Programmatic Testing of the Standard Template Library Containers

Jason McDonald* Daniel Hoffman† Paul Strooper‡

May 11, 1998

Abstract

In 1968, McIlroy proposed a software industry based on reusable components, serving roughly the same role that chips do in the hardware industry. After 30 years, McIlroy's vision is becoming a reality. In particular, the C++ Standard Template Library (STL) is an ANSI standard and is being shipped with C++ compilers. While considerable attention has been given to techniques for *developing* components, little is known about *testing* these components.

This paper describes an STL conformance test suite currently under development. Test suites for all of the STL containers have been written, demonstrating the feasibility of thorough and highly automated testing of industrial component libraries. We describe affordable test suites that provide good code and boundary value coverage, including the thousands of cases that naturally occur from combinations of boundary values. We show how two simple oracles can provide fully automated output checking for all the containers. We refine the traditional categories of black-box and white-box testing to specification-based, implementation-based and implementation-dependent testing, and show how these three categories highlight the key cost/thoroughness tradeoffs.

1 Introduction

Our testing focuses on *container classes*—those providing sets, queues, trees, etc.—rather than on graphical user interface classes. Our approach is based on *programmatic testing* where the number of inputs is typically very large and both the input generation and output checking are under program control. We do not rely on keyboard and mouse capture and playback, or on file and screen capture and comparison. We have found these techniques unsuitable for automated testing of container classes, where the interface is through function calls rather than keyboard, mouse, screen, and file.

*Software Verification Research Centre, School of Information Technology. The University of Queensland, 4072, Australia, jasonm@csee.uq.edu.au

†Dept. of Computer Science, Univ. of Victoria, P.O. Box 3055 MS7209, Victoria, B.C., V8W 3P6 Canada, dhoffman@csr.uvic.ca

‡Software Verification Research Centre, School of Information Technology, The University of Queensland, 4072, Australia, pstroop@csee.uq.edu.au

With programmatic testing, the economics of testing are fundamentally changed. Traditionally, testing approaches have sought to minimize the number of test cases, with reduction of test suite development cost a secondary goal. This approach makes sense when humans are involved in some aspect of each test case, making test cost roughly proportional to the number of test cases run. With programmatic testing the priorities are reversed. The primary goal is to minimize test suite development cost; significant automation makes test execution nearly free, even for thousands of cases.

Traditionally, test cases are divided into two categories. Black-box (or specification-based) test cases are based solely on the specification. White-box (or structural) test cases are driven by the implementation, especially the control structure of the code. Over time, the black-box/white-box distinction has proven useful. In developing test suites for STL, however, finer distinctions are needed to explain the key trade-offs between cost and thoroughness:

- *Specification-based*: all test cases are chosen on the basis of the specification, inputs are supplied and outputs observed using only public member functions, and output correctness is based solely on the specification.

- *Implementation-based*: some test cases are chosen based on the implementation, but all inputs are supplied and outputs observed using only public member functions, and output correctness is based solely on the specification. For example, suppose we have a vector class providing dynamic resizing on insert and delete, and a particular implementation that allocates memory in blocks of 10 elements. Then an implementation-based test suite might focus on vectors of size 9, 10, 11, 19, 20, 21, etc. This test suite could still be used, though less effectively, on an implementation that reallocates all of its dynamic memory on insertion and deletion.

- *Implementation-dependent*: some test cases depend on aspects of the implementation not mentioned in the specification. Implementation-dependent test cases arise from two main sources: code modification and specification nondeterminism. For example, many test suites use embedded assertions to dynamically evaluate the correctness of internal data structures. Nondeterminism is common where order is important. Suppose that we are given a function specified to sort a list of keyed records, and an implementation that does stable sorting. A test suite that relied on stability in output checking would be implementation-dependent: it would fail when applied to an unstable sort implementation.

These categories are important because of the impact they have on test development and maintenance. Specification-based tests are the most desirable because they are the simplest and are portable across multiple implementations of a given specification. Implementation-dependent test suites are the least desirable because they are the most complex and the least portable. Still, an implementation-dependent test suite is the best choice in some cases. Implementation-based test suites are a compromise: more complex than specification-based suites, but still portable across implementations.

