Automated Testing of C++ Templates

Kirk Sayre and Jesse Poore
Software Quality Research Lab
University of Tennessee
203 Claxton Education Building
1122 Volunteer Boulevard
Knoxville, TN 37996
{sayre, poore}@cs.utk.edu

2nd September 2005

Abstract

The object-oriented design of software in C++ using class templates presents many advantages. However, testing template based C++ software involves some unique issues not usually faced in testing non-template based software. This paper will describe these testing issues and proposed solutions using an automated testing framework based on Markov chain usage models. The framework is being used to test a library of codes used in computational materials research.

1 Introduction

Section one provides a brief overview of C++ templates and a discussion of the issues involved in testing software designed using libraries of C++ class templates. Section two gives a high level description of a usage model-based automated testing framework designed to address these issues. An ongoing application project in computational materials research is described in section three. Results are summarized in section four.

1.1 C++ Class Templates

C++ class templates allow software to be developed using the concepts of generic programming [2]. Generic programming is the abstraction of parameterized procedural schemata, independent of underlying data representation, from specific, executable, efficient algorithms. These generic algorithms (expressed as templates) can then be instantiated with different data representations to address a variety of specific applications [1]. This allows an algorithm to be developed once and then used with many different types of data, thereby making the software easier to maintain and potentially more reliable. One or more C++ class templates can be used to implement an API, or application programming interface.
The following is the definition of a very small C++ class template that will be used as an example throughout the rest of the paper. This C++ class template provides some simple operations that can be performed on a list of numbers.

template <class NumberType,
        class NumberContainerType>
class NumberOps {

private:
        NumberContainerType the_number_list;
        size_t num_elements;

d public:

        NumberOps(NumberContainerType &a_number_list,
                size_t num_elements) :
                the_number_list(a_number_list),
                num_elements(num_elements) {} 

        NumberType avg() {
                NumberType r = 0;
                for (size_t x = 0; x < num_elements; x++) {
                        r += the_number_list[x];
                }
                return r/((NumberType) num_elements);
}

        NumberType max() {
                NumberType max =
                        std::numeric_limits<NumberType>::max() * -1;
                for (size_t x = 0; x < num_elements; x++) {
                        if (the_number_list[x] > max) {
                                max = the_number_list[x];
                        }
                }
                return max;
}

        NumberType min() {
                NumberType min =
                        std::numeric_limits<NumberType>::max();
                for (size_t x = 0; x < num_elements; x++) {
                        if (the_number_list[x] < min) {
                                min = the_number_list[x];
                        }
                }
        }
1.2 Automated API Testing

The basic process of testing an API is to create (either manually or automatically) a test harness (i.e., test program) which calls functions in the API with “interesting” data [3]. The test program is run using the API under test, and the results of running the test program are checked for correctness.

An automated testing framework can be designed to make it easier to run and assess many tests. Several aspects of API testing can be automated:

- Generation of test cases (i.e., test programs).
- Selection of test data.
- Execution of large groups of test cases.
- Checking system responses for correctness (i.e., the test oracle).
- Analysis of test results.

This framework is based on a usage model describing the expected use of the API under test.

1.2.1 Markov Chain Usage Models

In usage model-based testing [10], all possible uses of the software under test are represented as a finite state Markov chain, called a Markov chain usage model. A use of the software is a sequence of external stimuli applied to the software. The sequence begins with the software in a known externally observable state and progresses towards a terminate state. The states of use of the software are represented as states in the Markov chain. User actions, i.e., external stimuli, that cause changes in the current state of use are represented as arcs in the Markov chain. The probability that a user will perform a certain action given that the software is in a certain state of use is represented by state transition probabilities in the Markov chain.

1.3 Issues in Testing C++ Class Templates

While generic programming with C++ class templates provides significant benefits, there are some practical testing issues that must be addressed. Some of these issues are more general problems that must be addressed when testing an API, regardless of how the API was implemented, while others are unique to testing C++ template-based APIs.

A general solution to the issues involved in testing is usage model-based automated testing. A detailed presentation of how this automated testing solves the issues involved in the testing of C++ template based APIs follows.
1.3.1 Issue: APIs Evolve Over Time

Given that APIs tend to evolve over time, the tests of the API must also evolve over time. The necessity of changing the test set makes hard-coded test suites difficult to maintain. However, when performing usage model-based testing, test cases are generated automatically from a model describing the system under test. Changes to the system under test will be mirrored by localized changes to the usage model of the system, rather than by changes to many individual test cases. After changing the model the test suite can be automatically regenerated, thereby making the test suite easier to maintain than a hard-coded test suite.

