

Automated Test Coverage

Take Your Test Suites To the Next Level

PSQT/PSTT 2004 East

Jay Berkenbilt

Vice President, Software and Technology

Apex CoVantage

<http://www.apexcovantage.com>

Email: ejb@ql.org

1 Abstract

This paper discusses the advantages of adding automated test coverage checks to existing automated testing frameworks and presents a specific approach for doing so. The approach discussed here is lightweight and can be implemented in nearly any programming language. The paper includes a small sample function that includes test coverage checks. There is also a reference to a self-contained downloadable example that serves as a sample implementation of the techniques presented. The reader is assumed to have a basic knowledge of programming and automated software testing.

2 Introduction

All software test enthusiasts know the value and importance of thorough automated test suites. By requiring every piece of code to have an automated test suite, we enforce a *designing for testability* mentality among our developers and produce software whose quality is much higher than the norm. To test software thoroughly, it is necessary to build test suites that include not only *black box* tests, but also *white box*, or more aptly named *clear box* tests that are carefully crafted to exercise the code in particular ways. By ensuring that our test suites fully exercise our code, we can produce code that is as close to bug free as any code ever comes.

Even the most thorough black box/clear box test suite may be subject to a few weaknesses. Consider the following scenarios:

- If the last test case were removed from any passing test suite, all prior cases would still pass. How would you know whether any functionality exercised by the last test case was still being exercised by the test suite?
- Suppose a particular test case were crafted to exercise a specific boundary condition, but internal parameters were changed in a way that moved the boundary. The test case may still generate the expected results and therefore still pass. How would you know whether the boundary condition is still being tested?

- If a programmer who is maintaining a piece of software modifies a test file, how can you be sure the modification hasn't undermined an existing test case, making it no longer exercise the code it was designed to test?
- If you are reviewing someone's code, how can you easily guarantee that a particular condition is and will always be exercised by the test suite?

The automated test coverage analysis approach described here addresses these and other issues by creating a tight coupling between the test suite and the conditions in the code that the test suite is supposed to exercise. Specifically, it introduces a mechanism by which the overall test suite will fail unless all tested scenarios are exercised, even if every individual test case in the test suite passes. This is achieved with minimal impact on existing test suites and minor impact on the code being tested, and it works in a wide range of languages and platforms without the need for expensive third-party tools.

The techniques discussed here are programming and testing techniques: they require explicit implementation by software developers while a system is being coded. These techniques are distinct from (and complementary to) the use of “black-box” tools that analyze running software for code coverage in the traditional sense; *i.e.*, which “lines of code” are covered during a specific collection of program executions.

3 Basic Testing Philosophy

There are a variety of approaches to automated software testing, each of which has its own advantages and disadvantages. The author's preferred approach is *designing for testability* and subsequent creation of what might be called *self-testing software*. When using this approach, deciding how a specific area of functionality will be tested is central to the design. Every requirement and every piece of code must be testable, and the way in which it will be tested must be planned before any implementation is done.

Applications or modules implemented in this fashion will often have an embedded test mode where the application itself or an external driver that “lives” with the software and is closely linked with it may exercise the code in a preset way, comparing the software's output with an established baseline. The software may contain code whose sole purpose is to assist with the testing process. This type of testing is performed largely by the developer. Test suites are coded incrementally at the same time as the software is being written. As defects in the software are found and corrected, the test suite is modified to exercise the defect before the problem is fixed, thus ensuring that the problem *stays* fixed. After all, the fact that the test suite did not enable the bug to be caught in the first place can well be considered a bug in the test suite!

This form of testing is complementary to the more traditional software testing that may be performed manually by a testing team or automatically using external tools to play back sequences of prerecorded events. It is also complementary to the normal Alpha and Beta testing cycles that should precede most software releases. All of these types of testing should be used together to ensure maximum software quality.

4 Basic Test Suite Functionality

When we say that a test suite passes, we must be very clear about exactly what this means. A good test suite must have the property that an overall “pass” is a very strong indicator of correctly working software. First, we will limit ourselves to discussion of traditional test suites without the addition of the test coverage features discussed in this paper.