Section 2 of this paper outlines the literature on class testing and Section 3 briefly describes the ClassBench framework which is used as the testing platform for the STL test

suite. Section 4 presents the STL in more detail. Section 5 outlines the key testing issues and describes test suites that illustrate the use of the three testing categories in detail. Section 6 summarises our experience testing the STL containers and Section 7 presents our concluding remarks.

2 Related Work

Testing classes is similar to testing software modules, and early work by Panzl [17] on the regression testing of Fortran subroutines addresses some of the issues. The DAISTS [7], PGMGEN [10], and Protest [11] systems all automate the testing of modules using test cases based on sequences of calls.

In the literature on object-oriented testing [2], considerable attention has been paid to class testing. Frankl [6] has developed a scheme for class testing using algebraic specifications. Fiedler [5] describes a small case study on testing C++ objects. The ACE tool, developed by Murphy, et al., [15] supports the testing of Eiffel and C++ classes, and provides strong evidence for the benefits of testing classes before testing the system using the classes. Arnold and Fuson [1] discuss class testing issues and techniques. The text by Marick [14] devotes two chapters to class testing, focusing primarily on test plans. The FOOT framework, developed by Smith and Robson [19], allows a class to be tested either manually or automatically, using either a built-in testing strategy or a previously defined test sequence.

The work just described constitutes important progress in automated class testing but contains precious few actual test suites for industrial class libraries. Only the work by Fiedler and the work by Murphy describe such suites. The Fiedler paper provides no information on the testing techniques used. The Murphy paper uses a tool adapted from a predecessor to ClassBench [10, 12].

3 ClassBench

The ClassBench framework [13, 9] is a collection of tools and techniques supporting the automated testing of C++ classes. With ClassBench, the tester performs three tasks:

1. *Develop the testgraph.* The testgraph nodes and arcs correspond to the states and transitions of the class-under-test (CUT). However, the testgraph is vastly smaller than the CUT state/transition graph.

2. *Develop the* `Oracle` *class.* The `Oracle` is used to check the correctness of the behaviour of the CUT. It provides essentially the same operations as the CUT but supports only the testgraph states and transitions. As a result, the `Oracle` is usually significantly cheaper to implement than the CUT, and is more reliable as well. The `Oracle` member functions are invoked frequently for output checking and, perhaps surprisingly, for input generation.

3. *Develop the* `Driver` *class.* The `Driver` class contains `cut` and `orc`, instances of the CUT and `Oracle`. `Driver` also provides three public member functions: `reset` places

Containers	Adaptors	Iterators	Algorithms
set	stack	input	non-mutating sequence
multiset	queue	output	mutating sequence
map	priority queue	forward	sorting related
multimap		bidirectional	generalized numeric
vector		random	
deque		istream	
list		ostream	

Table 1: STL inventory

both `cut` and `orc` in the initial state, `arc` generates the transition in `cut` and `orc` associated with the testgraph arc traversed, and `node` checks that, in each node, the `cut` behavior is consistent with the `orc` behavior.

To add flexibility to the framework, each test suite may have a *test suite parameter*. This parameter is typically used to determine the size of the container to be tested, so that containers of different sizes may be tested without recompiling the test suite.

To support the tester, the ClassBench framework provides a graph editor, a graph traversal algorithm, and support code in the form of skeletons and demonstration test suites. The testgraph editor provides the facilities commonly available in today's graph editors. Testgraphs can be accessed from disk; nodes and arcs can be added and deleted. The testgraph traversal classes automatically traverse a stored testgraph, calling the appropriate `Driver` member functions: `reset` is called at the start of each path, `arc` is called each time a testgraph arc is traversed, and `node` is called each time a testgraph node is visited.

4 The Standard Template Library

The STL [16] provides the services outlined in Table 1. Seven containers are provided, each implemented as a generic class. There are also three container adaptors; each of these is a wrapper around one of the other containers, adapting the container to produce a new type of container. There are seven types of iterators that provide an interface that is used by the algorithms to access the containers. The iterators can be thought of as a generalization of the concept of pointers. Eighty-one algorithms are included, divided into the four groups shown in Table 1. Several characteristics of the STL make it significantly different from other class libraries:

- In addition to the member functions specific to the container type, each container also provides an iterator interface. The iterators provide a standardized interface to all containers. Each algorithm accesses a container solely through the iterator interface, allowing a single algorithm to operate on a variety of containers, and avoiding the "algorithm explosion" problem that results when each algorithm must be implemented separately for each container [18]. Because the algorithms and containers

```
// constructors
list();
list(const list<T>& x);
// assignment amd equality
list<T>& operator=(const list<T>& x);
template <class T> bool operator==(const list<T>& x, const list<T>& x);
// access member functions
bool empty();
size_type size();
size_type max_size();
// insert member functions
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
// erase member functions
void erase(iterator position);
// special list functions
void splice(iterator position, list<T>& x);
```

Figure 1: *List* member functions

can be combined so freely, it is not sufficient to test the code on a few expected cases; thorough testing will require *many* test cases.

- The STL is a part of the new ANSI C++ standard and is already shipped with most C++ compilers. Thus, STL reliability is critical and development of a substantial test suite is justified.
- The STL standard includes performance specifications for each algorithm and container. Thus, an STL conformance test suite should evaluate performance as well as functionality.

In this paper, we focus on the testing of the seven containers and the three container adaptors, including the iterators provided by each container. Testing of the STL algorithms and performance testing are both work in progress.

4.1 Example: The *List* Container

To illustrate the interface provided by the STL containers, we use the *List* container as an example. *List* provides a type of sequence. Sequential access to the list is allowed via the bi-directional iterators provided by the class. Elements may be inserted or deleted at any iterator position in constant time.

The interface to *List* varies slightly for different versions of the STL, and the interface we describe here is the Hewlett Packard reference implementation. This interface contains 42 member functions. Figure 1 shows some of these member functions. The type `T` represents the generic parameter to *List* that identifies the type of elements stored in *List*. The types

`iterator` and `size_type` are defined by *List*, and define an iterator for the *List* class and a type that is used to describe quantities of list elements respectively.

Figure 1 shows two of the constructors for *List*. The constructor `list()` creates an empty list, whereas `list(`l`)` creates a copy of the list l. There are two other *List* constructors.

There are operators for list assignment (`operator=`) and list equality (`operator==`).

There are sixteen member functions that can be used to access elements in a list or to obtain other information about a list. For example, the function `empty()` returns true or false according to whether the list is empty, `size()` returns the number of elements in the list, and `max_size()` returns the maximum number of elements that can be stored in a list.

There are six member functions to insert elements in a list and four member functions to erase elements from a list. The call `insert(`i`,`x`)` inserts x at the position pointed to by iterator i, and `insert(`i`,`n`,`x`)` inserts n copies of x at the position pointed to by iterator i. The first call returns an iterator that points to the newly inserted element. Similarly, the call `erase(`i`)` deletes the element pointed to by iterator i.

Most of the algorithms in the STL are generic and can be applied to a number of containers. However, for efficiency reasons, some algorithms work only for *List*, or require a specialized version for *List*. For example, the `splice` function can be used to splice parts of one list into another list. There are three versions of `splice`; for the one shown in Figure 1, the call `splice(`i`,`l`)` destructively copies all elements from list l to the position pointed to by iterator i. *List* also provides functions to remove all elements from a list equal to a given value, to remove duplicate elements from a list, to merge two sorted lists, to reverse a list, and to sort a list.

The bi-directional iterator for *List* provides `operator*` that can be used to both read and write values pointed to by an iterator. It also provides `operator==` and `operator!=` to test equality and inequality of two iterators. Finally, it provides prefix and postfix versions of both `operator++` and `operator--` to traverse a list in a forward or backward direction.

5 STL Testing

5.1 Testing Issues

The test suite for the STL containers is driven by four key decisions, each briefly described below.

5.1.1 Programmatic Testing

Exhaustive testing is not feasible for any STL container. Consider the *List* class. Even if only lists of size 10 are considered, most of the possible states will never be reached. With care, however, all members of certain families of tests can be executed affordably. For example, for a particular list of 100 elements, it is feasible to delete every element from the list, checking the entire list contents after every deletion. This thinking illustrates the new economics of programmatic testing. Traditionally, testing methods have sought to minimize the number of test cases; testing was at least partially manual and each execution

of each test case was expensive. With programmatic testing, repeating a test case costs very little. Consequently, the goal is to minimize test suite development, while keeping machine costs acceptable.