1.3.2 Issue: Testing an API May Require Many Tests

Some APIs can only be adequately exercised by a large number of test cases. The testing of such APIs poses the problems of generating, running, and evaluating a large number of test cases. These problems are addressed by a usage model-based automated testing framework.

- **Test Generation** - Tests are generated automatically from a usage model.
- **Test Execution** - An automated testing framework supports running many tests without user intervention.
- **Test Evaluation** - An automated testing framework may be designed to support various automated test oracles, which allows for the automated evaluation of the results of running test cases.

1.3.3 Issue: Testing Must Take Into Account Expected Use

Expected use is explicitly addressed when building a usage model. Expected use of the system under test is represented in a usage model by the state transition probabilities in the model. Different types of use may be reflected in a usage model by changing the model’s state transition probabilities.

1.3.4 Issue: A Single Class Template Leads to Many Instantiated Classes to Test

A strength of using C++ class templates to implement an API is that a single C++ class template can be instantiated and used with different sets of template parameters. This advantage to software development leads to a problem for testing because a single C++ class template can be instantiated in many different ways, resulting in many different classes that need to be tested. A usage model for testing that class template must be designed in a generic manner, i.e., the usage model must not depend on the class template under test being instantiated with any particular data types. To support modeling in this manner additional functionality must be built into the testing framework to allow for the creation of a generic model and the generation of appropriately instantiated test cases from the generic model.
1.3.5 Issue: Class Templates Inherit Functionality From a Parent Class

The usage model for testing a class template derived from a parent class should be able to model the behavior inherited from the parent class in a general manner. This will allow for the same model to be used to test many class templates inheriting from the same parent class with minimal (possibly no) changes to the usage model. In addition, the automated testing framework must take advantage of commonalities in the class templates under test.

2 Overview of the Automated Testing Framework

A testing framework has been implemented to support the automated usage model-based testing of C++ templates. The automated testing process will be presented from a user’s perspective by walking through the steps in the testing process, with a discussion of the work products and actions associated with each step.

2.1 Testing Process

2.1.1 Usage Model Construction

The first step is to construct a usage model of the C++ template to be tested. In this model high level test actions representing state transitions will be augmented with C++ code implementing the test action. The model designer must ensure that the model is general, i.e., the C++ code implementing the test actions should not depend on the C++ template under test being instantiated with any specific parameters. The usage model will be used in later steps to automatically generate test cases.

A generic model for testing the NumberOps C++ class template follows.

![Figure 1: Usage Model for NumberOps Class Template](image)

The model may be written in TML, a language for defining usage models [5]. In TML, a state X is represented as [X], an arc with high level test action Y is represented as "Y", the index of the current step in a generated test case is represented by $s$, and code implementing a high level test action is prefixed with $|$. TML, among other things, allows for the inclusion of C++ code defining the implementation of high level test actions. The TML representation of the model to test the NumberOps class template is as follows.

($ assume(1) $)
In the above example C++ code implementing the high level test action associated with an arc appears on lines containing $\$. This code will be used to automatically generate test programs from the usage model.

In order to fully support testing template based software, an additional construct, the test data item, has been added to the TML model representation. A test data item $Z$ is represented by $@Z@$ in the above model. During a post processing step test data items appearing in generated test programs will be automatically expanded into C++ code based on the instantiation of the template under test. The expansion of test data items is implemented by a test action class template. Test data items are discussed in...
more detail in section 2.1.2.

2.1.2 Test Action Class Template Construction

After constructing the usage model, the test action class template for the C++ template under test must be constructed. A test action class template is itself a C++ class template that contains methods addressing two main functional areas.

- **Expanding test data items in scripts generated from the usage model.** A test action class must implement a method that takes the name of a test data item as an argument and returns the appropriate C++ code for the expansion of the test data item based on the current instantiation of the test action class template. This allows the same test action class template to be used to test many different instantiations of the C++ class template under test. The test data item processing method will be used in later steps to automatically select test data to use in test cases.

- **Implementing test actions.** A test action method is responsible for calling a method of the class template under test and checking the result of the call for correctness. Examples of calls to test action methods in a test action class are `test_avg()`, `test_min()`, and `test_max()` in the example usage model.