Test suite passage must be defined in such a way that any unexpected error condition in the application or in the test suite itself will cause the test suite to fail. We define a test suite as passing when *all* of the following conditions are met:

- Each individual test case passes.
- The expected number of test cases is run.
- The test driver itself terminates normally.

An individual test case is considered to pass when its output and exit status match those expected by the test suite.

Notice that the conditions described above are stronger than just saying that the test suite passes when all individual test cases pass. If there is a logic error in the test suite itself that causes a test case to be skipped entirely, this should make the test suite fail. Likewise, if the test driver crashes part way through but all tests up to that point have passed, the test suite should fail. By requiring the test driver to terminate normally after executing the correct number of test cases, we add some extra security to the test suite.

The test coverage system adds still further security to the test suite by ensuring not just that we have seen the correct number of test cases, but also that they have actually exercised the code as the developer intended.

5 Test Coverage System Overview

We define the following terms as they are used in this document:

- Coverage Case: any “event” within the software that we wish to detect using the automated test coverage system
- Coverage Scope: a way of assigning coverage cases to a specific test suite, allowing multiple test suites to exercise different coverage conditions within one body of code

The automated test coverage system consists of three components:

- Coverage Calls: function calls to the coverage system made from inside the code being tested to tell the system that a specific coverage case has occurred
- Coverage Case Registry: a file that lists all the coverage cases that belong to a given scope
- Coverage Analyzer: the program that compares the coverage calls and their results to the coverage case registry

6 A Quick Example

Before discussing the details of the coverage system, we present a minimal coverage example. Here is a simple C++ code fragment that tests a condition and takes one of two actions:

```
if (condition)
{
    do_something(vall);
}
else
{
    do_something_else();
}
```

Any test coverage system could ensure that both halves of this branch were executed in a given test suite. Suppose, however, that we wanted not only to ensure that both branches had been executed, but that the first branch had been executed at least once with the value of `vall` greater than 5 and at least once with the value of `vall` not greater than 5. Using our coverage system, we could add coverage calls to encapsulate these conditions. The resulting code fragment would then look like this:

```
if (condition)
{
    TCOV::TC("example", "something", (vall > 5 ? 0 : 1));
    do_something(vall);
}
else
{
    TCOV::TC("example", "something else");
    do_something_else();
}
```

The call to the function `TCOV::TC()` is an example of a coverage call. We will now explore coverage calls and the rest of the system in depth.

7 Test Coverage System Details

Here we describe each component of the test coverage system in detail and discuss how to integrate the coverage system into an existing test framework.

7.1 Coverage Calls

Each individual coverage case appears in the code as a call to the function `TCOV::TC()`. This function takes three parameters: a string literal containing the name of the scope, a string literal containing the name of the coverage case, and an optional numeric argument, which may be an expression. The numeric argument is used to distinguish different circumstances under which a specific coverage call may be made. In our first example above:

```
TCOV::TC("example", "something", (vall > 5 ? 0 : 1));
```

the scope is `example`, the coverage case name is `something`, and the numeric argument is dependent upon the value of the variable `val1`. This enables the coverage system to distinguish between whether this part of the code was executed with the value of `val1` greater than 5 or not and to require that both cases occur in the test suite.

In this scheme, the code being tested is sprinkled with test coverage calls such as the one above. These calls serve both as the interface from the code into the test coverage system and as a promise to the reader that the code in question is exercised by the test suite. In effect, this enables the code to test the test suite in addition to having the test suite test the code. It also enables the developer to add coverage calls to the code while it's being implemented. This is when the list of boundary conditions, confusing corner cases, etc., is as fresh in the programmer's mind as it ever will be. The programmer could just keep a running list of ideas for the test suite, but by coding a coverage case, the programmer *guarantees* that the code will be exercised in the test suite in exactly the intended way.