5.1.2 Iterators and Algorithms

The STL provides 10 containers and 81 algorithms. Most of the algorithms work with most of the iterators, creating a potential problem for the tester: it would be very expensive to develop tests for every algorithm/container combination. To avoid this expense, we aim to thoroughly test the iterators for each container and then test all the algorithms on a single container. This approach is motivated by the central design decision of the STL itself: allow an algorithm to access a container only through the iterator interface. In the STL, the algorithm/container separation forces careful specification of the iterator interface. In our STL test suite, this separation forces careful testing of the iterator interface, but permits significant savings in testing the algorithms.

As explained previously, this paper focuses on the testing of the containers and their iterators.

5.1.3 Oracles

Because programmatic testing generates so many test cases, test oracles must be carefully planned and implemented. Otherwise, the development cost of the oracle code will be unaffordable. With the large number of STL containers, considerable sharing of oracle code is essential. In our STL test suite, we use two oracles: one for the set-like classes—*Set*, *Multiset*, *Map*, *Multimap*—and one for the sequence-like classes—*Vector*, *Deque*, *List*, *Stack*, *Queue*, and *Priority_Queue*.

5.1.4 Testing Templates

It is important to test template classes for different types, and the C++ type conversion mechanism provides a convenient method to do so. To test a new type `T`, all we need to do is provide a new class `Element` with a constructor that maps `int` to `T`. Usually, we test classes for one builtin and one user-defined type [9]. In the STL test suite described in this paper, we test most classes using integers only. The *Deque* class was also tested with an integer-like class that allowed the element size to be varied.

5.2 Specification-based Testing: List

A specification-based test suite was developed for *List* using the ClassBench framework and the testgraph is shown in Figure 2. The state `EMPTY` represents a list containing no elements; `ALL`, `ODD` and `EVEN` represent lists of contiguous and non-contiguous ascending element values; `REV` represents a list of descending element values; and `DOUBLE` represents a list containing duplicate element values. The test suite parameter, *parm*, determines the maximum element value for the list. Elements are in the range 0..*parm*-1. Beside each state, the state value is shown for a test suite parameter of 10. The initial state, distinguished by an arc with no source node, is `EMPTY`.

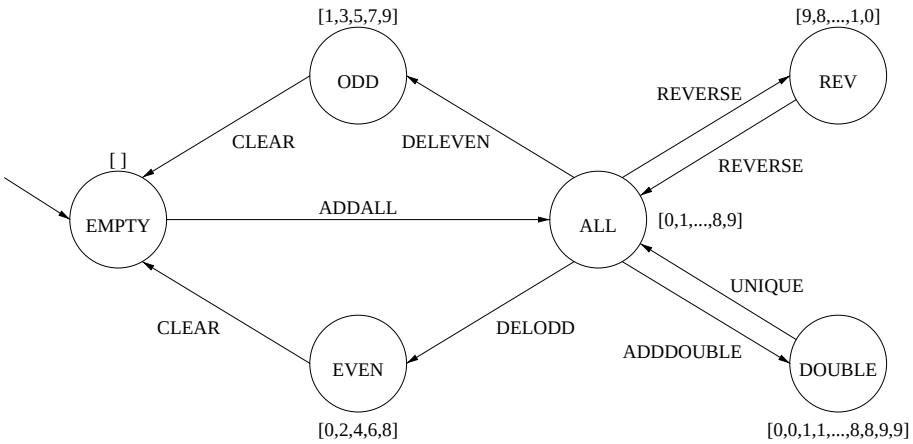

Figure 2: Testgraph for *List* class

The oracle class, which is also used in the test suites for *Vector*, *Deque*, and the container adaptors, stores the sequence of elements in an array. The oracle provides 21 member functions and two simple iterator classes for manipulating and observing the sequence. Unlike the STL iterators, the oracle iterators signal exceptions when illegal actions take place.

Part of the declaration of the `Driver` class for the test suite is shown in Figure 3. The driver follows the usual form of a ClassBench driver, providing the `reset`, `arc`, and `node` functions. Due to the large number of member functions to be checked at each node, the `node` function delegates its work by calling other check functions. We briefly describe several of these below.

The `CheckCut` function compares the CUT with the oracle by checking the results of `size` and `empty` and by iterating over the CUT and oracle and comparing each element.