Support for test data items in a usage model makes it possible to test different instantiations of a C++ class template using the same generic usage model. Steps in a test case or data used in a test case that are dependent on the parameters with which the template under test is instantiated can be referred to in the usage model with test data items. The expansion of the test data items that appear in a model is implemented in a test action class template corresponding to the usage model. The expansion step is implemented in a generic manner and hence can provide different expansions of test data items based on the parameters with which the test action class is instantiated.

The expansion of test data items is performed by the `callPostProcessingMethod()` method of a test action class. The `callPostProcessingMethod()` method of the `NumberOps` test action class is as follows.

```cpp
string callPostProcessingMethod(string method_name) {
    // Randomly choose the size of the test data set.
    if (method_name == "getNumberElements") {
        num_elements =
            index_random_number.getRandomNumber(5, 0);
        return to_string(num_elements);
    }

    // Create C++ code to initialize the test
```
// data with hard coded values.
if (method_name == "initializeList") {

  string r = "";

  // Create a random list of values for
  // test data.
  init_random(list, num_elements);

  // Generate C++ code to assign these values
  // to a variable in the generated test case.
  r += to_cpp(list, " list") + "\n";

  // Also create code to precompute the
  // average for use as a test oracle.
  r += "avg_value = 0;\n";
  r += "for (size_t i=0;\n";
  r += " i<num_elements; i++) {\n";
  r += "  avg_value += list[i];\n";
  r += "}\n";
  r += "if (num_elements > 0) {\n";
  r += "  avg_value /= num_elements;\n";
  r += "}\n\n";
  return r;
}

// An unknown method was called.
cerr << "ERROR: Unknown test data item '" << method_name << "' called.\n";
exit(-1);
}

The callPostProcessingMethod() method takes the name of a test data item as an argument, and returns appropriate C++ code to replace the given test data item. For example, processing the @initializeList@ test data item involves generating a list of random numbers, generating C++ code for initializing a list of numbers to random values, and then generating C++ code for precomputing the average of this list of numbers. Because the callPostProcessingMethod() method is implemented in a generic manner in the test action class template, the list initialization code generated by the post processor to replace a @initializeList@ test data item will automatically change when generating tests to test a different instantiation of the NumberOps class template.

In the NumberOps test action class template the test action method test_avg() implements the high level test action “Compute the average” in the usage model.

  void test_avg(int step) {

test_avg() uses the NumberOps object created in a constructor test action method (curr_obj) to compute the average of the list of numbers created during the post processing step as a result of processing the @initializeList@ test data item. test_avg() then checks the average as computed by the NumberOps object against the precomputed average and reports an error if the two results differ. The CppUnit unit testing framework is used to structure the generated test cases and report errors to the user.

Note that the testing method proposed in this paper does not make any assumptions about the type of test oracle used. While the example given makes use of an oracle that explicitly calculates the expected value of a test action, this is not a general requirement. The only requirement imposed on test oracles by the framework is that a test oracle must be expressed as a C++ boolean expression in the test action class template. A general discussion of methods for creating automated test oracles appears in section 3.

2.1.3 Test Configuration File Construction

The usage model and test action class template are designed in a generic manner and will be reused when testing any instantiation of the C++ class template under test. A specific instantiation of the template under test is described with a test configuration file. There will be one test configuration file for each class template instantiation tested. Test configuration files are written in XML.

The following information is defined in a test configuration file.

- The parameters to use when instantiating the class template under test.
- The usage model to use when generating tests.
- The name of the file containing the class template to test.