Although addition of coverage cases is intended to be done by the programmer, this functionality can also be very useful for the reviewer. When reviewing someone's code, insertion of a test coverage case into the code is a great way to determine whether the test suite is exercising the code in a particular way. It also ensures that if it is not, this missing functionality will be added to the test suite before it will pass again.

7.2 Coverage Case Registry

All coverage cases have to be *registered* in the coverage case registry. The coverage case registry lists all the coverage cases that are expected to appear in a specific coverage scope along with the maximum expected value of the numeric argument passed as the third argument to the coverage call. In our sample implementation, the coverage case registry is an ASCII file named `scope.testcov`, where `scope` is the scope whose cases it lists.

Exactly one coverage call must appear in the code for every coverage case in a given scope. That call must be made at least once with each numeric value from zero up to and including the number indicated in the coverage case registry. In the above example, the numeric argument to the `something` coverage case in the `example` scope may have the value of either 0 or 1 under normal conditions. Therefore, the file `example.testcov` would contain the following line:

```
something 1
```

This indicates that this coverage call must be called at least once with the value 0 and at least once with the value 1.

In addition to the list of coverage cases in this scope, the registry file contains a list of other scopes to allow but ignore in the code. Any other scope in a coverage call generates an error. This allows multiple scopes to be present in a collection of source files while still ensuring that coverage cases won't be overlooked as a result of a typographical error in the scope name. If the example above also contained coverage calls for a scope called `other`, we would expect to see this line:

```
ignored-scope: other
```

in the coverage registry file as well. This tells the system that it may safely ignore coverage calls in the `other` scope.

7.3 Coverage Analyzer

The remaining component of the coverage system is the coverage analyzer. The coverage analyzer is a short program that is run once at the beginning and once at the end of the test suite. At the beginning of the test suite, the coverage analyzer performs the following tasks:

- Ensure that every coverage case listed in the registry is unique.
- Ensure that every coverage call in the source code either belongs to the current scope and appears in the registry or that it belongs to one of the explicitly ignored scopes.
- Ensure that every coverage case in the current scope appears exactly one time in the code.

If all of these checks succeed, the test coverage system is activated. Otherwise, the test suite is aborted.

In our sample implementation, we impose lexical constraints upon coverage calls so that the analyzer can find them without having to have any ability to parse the programming languages that may be in use. We impose the following constraints:

- The names of the scope and coverage case must be string literals and must appear on the same line as the name of the coverage function.
- The name of the coverage function may be preceded on its line by whitespace only.

No restrictions are placed on the numeric argument. It may appear on the same line or a different line, and it may be a numeric literal or any expression with a numeric value.

At the end of the test suite, the coverage analyzer ensures that every coverage call has been called at least once with each required numeric argument. It then reports a list of all missing and extra coverage cases. The coverage analyzer's checks are cumulative throughout the entire test suite run.

7.4 Coverage Call Implementation

To understand how this works, it is necessary to know exactly what the coverage call actually does. In our sample implementation, the coverage call determines whether coverage is active in the scope passed as its first argument. If so, then it appends the name of the coverage case and the numeric value to an output file. The names of the scope and output file are read from environment variables. These environment variables will have been set by the test framework based on the results of the first run of the

coverage analyzer. At the end of the test suite, the coverage analyzer will read this file to find out which coverage cases were seen.

Here are sample C++ and perl implementations of the test coverage call:

```
// C++ implementation

// Determine whether coverage is active in the current scope
static bool tc_active(char const* const scope)
{
    std::string value;
    // The return value of get_env indicates whether an environment
    // variable is set. If the second argument is non-null, it is
    // initialized with the value of the environment variable.
    return (get_env("TC_SCOPE", &value) && (value == scope));
}

void TCOV::TC(char const* const scope,
              char const* const ccase, int n /*= 0*/)
{
    // Return immediately if coverage is not active for this scope.
    if (! tc_active(scope))
    {
        return;
    }

    // Get the name of the output file.
    std::string filename;
    if (! get_env("TC_FILENAME", &filename))
    {
        return;
    }

    // Append data from this call to the output file.
    FILE* tc = fopen(filename.c_str(), "ab");
    if (tc)
    {
        fprintf(tc, "%s %d\n", ccase, n);
        fclose(tc);
    }
}

# Perl implementation

package TCOV;

sub TC
{
    my ($scope, $case, $n) = @_;
    local $!;
    $n = 0 unless defined $n;
    return unless ($scope eq ($ENV{'TC_SCOPE'} || ""));
    my $filename = $ENV{'TC_FILENAME'} || return;
    my $fh = new FileHandle(">>$filename") or return;
    binmode $fh;
    print $fh "$case $n\n";
    $fh->close();
}
}
```