`CheckIterators` tests the bi-directional iterators by traversing forwards and backwards over the CUT and oracle using the pre- and postfix versions of `operator++` and `operator--`, and comparing the results of `operator*`. The constant and non-constant versions of both the forward and reverse iterators are tested in this manner.

`CheckCopyConstructor` calls the copy constructor to create a copy of the CUT and then checks it against the oracle. The other constructors are checked similarly.

`CheckOpEqualEqual` creates additional instances of the CUT and oracle representing each testgraph state and then checks that `operator==` returns the appropriate value when comparing each of these with the current state. The `=` and `<` operators are checked similarly.

`CheckInsert` tests all four `insert` functions. For each, it attempts to insert a new element at each iterator position within the CUT and checks the result against the oracle. The checking for `splice` and `erase` is done in a similar manner.

The driver does not test the `max_size` function. The specification [20] does not give a specific value, but indicates that a list should be able to hold at least `max_size` elements. However, none of our testing platforms have sufficient memory to construct a list of `max_size` elements. The return value of `max_size` on all of our testing platforms would require a minimum of 4 gigabytes of memory.

```
class Driver {
public:
      Driver(int parm0);
      void reset();
      void arc(int arcNum);
      void node() const;
private:
      TestCut cut;
      Oracle orc;
      int parm;

      void CheckCut() const;
      void CheckIterators() const;
      void CheckCopyConstructor() const;
      void CheckOpEqualEqual() const;
      void CheckInsert() const;
};
```

Figure 3: Partial declaration of `Driver` class for *List* test suite

The specification-based test suite for *List* achieves statement coverage of 99.2%. The only statements not executed are those in the function body of the `max_size` member function. Since the specification-based test suite meets our goals for coverage, we did not conduct any further testing on *List*.

5.3 Implementation-based Testing: Deque

Like *List*, the STL *Deque* class provides a type of sequence. However, *Deque* allows random access rather than the sequential access provided by *List*. Further, *Deque* supports constant time insertion and erasure of elements at either end of the sequence, but insertion in the middle of the sequence takes linear time.

The *List* and *Deque* interfaces are similar. While *Deque* does not provide any of the specialist *List* functions for splicing, merging, sorting, reversing, removing duplicates or removing all instances of an element, it adds some other services. *Deque* adds constant and non-constant versions of `operator[]`. *Deque* also provides random access iterators. In addition to the functionality of a bi-directional iterator, a random access iterator can move multiple elements forwards or backwards along a container using the `+=` and `-=` operators, and it can access elements forwards or backwards of the current position using the `[]` operator.

5.3.1 Specification-based Test Suite

The specification-based test suite for *Deque* is similar to the test suite for *List*. The test suite uses the same oracle that was used in the *List* test suite. Duplicate elements and the ordering of element values are not significant in *Deque*. Thus, the testgraph states `DOUBLE` and `REVERSE` and their associated arcs were omitted from the testgraph and driver. In the

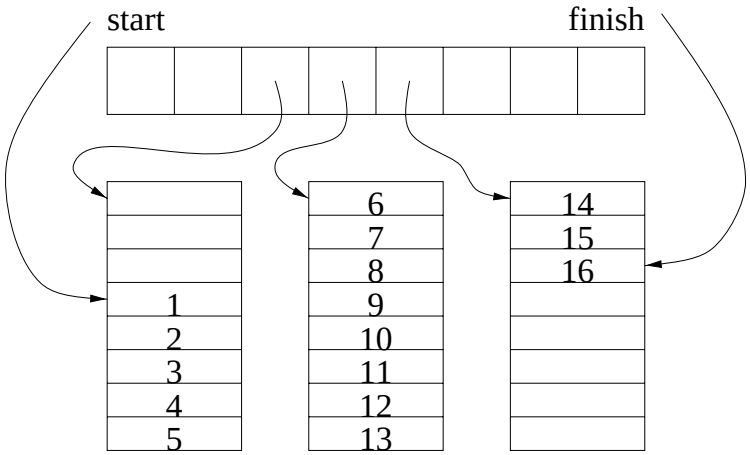

Figure 4: A typical instance of the `deque` data structure

driver, the code to test the specialist list operations that are not provided by *Deque* was omitted, and code to test both versions of `operator[]` was added. The `max_size` function was not tested, for the same reasons as outlined for the same function of the *List* class. The specification-based test suite for *Deque* was executed with test suite parameters of 1, 2, 5, 10 and 100. This yielded statement coverage of only 86.8% percent, which did not meet our goal of exercising every statement apart from those in the `max_size` function.

