.e., we wish to decide if problem Q satisfies the conditions of restricted confluence, where M_q is a Turing Machine that computes the function associated with problem Q. Modify M_q so that once its computation is complete, the contents of tape one (which stores the elements of the collection) consist only of the initial contents collection and the number of the final state of the computation. It is a straightforward computation to modify the "program" of M_q to do this. Input the modified M_q to machine M. It follows that machine M then outputs that the modified M_q satisfies the GCT confluence problem if and only if M_q satisfies the RCT confluence problem. Hence the GCT-CP is undecidable. □

We now give a second proof of Theorem 10, this time using a formulation in terms of the rewriting relation associated with complete traversals, as defined in section 2. Thus the problem is that of deciding whether the rewriting relation for complete traversals is confluent.

Second Proof: By a slight modification of the construction of the $\mathcal{F}$ in the example in section 2, we can prove undecidability of GCT-CP by directly reducing to it the problem of whether a function is symmetric. Assume we have a decision procedure for GCT-CP. Given a function

$f : N \times N \to N$, where N is the set of all natural numbers, construct $\mathcal{F}$ as follows:

$\mathcal{F}(x, C)$:

1: $count \leftarrow count + 1$
2: **if** $count = 1$ **then**
3: $\quad save \leftarrow x$
4: **else**
5: $\quad$ **if** $count = 2$ **then**
6: $\quad\quad$ insert $|save| + |x| + 1 + f(save, x)$ in C
7: $\quad$ **end if**
8: **end if**

where the term $|save| + |x| + 1$ is added to ensure that the value inserted is not already in the initial set. Let R be the rewriting relation for $\mathcal{F}$ and the initial C be any set containing two distinct integers. Then R is confluent if and only if f is symmetric, since if there are values m and n for which $f(m, n) \neq f(n, m)$ then when initially $C = \{m, n\}$ there are two distinct final values for C,

$$\{m, n, |m| + |n| + 1 + f(m, n)\}$$

and

$$\{m, n, |m| + |n| + 1 + f(n, m)\},$$

whereas if f is symmetric then for any initial $C = \{x, y\}$ the final value for C is always

$$\{x, y, |x| + |y| + 1 + f(x, y)\}.$$

Thus by using the assumed decision procedure for GCT-CP, we could decide whether f is symmetric, contradicting Theorem 8. □

8. Related Work

Iterators play a key role in many component libraries, not just those that provide generic components, but no general-purpose library that we are aware of provides for complete traversal patterns. In the Java 2 platform [15], the documentation of interface `java.util.Iterator` contains the following warning in the description of method `remove`, which "removes from the underlying collection the last element returned by the iterator": *"The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method."* It is therefore clear that iterators in this framework do not directly support complete traversals.

The JGL libraries [12] can be considered to be a Java implementation of the design behind STL, and thus its iterator hierarchy does not support modifying a container while is being traversed.

Although STL's design did not consider supporting complete traversals from its inception, several elements present in its design and implementations allow for cleanly extending the framework to include them, in several alternative forms. In particular, implementing the new component presented in section 3 was simplified both by the way the STL design isolates storage allocation and deallocation and by an easily identifiable design pattern found in the most commonly used implementation of STL's sorted associative containers.

Lieberherr and Wand report on DJ, a Java library that can be used to specify traversals associated with object graphs in the context of Aspect-Oriented Programming [16]. These traversals must consider the possibility of the underlying graph being modified, and hence are complete traversal instances.

Eichelberger and Gudengberd [3] present a partial UML characterization of STL but do not identify design patterns. In fact, we are not aware of any previous work analyzing STL in light of the concept of design patterns. STL's implementation of sorted associative containers uses two of the so called GOF patterns,[9] namely *Adapter* and *Iterator*. We used the *object adapter* pattern, which signifies that the adapter keeps a reference to the "adaptee." The other flavor of this pattern is known as the *class adapter* (i.e. the adapter inherits from the adaptee), which is used by the implementation discussed in Plauger, Stepanov, Lee, and Musser [11]. A known weakness of the latter with respect to the former is that it cannot handle the adaptation of potential subclasses of the adaptee.

9. Conclusions and Future Work

In this paper we have described a new generic component for complete traversals based on a design pattern extracted from a commonly used implementation of STL sorted associative containers. In comparisons with two previous methods of extending generic algorithms and collections to support complete traversals, we have seen that the new component appears to have the overall advantage, since it is close in performance and has an interface that better conforms to containers in the original STL framework. We have also shown that complete traversals are general enough that any computable relation can be expressed as an instance of them. Even assuming termination, however, the problem of determining confluence, i.e., whether a given complete traversal pattern always terminates in the same collection, has been shown to be undecidable.

[9] GOF stands for "Gang of Four," and refers to the four authors of [5].

An interesting direction for future work would be to impose constraints on the nature of the input collections and programs/iteration patterns and show that, for this restricted domain, one could decide whether a program (in the domain) is confluent for all collections in the domain.

Acknowledgments

We would like to thank Bob McNaughton for his assistance with the proof of theorem 8. Two former students, Eric Gamess (now with the University of Puerto Rico at Mayaguez) and Jesús Yépez (now with Emida Technologies), programmed the new generic component and conducted the performance assessment presented in sections 3 and 4. Last, but not least, we would like to extend our gratitude to the anonymous referees for their constructive comments.

References

[1] M. H. Austern. *Generic Programming and the STL — Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1999.

[2] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[3] H. Eichelberger and J. Wolff v. Gudenberg. UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany,* October 2000.

[4] Eric Gamess, David R. Musser, and Arturo J. Sánchez-Ruíz. Complete traversals and their implementation using the standard template library. *CLEI Electronic Journal,* 1(2), 1998. Available from `http://www.dcc.uchile.cl/~rbaeza/clei/`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns CD.* Addison-Wesley, 1995.

[6] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[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] H. R. Lewis and Ch. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, second edition, 1998.

[9] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, second edition, 2001.

[10] David R. Musser and Arturo J. Sánchez-Ruíz. Theory and generality of complete traversals. In Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors, *Generic Programming,* volume 1766 of *Lecture Notes in Computer Science,* pages 91–101. Springer-Verlag, 2000.

[11] P.J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library.* Prentice-Hall, 2001.

[12] Recursion Software, Inc. JGL libraries. `http://www.recursionsw.com`.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[14] Silicon Graphics, Inc. Standard Template Library programmer's guide. `http://www.sgi.com/tech/stl`.

[15] Sun, Inc. Java 2 Platform, Standard Edition (J2SE). `http://java.sun.com`.

[16] Mitchell Wand and Karl Lieberherr. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University, May 2001.