In this implementation, we activate the test coverage system by setting the environment variable `TC_SCOPE` to the coverage scope and `TC_FILENAME` to the absolute path of the file that the coverage system will append to.

7.5 Integration of Test Coverage into Existing Test Frameworks

Integrating the coverage system into an existing test suite is not difficult at all. All that is required is adding some mechanism to ensure that the appropriate environment

variables are set and that the analyzer is invoked once at the beginning and once at the end of the test suite. We also must add an additional condition for test suite passage:

- The test coverage analysis showed full coverage of all registered coverage cases with no missing or extra coverage cases.

Our sample implementation also appends identifying information to the coverage output file before each test case and makes a backup copy of the test coverage output file each time the overall test suite passes. This makes it easier to figure out after a coverage failure which test case previously exercised the specific coverage condition that is no longer occurring.

8 Example Code: search

Here we present a simple example application that uses our test coverage system. This is a program that finds a number in a sorted array using either a straight linear scan or a binary search, depending upon the size of the array. This is a toy program for purposes of illustrating the test coverage system, but it has enough interesting cases to be illustrative. Here is the code without any coverage calls:

```
#include <vector>

static unsigned int const SCAN_THRESHOLD = 10;

// Find an item by doing a linear scan
int scan(std::vector<int> const& v, int to_find)
{
    int nitems = v.size();
    for (int i = 0; i < nitems; ++i)
    {
        if (v[i] == to_find)
        {
            return i;
        }
    }
    return -1;
}

// Find the item with a binary search
int bsearch(std::vector<int> const& v, int to_find)
{
    int low = 0;
    int high = v.size() - 1;
    while (low <= high)
    {
        int test = (low + high) / 2;
        if (v[test] == to_find)
        {
            return test;
        }
        else if (v[test] < to_find)
        {
            low = test + 1;
        }
        else
        {
            high = test - 1;
        }
    }
    return -1;
}

// Call one of scan() or bsearch() based on the size of the vector
int search(std::vector<int> const& v, int to_find)
{
    if (v.size() <= SCAN_THRESHOLD)
```

Automated Test Coverage: Take Your Test Suites To the Next Level

```
{
    return scan(v, to_find);
}
else
{
    return bsearch(v, to_find);
}
}
```

The function's implementation is quite straightforward: if the list is smaller than a certain threshold, do a linear scan; otherwise, do a binary search. The binary search algorithm is just a classic textbook implementation of binary search. We don't handle cases like detecting that the array isn't sorted, handling duplicate items, etc. This code is really to demonstrate test coverage rather than to talk about searching algorithms!

Although this code is simple and straightforward, there are a lot of conditions that a good test suite should exercise. For example:

- Did both types of search get exercised both for numbers that were found and numbers that were not found?
- Did we have numbers that were not found because they were too low, too high, and in the right range but just not there?
- Did all three branches of the binary search code get exercised (increasing the lower bound, decreasing the upper bound, and finding the value)?

This code also presents a good example of how a test case may later no longer test the condition for which it was originally designed. For example, consider the `SCAN_THRESHOLD` variable. If we make it too low or too high, we may find that cases designed to exercise a condition in one type of search suddenly find themselves getting processed with the other type of search. What we really need is some assurance that the test suite exercises the conditions in the code that we want it to. The test coverage system allows us to make these assurances into a permanent part of the test suite.