Deque is implemented as a sequence of fixed-size blocks of storage, each of which contains one or more elements of the deque, as illustrated in Figure 4. These blocks are allocated and deallocated at each end of the deque as it grows or shrinks. An array of pointers to the blocks is stored and the class begins by allocating the pointer at the middle of the array to point to the first block allocated for the deque. When the deque has reached the start or the end of the pointer array, the array is reallocated to allow the deque to grow further. A significant portion of the *Deque* implementation is devoted to managing this data structure.

A close examination of the implementation reveals three reasons why the specification-based test suite failed to achieve full statement coverage. Firstly, the test suite never builds a deque that spans more than one block in the data structure. Secondly, the test suite never attempts to insert a group of elements larger than one block into a deque by using the "copy iterator range" forms of the constructor and `insert` functions or the form of `insert` that inserts multiple copies of a single element. Finally, the test suite never causes the block pointer array to be reallocated.

5.3.2 Implementation-based Test Suite

The solution to these problems is to develop an implementation-based test suite for *Deque*.

The tested implementations of STL use a block size equal to the maximum of the element size and 4096 bytes. Thus, to make the structure span multiple blocks a deque containing at least 1025 integers (an integer takes four bytes) is required. However, when the test suite parameter is increased to 1025, the test suite takes an unacceptably long time

```
class integer
{
      integer(const int);
      integer& operator=(int);
      integer& operator=(const integer&);
      int operator==(const integer&);
      ...
private:
      int value;
      char blank[EXTRA_SIZE];
}
```

Figure 5: Partial declaration of the `integer` class

to execute, mainly due to the complex checking required for the deque `insert` functions.

We solved this problem by substituting the class `integer`, shown in Figure 5, for `int` as the element type for the deque. The `integer` class is a wrapper around `int` that uses `EXTRA_SIZE` more bytes of memory. Thus, we reduce the number of elements needed to fill a block and consequently the execution time to test deques that span multiple blocks. For the implementation-based test suite, we set `EXTRA_SIZE` to 60, producing a *Deque* with 64-byte elements stored in blocks each containing 64 elements.

We also added code to the driver to test the insertion of groups of elements larger than one block, and code to cause the block pointer array to be reallocated. The implementation-based test suite was executed with test suite parameters 1, 2, 63, 64, 65, 127, 128 and 129, corresponding to testing around the boundaries of one, two and three blocks.

The implementation-based test suite revealed a performance error in the `insert( iterator, const T&)` function. The specification for this function states that it takes constant time if the element is inserted at the front or back of the deque, and linear time in the minimum of the distances from the insertion point to the front and back of the deque, if the element is inserted at any position between the first and last elements. However, the if statement that is meant to select the minimum of the two distances always selects the distance from the front, making the insertion take time linear in the distance from the front of the deque. This error was detected when the statements that use the distance from the back of the deque were not executed by the appropriate test cases.

The implementation-based test suite for *Deque* achieved statement coverage of 98.9%. The only statements not executed were the body of `max_size` and the else part of the erroneous if statement in `insert( iterator, const T&)` which is unreachable due to the error discussed above.

5.4 Implementation-dependent Testing: Set

The STL *Set* class stores a sorted collection of elements without duplicates. The class is parameterized by the element type and a comparison function for that element type. The *Set* class supports insertion and deletion of elements in logarithmic time.

The interface to *Set* is similar to that of *List*. Since *Set* stores a sorted collection, it

does not provide the functions for adding, retrieving and removing elements from the front and back of a list (`push_front`, `pop_front`, etc). Since duplicate elements are not allowed, the multiple-element versions of the constructor and `insert` functions are also omitted. Finally, *Set* does not provide the specialist *List* functions, but instead provides some other services for accessing the set. `Set` provides bi-directional iterators which return elements in the order defined by the comparison function object.