An example test configuration file for testing the NumberOps class template when instantiated to hold a list of floating point numbers in a STL vector is as follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
```
Other instantiations of the NumberOps class template can be tested automatically simply by creating a new (or modifying an existing) test configuration file and generating new tests. No other effort is required.

2.1.4 Testing the C++ Class Template

After developing the usage model, test action class template, and test configuration file(s), all remaining steps in the testing process are fully automated. Given a test configuration file and the number of tests to run, the appropriate number of tests are generated from the usage model using the JUMBL usage modeling tool [6], the tests are processed to expand test data items into C++ code as needed, the tests are compiled and then run. The results of individual steps in the test case are checked automatically by the test action methods in the test action class, and pass/fail results for the generated test cases are reported to the tester. Once configured, this testing scheme makes it very easy to create and run extremely large test sets.

In section 1.2 several automatable aspects of API testing were listed. The means by which the listed aspects of automated testing are addressed by the testing framework are as follows.

- **Generation of Test Cases** - Given the number of test cases to generate and the usage model of the class template under test, the testing framework generates the requested number of test scripts from the usage model of the class template under test using the JUMBL usage modeling tool. Currently the testing framework supports generating test cases via random walks of the usage model or through
the use of a Chinese post man’s algorithm, which covers all states and arcs in the model with a minimum number of test steps. The JUMBL supports additional test case generation methods, such as generating test cases in order from most likely test case to least likely, which are not used in the automated testing framework.

- **Selection of Test Data** - Once a set of test cases have been generated, they must be augmented with concrete test data and grouped into an executable test suite. Given a set of test scripts and the appropriate test action class template, the testing framework automatically creates and runs a post processing utility to concretize and group the given test scripts. The post processing utility uses the test action class template to replace test data items with specific test data and groups the augmented test cases together into an executable test suite.

- **Execution of Large Groups of Test Cases** - After test scripts have been automatically generated from the usage model and then concretized and grouped into an executable test suite by the post processing utility, the test suite is automatically compiled and run by the testing framework. During execution, failure information is automatically logged to a test report.

- **Checking of System Responses for Correctness** - Each test action method implemented in the test action class template should include code to check the results of the test action for correctness, i.e., each test action method should implement a test oracle (3).

- **Analysis of Test Results** - Given the usage model, a record of generated test scripts, and test failure information, the testing framework can perform a statistical analysis of the testing performed (2.1.5).

### 2.1.5 Analyzing Test Results

Because all tests run against the system under test were generated from a usage model, statistical analysis can be performed on the test results to gain insight into the reliability of the system under test and the quality of testing performed [8, 4]. The statistical results presented in the following sections may be computed using the JUMBL usage modeling tool.

**Kullback Discriminant**

The Kullback discriminant is an information-theoretic measure of the similarity between a statistical model and a true distribution. In this case, the statistical model is given by the executed tests, while the “true” distribution is assumed to be the generating chain. Thus the Kullback Discriminant measures the similarity between the statistics of the test cases which have been executed, and the statistics of all test cases [9].

The Kullback discriminant is nonnegative, and zero only when the two distributions coincide. Thus as the statistics of the generated sample of tests approach the statistics of all tests, the Kullback discriminant approaches zero. The Kullback discriminant may therefore help determine when a sufficient amount of testing has been performed.
One difficulty with using the Kullback discriminant to determine when to stop testing is that its interpretation depends on the level of randomness in the usage model. For example, in a usage model with much randomness a Kullback discriminant value of 0.1 may be very small and hence indicate that testing should be stopped. However, in a model with little randomness, a Kullback discriminant of 0.1 may be large and indicate that testing should continue. For this reason a modified Kullback discriminant computation that takes the randomness of the usage model into account is used. The modified Kullback discriminant is called the relative Kullback [9]. A small relative Kullback discriminant always indicates that the testing performed closely matches the usage behavior described in the usage model, regardless of the level of randomness of the usage model.

Simple Single Use Reliability

A simple Bayesian reliability model [7] may be used to estimate the reliability of the system under test. This model has the useful properties of being defined when testing has revealed no failures, of having a defined variance, and of being able to make use of pre-test information about the reliability of the software under test.