Here is the code again. This time, we're also including a small `main()` routine to test the `search()` function as well as several coverage calls. The main routine just creates an array of consecutive multiples of 3. In a real application, we might want to test with other arrays, but again, our purpose here is only to illustrate the test coverage system.

```
01 #include <iostream>
02 #include <stdlib.h>
03 #include <vector>
04 #include "TCOV.hh"

05 // If you change this threshold to something greater than 15 or less
06 // than 4, the test suite passes but coverage fails.
07 static unsigned int const SCAN_THRESHOLD = 10;

08 // Find an item by doing a linear scan
09 int scan(std::vector<int> const& v, int to_find)
10 {
11     int nitems = v.size();
12     for (int i = 0; i < nitems; ++i)
13     {
14         if (v[i] == to_find)
15         {
16             TCOV::TC("search", "linear scan found");
17             return i;
18         }
19     }
20     TCOV::TC("search", "linear scan not found",
21             ((to_find < v[0]) ? 0 : // too low
```

Automated Test Coverage: Take Your Test Suites To the Next Level

```

22         (to_find > v.back()) ? 1 :      // too high
23         2));                          // just not there
24     return -1;
25 }

26 // Find the item with a binary search
27 int bsearch(std::vector<int> const& v, int to_find)
28 {
29     int low = 0;
30     int high = v.size() - 1;
31     while (low <= high)
32     {
33         int test = (low + high) / 2;
34         if (v[test] == to_find)
35         {
36             TCOV::TC("search", "bsearch found");
37             return test;
38         }
39         else if (v[test] < to_find)
40         {
41             TCOV::TC("search", "bsearch increased low");
42             low = test + 1;
43         }
44         else
45         {
46             TCOV::TC("search", "bsearch decreased high");
47             high = test - 1;
48         }
49     }
50     TCOV::TC("search", "bsearch not found",
51             ((to_find < v[0]) ? 0 :      // too low
52             (to_find > v.back()) ? 1 :  // too high
53             2));                          // just not there
54     return -1;
55 }

56 // Call one of scan() or bsearch() based on the size of the vector
57 int search(std::vector<int> const& v, int to_find)
58 {
59     if (v.size() <= SCAN_THRESHOLD)
60     {
61         return scan(v, to_find);
62     }
63     else
64     {
65         return bsearch(v, to_find);
66     }
67 }

68 int main(int argc, char* argv[])
69 {
70     if (argc != 3)
71     {
72         std::cerr << "Usage: search n-items to-find" << std::endl;
73         exit(2);
74     }
75     int nitems = atoi(argv[1]);
76     if (nitems < 1)
77     {
78         TCOV::TC("search", "nitems < 1");
79         std::cerr << "search: n-items must be >= 1" << std::endl;
80         exit(2);
81     }
82     int to_find = atoi(argv[2]);
83     std::vector<int> v;
84     for (int i = 0; i < nitems; ++i)
85     {
86         v.push_back(i * 3);
87     }
88     int idx = search(v, to_find);
89     TCOV::TC("search", "idx location",
90             ((idx == -1) ? 0 :          // not found
91             (idx == 0) ? 1 :          // first item
92             (idx < nitems - 1) ? 2 :  // somewhere in the middle
93             (idx == nitems - 1) ? 3 : // last item
94             4));                          // can't happen
95     std::cout << "index: " << idx << std::endl;
96     return 0;
}

```

Observe that we have added eight test coverage calls, shown above in boldface type. These calls are all in the `search` scope. Here is the `search.testcov` file that would accompany this code:

```
linear scan found 0  
linear scan not found 2  
bsearch found 0  
bsearch increased low 0  
bsearch decreased high 0  
bsearch not found 2  
idx location 3  
nitems < 1 0
```

Now we'll look at some specific coverage calls to study what they do. Of the eight calls, five of them (lines 16, 36, 41, 46, and 78) are simple coverage calls with the numeric argument omitted. These calls are just inserted into the normal flow of the code to tell the system that the part of the code that contains them has been executed. You can see that this type of coverage call has been placed at each significant branch in the code as well as for the one error condition that we want to be sure to exercise in the test suite.