5.4.1 Specification-based Test Suite

The specification-based test suite for *Set* is similar to that of *List*. The testgraph has four states: `EMPTY`, `ODD`, `EVEN`, and `ALL`. These represent the empty set, the set of odd integers less than the test suite parameter, the set of even integers less than the test suite parameter, and the set of all integers less than the test suite parameter respectively. The oracle, which is also used for *Multiset*, *Map* and *MultiMap*, stores the number of occurrences of each element less than the test suite parameter in an array. The driver checks each member function of *Set* in a similar manner to the *List* driver. The driver does not test the `max_size` function for the reasons outlined earlier.

The *Set* class is implemented using a red-black tree data structure [4]. Red-black trees are similar to AVL trees, but use less stringent restrictions on the balancing of the tree to reduce the amount of rebalancing required, thus improving efficiency.

The specification-based test suite was executed with test suite parameters of 2, 5, 10 and 100. This yielded statement coverage of 97.3%. The unexecuted code consisted of the `max_size` function and part of the code that rebalances the tree after insertions and deletions.

While the specification-based test suite achieves reasonably good statement coverage, it achieves very low coverage of the underlying red-black tree data structure. Very few of the possible red-black trees are created and exercised by the test suite. Whereas the *List* and *Deque* data structures do not depend on the values of each element, the red-black tree used by *Set* depends on the element order determined by the comparison function. Thus, the small set of states used in the testgraph is not sufficient to adequately exercise the red-black tree structure. It turns out that generating specific red-black trees using a specification-based test suite is known to be difficult [3].

5.4.2 Implementation-dependent Test Suite

To adequately cover the red-black tree data structure, we constructed an implementation-dependent test suite.

Ruskey [8] has developed an algorithm for generating, in string form, all red-black trees containing a particular number of nodes. We added a function `setTree` to the red-black tree implementation to construct a red-black tree data structure from a red-black tree string. The implementation of this function is dependent on the particular implementation of the red-black tree, and the test suite is thus implementation-dependent rather than implementation-based.

The implementation-dependent test suite takes a tree string as the test suite parameter and calls `setTree` to construct the appropriate instance of the data structure. This instance

	Spec-based		Imp-based		Imp-dependent	
Class	Lines	Coverage	Lines	Coverage	Lines	Coverage
Vector	399	94.4%	n/a	n/a	n/a	n/a
List	584	99.2%	n/a	n/a	n/a	n/a
Deque	388	86.8%	393	98.9%	n/a	n/a
Set	367	97.3%	n/a	n/a	351	99.3%
MultiSet	372	98.6%	n/a	n/a	n/a	n/a
Map	420	97.8%	n/a	n/a	n/a	n/a
MultiMap	399	98.4%	n/a	n/a	n/a	n/a
Stack	167	100%	n/a	n/a	n/a	n/a
Queue	167	100%	n/a	n/a	n/a	n/a
PriorityQueue	126	100%	n/a	n/a	n/a	n/a

Table 2: Summary of STL Container testing results

is then exercised using a simplified version of the checking done by the specification-based test suite.

The implementation-dependent test suite was executed once for each red-black tree containing 10 or fewer nodes. Statement coverage of 99.3% was achieved. The only statements not executed were in the function body of `max_size`.

6 Summary of Test Suites and Results

Specification-based test suites were developed for the seven container classes and the three container adaptor classes. Two oracles were developed: one for the sequence-like classes (including the three adaptor classes) and one for the set-like classes.

Additional code was developed to test the iterator interface of each container class. All seven container classes use either bi-directional or random-access iterators, so two different checking routines were developed. The appropriate code was compiled into the test suite for each class.

For each test suite, a test plan was developed, describing the strategies to be used to test each member function. The test plans were useful for checking the completeness of test suites.

Three different implementations of STL were tested. The Hewlett-Packard implementation was tested on the AIX operating system using the IBM xlC compiler. The GNU implementation was tested under Solaris 2.5 and Linux 2.0.32 using the GNU g++ 2.7.2.2 compiler. The Rogue Wave implementation was tested on Windows NT 4.0 using Borland C++ 5.02. The same test suites were used for all three implementations.

Table 2 shows, for each test suite, the number of lines of code in the driver and the percentage of basic blocks covered by the test suite.