The simple single use reliability \( r \) is estimated as

\[
    r = \frac{s + b}{f + s + a + b}
\]

where \( s \) is the number of successfully executed test cases, \( f \) is the number of failed test cases, \( b \) is an estimate of positive pre-test reliability information (interpretable as the number of successful tests “executed” prior to the start of testing), and \( a \) is an estimate of negative pre-test reliability information (interpretable as the number of failed tests “executed” prior to the start of testing). As the amount of testing performed increases, the pre-test reliability information (\( a \) and \( b \)) is washed out by the testing experience, in which case the simple single use reliability converges to the standard estimator of

\[
    r = \frac{s}{f + s}.
\]

The simple single use reliability is interpreted as the probability that a use generated randomly from the usage model will not fail. A use is considered to have failed if any of the steps in the use fail.

The variance associated with the simple single use reliability estimate is

\[
    var(r) = r\left(\frac{1}{f + a + b}\right) / \left(\frac{f + s + a + b}{f + s + a + b + 1}\right).
\]

A small variance indicates that the estimated reliability is accurate and that further testing is unlikely to provide any new information about the reliability of the system.

Single Event Reliability

In contrast to [7], the single event reliability provides an estimate that a randomly selected test step represented in the usage model will succeed when executed in a test. Note that it is possible for testing to result in a high single event reliability and a low overall system reliability. This occurs when the expected length of a test case generated is very long, which provides many opportunities for a test step to fail in a typical use of the software.
3 Test Oracles

A test oracle is function $O : H \times R \rightarrow \{true, false\}$, where $H$ is the set of possible test sequences and $R$ is the set of generated system responses. $O(x, y) = false$ means that $y$ is not the correct response to test sequence $x$. $O(x, y) = true$ means that the response to test sequence $x$ has not been judged to be incorrect. Note that this definition of a test oracle does not require the oracle to always correctly identify failed test sequences. It only requires that a test sequence identified as failed will actually have failed, i.e., there will be no false positives when identifying failed test sequences.

The implementation of an automated test oracle requires the following.

1. As a test case is executed, relevant testing history must be stored by the automated test oracle system. Stored information about the executed test sequence will be used by an automated oracle to compute $O(x, y)$ for a given system response $y$. The testing history stored by the oracle corresponds to $x$, an element in set $H$.

2. The response of the system under test must be observed by an automated testing framework and given to the automated oracle. If one or more system responses cannot be observed and collected automatically, it will not be possible to construct a fully automated oracle for the system under test. The observed system response corresponds to an element in set $R$.

3. The automated test oracle must have some means of computing expected responses. There are several general methods that may be used to compute expected responses. These methods are discussed in detail in the following sections.

4. The automated oracle must have some means of communicating its verdict back to the testing framework.

Task three, computing the expected response to a test sequence, is the greatest challenge. There are two typical situations that occur when computing expected system responses in a test oracle. First, it may be the case there is a single expected response to a test sequence $H$ and that there is a relatively straightforward means of determining the expected response. This situation can be handled by a function-based test oracle (section 3.1). Second, it may be the case that there is a single expected response to a test sequence $H$ but there is no straightforward or practical method for determining the expected response. This situation may best be handled by a relation-based test oracle (section 3.2).

Note that different general functions of the system under test will be exercised by different test sequences. The behavior of different system functions may best be checked by different types of test oracles, i.e., all system functions need not be checked by the same kind of test oracle.
3.1 Function-Based Oracle

A function-based test oracle implements a function $E : H \rightarrow R'$ for computing expected system responses, where $H$ is the set of possible test sequences and $R'$ is the set of expected system responses. It is not required that $R = R'$, i.e., the set of responses that the system can generate does not have to be equal to the set of expected system responses. For all $h \in H$, $E(h)$ is the correct response to test sequence $h$. The test oracle function $O : H \times R \rightarrow \{\text{true}, \text{false}\}$ is then defined such that

$$\forall h \in H \land \forall r \in R, O(h, r) = \begin{cases} 
\text{true} & \iff E(h) = r \\
\text{false} & \iff E(h) \neq r 
\end{cases}.$$  

Because $E$ maps a test sequence to the single correct response, a function-based oracle will never declare an incorrect response to be correct.

The implementation of $E$ does not share all of the non-functional requirements of the system under test. The primary non-functional requirement of $E$ is that it must be verifiably correct. Non-functional requirements of the system under test, such as limits on execution speed, memory usage, and disk usage can be disregarded to a large degree so as to implement $E$ in as simple and correct a manner as possible. The methods of implementing $E$ address the main requirement of $E$, correctness and simplicity, in two different ways.

3.1.1 Duplicate Implementation in a High Level Language

It is often the case that a highly optimized function written in a language such as C or C++ can be written in a much simpler (though less efficient) manner in a high level language such as Matlab or Python. Because the efficiency of the implementation of $E$ is secondary to its understandability and correctness, inefficiencies due to algorithm choice, data representation choice, and language choice can be tolerated if the choices lead to a simple and verifiably correct $E$ implementation.

For example, when testing a distributed matrix inversion routine it is possible to implement $E$ by using the built-in Matlab inversion function to compute the inverse of the matrix. In this case $E$ would not be as efficient as the distributed matrix inversion routine under test, but $E$ would be simple and correct, assuming Matlab works correctly.

The automated testing framework includes support for converting data structures to and from C++, Python, and Scilab and for generating and running such programs during test execution.

3.1.2 Use of Existing Alternate Implementation

If an alternate implementation of the system function of interest is available, it may be used as a test oracle. In this case appropriate information from $H$ is passed to the alternate implementation, which computes and returns the result back to the test oracle. Alternate implementations might be taken from legacy systems, competitor’s systems, or alternate implementations within the system under test.

In the project described in 4, the system under test includes function objects used to calculate demagnetizing fields due to dipole-dipole interactions within magnetic materials. The system under test currently supports two implementations of the dipole-
dipole interaction policy, one which is fast and efficient for large data sets, and one which is simple but slow for large data sets. The simple implementation was used to implement $E$ for testing the efficient dipole-dipole interaction policy. A disagreement in the values computed by the two dipole-dipole interaction policy implementations indicates a possible failure in the system under test.

### 3.2 Relation-Based Oracle

When $E$ is a relation on $H \times R$, the test oracle function is defined such that $\forall h \in H \land \forall r \in R, V_h = \{x|(h, x) \in E\} \land O(h, r) = \begin{cases} \text{true} & r \in V_h \\ \text{false} & r \notin V_h \end{cases}$, where $V_h$ is the set of valid, i.e., not obviously incorrect, responses to a given test sequence $h$.

Implementation of a relation-based oracle involves determining whether a given response $r$ appears in the set of valid responses $V_h$ to test sequence $h$. Set $V_h$ has the property that if $r$ is not in $V_h$, the response is incorrect and test sequence $h$ has failed. Note that it is possible for $r$ to be in $V_h$ and still be incorrect. Rather than explicitly enumerating $V_h$, it is often easier to identify a set of characteristics that differentiate the elements in $V_h$ from other system responses and then check to see if a given response $r$ possesses these characteristics.

A relation-based test oracle may be used when the identification of a single expected response to $h$ is impractical. The impracticality may be due to:

- **Time constraints** - The computation of the expected response may take a very long time.
- **Effort Constraints** - The effort required to implement software to determine the expected response may be too great.

An example of a relation-based oracle for a positive square root function $f(x) = \sqrt{x}$ can be defined given $V_h$ defined as $V(x) = \{y|(y \geq 0) \land (y \leq x) \land ((y + 1)^2 > x)\}$. The oracle for the square root function can check the correctness of system responses without requiring an alternate implementation of the square root function.

### 4 Case Study

The $\Psi$-Mag toolkit is a large C++ template based scientific computing library written by the Materials Research Institute at the Oak Ridge National Laboratory (ORNL). It is being tested with the described framework. The $\Psi$-Mag toolkit is used to create simulations of magnetic materials, spin-systems, and statistical physics. To support these simulations the $\Psi$-Mag toolkit provides functionality supporting configurable Monte Carlo methods, the construction of Hamiltonian equations used in energy and field calculations, histogramming, ordinary differential equation solvers, and the representation of spin systems. The $\Psi$-Mag toolkit is generic, making it easy to apply the toolkit to different problems and easy to configure and extend.

Since the toolkit will be used for leading edge research in computational materials by scientists worldwide and on a variety of computers, including the leadership class
computers at ORNL, this automated testing project was established. The project has the following goals.

1. Testing that is independent of the developers.
2. A published testing protocol and record of testing.
3. Ability to conduct and characterize an adequate kind and amount of testing.
4. Ability to respond quickly with solutions to field reported failures, including re-testing.
5. Ability to quickly retest routines that are modified.
6. A framework and protocol for testing new additions to the library.
7. To evolve, through experience, a set of statistics that will help to define stopping criteria for adequate testing.

4.1 General Test Results

The general results of $\Psi$-Mag testing to date are as follows.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests Run</td>
<td>~12K (~1200K Test Steps)</td>
</tr>
<tr>
<td>Run-Time Failures Found</td>
<td>25</td>
</tr>
<tr>
<td>Compile-Time Failures Found</td>
<td>6</td>
</tr>
<tr>
<td>Class Templates Tested</td>
<td>17 (~4 KSLOC)</td>
</tr>
</tbody>
</table>