The coverage information was obtained using the `tcov` utility with the xlC compiler. This utility is not available for g++ or Borland C++. The coverage percentages are based on the number of *reachable* basic blocks in each class. For example, some code in the red-

black tree implementation is only reachable when duplicate element values are permitted, and is thus not reachable in *Set* or *Map*.

The red-black tree data structure used for the *Set* class is also used for *Multiset*, *Map*, and *Multimap*. Thus the specification-based test suites for these classes, while achieving good statement coverage, do not achieve good coverage of the data structure. Implementation-dependent test suites for these classes would rectify this problem, and may be produced in the future.

Two errors were found during testing. Both of these were in the *Deque* class. The error in `insert( iterator, const T&)` noted in Section 5.3 was present in all three tested implementations. Additionally, the GNU version of *Deque* appears to have a bug in `insert( iterator, size_type, const T&)` which non-deterministically corrupts memory. As yet, we have not been able to determine the exact nature of this error.

7 Conclusions

With the software component industry finally coming of age, the issue of component correctness has become an important one. For example, an STL implementation is now shipped with each C++ compiler. While most of the current implementations are probably based closely on the original HP implementation, divergence has begun and will surely continue. Each variant will have many users, each of whom will use the library in unpredictable ways. Consequently, test suites exercising a wide variety of uses are badly needed.

We have presented a programmatic approach to automated component testing. Test execution is fully automated, from input generation to oracle. Thousands of cases are generated, to exercise important values and, especially, their combinations. Programmatic testing has a different economic basis to other testing methods: cost is dominated by the development effort of the test programs while the number of test cases is largely irrelevant.

We have demonstrated the feasibility of the programmatic approach with test suites developed for all the STL containers, and applied to STL implementations from three different vendors. The suites exercised almost all reachable statements, achieved good coverage of important state and parameter values and their combinations, and revealed two errors. The test suites also demonstrated the tradeoffs between specification-based, implementation-based, and implementation-dependent testing.

References

[1] T.R. Arnold and W.A. Fuson. Testing in a perfect world. *Commun. ACM*, 37(9):78–86, 1994.

[2] Robert V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6:125–252, 1996.

[3] H. Cameron and D. Wood. Insertion reachability, skinny skeletons, and path length in red-black trees. *Information Sciences*, 77:141–152, 1994.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] S.P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, pages 69–74, April 1989.

[6] P.G. Frankl and R.K. Doong. The ASTOOT approach to testing object-oriented programs. *ACM Trans. on Software Engineering Methodology*, 3(2):101–130, 1994.

[7] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program Lang. Syst.*, 3(3):211–223, July 1981.

[8] D. Hoffman, F. Ruskey, R. Webber, and L. White. Tree generation and automated class testing. *IEEE Trans. Soft. Eng.*, 1998.

[9] D. M. Hoffman and P. A. Strooper. ClassBench: A methodology and framework for automated class testing. *Software Practice and Experience*, 27(5):573–597, May 1997.

[10] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105. IEEE Computer Society, October 1989.

[11] D.M. Hoffman and P.A. Strooper. Automated module testing in Prolog. *IEEE Trans. Soft. Eng.*, 17(9):933–942, September 1991.

[12] D.M. Hoffman and P.A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach.* International Thomson Computer Press, 1995.

[13] D.M. Hoffman and P.A. Strooper. The testgraphs methodology: Automated testing of collection classes. *Journal of Object-Oriented Programming*, Nov./Dec. 1995.

[14] B. Marick. *The Craft of Software Testing.* Prentice Hall, 1994.

[15] G. Murphy, P. Townsend, and P.S. Wong. Experiences with cluster and class testing. *Commun. ACM*, 37(9):39–47, 1994.

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

[17] D.J. Panzl. A language for specifying software tests. In *Proc. AFIPS Natl. Comp. Conf.*, pages 609–619. AFIPS, 1978.

[18] M. Sirkin, D. Batory, and V. Singhal. Software components in a data structure precompiler. In *Proc. Intl. Conf. Software Engineering*, pages 437–446. IEEE Computer Society, May 1993.

[19] M.D. Smith and D.J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 4(6):45–53, June 1992.

[20] ANSI Standards Committee X3 (Information Processing Systems). Draft proposed international standard for information systems programming language C++. Document X3J16/96–0225 WG21/N1043, American National Standards Institute, December 1996.