The size of the test action class templates used in testing has been comparable to the size of the manually created test cases used in development testing. However, the templates used in the automated testing framework are applicable to testing many different instantiations of a class template, whereas the manually created test cases only test a small, fixed set of instantiations. Also, it is very easy to create and run many tests using the test action class templates and the automated testing framework.

A compile-time error occurs when an instantiation of a C++ template fails to compile when instantiated with template parameters that should be valid based on the specification of the template. Hence the discovery of these failures relies on being able to test many different instantiations of the template. Because of the ease of testing different class template instantiations with the automated framework, several compile-time failures were discovered through automated testing that were not revealed by developer testing.
4.2 Statistical Analysis of Test Results

The main test statistics used in analyzing the test records of the Ψ-Mag toolkit are the relative Kullback discriminant, the simple single use reliability estimate and variance, and the single event reliability estimate and variance. The test results thus far are shown in table 4.2. (Note that the simple single use reliability was computed using no pre-test reliability information, i.e., \( a = b = 1 \).)

**Table 2: Analysis of Test Results**

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Tests Run</td>
<td>100</td>
<td>266</td>
<td>1000</td>
<td>68889</td>
</tr>
<tr>
<td>Relative Kullback Discriminant</td>
<td>0.00199</td>
<td>0.147</td>
<td>0.782</td>
<td>0.0254</td>
</tr>
<tr>
<td>Simple Single Use Reliability</td>
<td>0.637</td>
<td>0.889</td>
<td>0.999</td>
<td>0.0165</td>
</tr>
<tr>
<td>Single Event Reliability</td>
<td>0.956</td>
<td>0.982</td>
<td>0.999</td>
<td>0.000229</td>
</tr>
</tbody>
</table>

In most cases 500 or more tests were run. However, tests of the Monte Carlo simulation class templates took significantly more time to run than tests of other class templates, which resulted in only 100 tests being run against various instantiations.

The relative Kullback discriminant reveals that in all cases the testing experience very closely matches use as represented by the usage model. The small size of the relative Kullback discriminant suggests that it may have been possible to rigorously test the class templates with fewer test cases. However, using the relative Kullback discriminant as a test stopping criterion must also take into account any test case information that is not explicitly represented in the usage model. For example, in testing the Ψ-Mag toolkit the initialization of data used in the creation and modification of objects under test is performed in a post-processing step and is not explicitly represented in the usage model. Because of this it is possible for the relative Kullback discriminant to indicate that testing closely matches expected use but still not adequately test a class template with various data values. To ensure that the Ψ-Mag toolkit was adequately tested, testing was performed well past the point at which the relative Kullback discriminant ceased to show significant change. We believe that this situation can be improved with further research.

The single use reliabilities varied significantly. This is because some class templates exhibited no failures under test, while others had multiple failures. In all cases sufficient testing was performed to achieve small variances in the estimated simple single use reliability. The small associated variances indicate that sufficient testing was performed to accurately estimate the reliability of the class templates under test.

In contrast to the simple single use reliabilities, the single event reliabilities showed less variation across tested class templates. The difference underscores the importance of taking the expected use of the software into account when testing. A change in the represented use of the software under test that shortens the length of typical uses could result in noticeable increases in the estimated reliability of the software. Thus, the characterization of expected use in the field is important.
5 Method Summary and Conclusion

In summary, C++ class templates can be automatically tested with the presented framework. The basic work to be performed to use the framework is as follows.

1. Construct a usage model for the class template to be tested. This usage model is augmented with C++ code to implement test cases generated from the model.

2. Make a test action class template implementing high level test actions annotated on the arcs of the usage model and implementing transformations of test data items to C++ code. The test action class template is written to be applicable to testing many different instantiations of the class template under test.

3. Create a test configuration file for each instantiation of the class template to be tested.

4. Automatically generate, execute, and check an initial set of tests.

5. Perform statistical analysis of the test results.

6. If the test statistics indicate the class template has been thoroughly tested, stop testing. Otherwise return to step 4.

The size of the work products created to test class templates with the automated testing framework is comparable to the size of manually created specialized test suites created prior to this testing project. Several errors not detected by manual testing have been detected during automated testing. The availability of statistical analyses of the testing performed greatly aids in evaluating the software under test and the performed testing. Usage model-based automated testing has proven to be a practical, effective means of testing a C++ template based scientific computing library. Work will continue on improving the framework and testing the Ψ-Mag toolkit.

References