The other three coverage cases make use of the numeric argument to the coverage call. There is a coverage case at the end of each of the two lower-level search functions (lines 20–23 and 50–53) that gets invoked when the search function fails to find the number in the array. These calls both distinguish between the case of the value being less than the first item, greater than the last item, or otherwise. This ensures that the search routines are both tested with numbers that fall out of range in each direction as well as with in-range numbers that are not in the array. These are important boundary conditions. The last coverage call in the main routine (lines 89–94) also uses the numeric argument. In this case, we're making sure that other important boundary conditions are tested including finding the first and last numbers in the array. Notice that this coverage call may potentially pass the value 4 as the numeric argument (line 94), but the coverage case is registered with a maximum numeric argument of 3. This illustrates a useful technique: use numbers greater than the registered maximum for cases that can't happen. If that case were ever to happen, it would result in the test coverage system reporting `idx location 4` as an *extra* coverage case. Just as it's often better to have an explicit `else` clause when coding conditionals, it's often better to use this type of technique when coding expressions to be used as the numeric argument to a coverage call. This is safer than just assuming you haven't missed any possibilities.

This code, along with its accompanying test suite and a working implementation of this coverage system is available for download as discussed at the end of the paper.

9 Performance and Security

Notice that some of the test coverage cases here are called inside of a loop. Although coverage calls have a nominal performance impact when the coverage system is inactive (they just check an environment variable and return), in critical or time-sensitive code, even this function call overhead could be undesirable. In some cases, there may be security implications of having the coverage calls accessible by the end user—a user who knew which environment variables to set could cause the application to append data to any file the application had write access to.

In cases such as these, it would be possible to use conditional compilation to omit the test coverage calls for a releasable build of the software. If this is done, it is especially important that the coverage calls have no side effects (a good idea in any case). To be sure, test suites should be designed so that they can still be run on releasable versions of the code with coverage analysis turned off.

10 Real-World Examples

In this section, we discuss three examples of actual uses of the test coverage system to detect and correct faults in test suites.

10.1 Unprintable Characters

A perl program was generating XML from the output of an OCR program. The OCR program sometimes put unprintable characters in its output. This perl program had a line of code to replace unprintable characters with the “?” character.

One of the input files was constructed to have several unprintable characters. A coverage case ensured that the perl substitution operator returned true (indicating that it replaced some characters) at least once in the test suite by using the return value of the substitution operator in the numeric argument to the coverage call.

During an update to the software, a junior programmer edited that test file to exercise some condition unrelated to the unprintable characters, and in doing so, removed the line that contained the unprintable characters. He made the corresponding edit in the expected output file, removing the line of output that contained the “?” characters.

Although that test case passed, the character stripping code was no longer exercised in the test suite. This caused a failure to be reported by the test coverage system, thus alerting the developer that his change had undermined the original purpose of the test case. The test case was redesigned so that both the new and old conditions could be tested.

10.2 Journal Issue With No Articles

A program was generating XML for some but not all of the articles in a journal issue. Certain articles would be excluded based on a series of rules. Sometimes an issue would have no articles to be generated because all of the articles met the exclusion conditions. A special case in the code existed to test this valid situation, and a coverage case ensured that this happened in the test suite. Only one issue in the test suite exercised this condition.

Six months later, the exclusion rules were changed. As it happened, one of the articles in the special test issue now no longer met the exclusion conditions. The test coverage system alerted us to the fact that the special “all articles excluded” case was no longer being exercised, thus enabling us to remove that article from the test issue. The coverage system further ensured that removing that article didn't undermine any other test conditions.

10.3 Waiting for Worker Thread

A multithreaded image manipulation program had a worker thread that calculated information that was not needed to draw the screen but that was needed to do certain specific manipulations on the image. Upon performing the first operation that required the results of the calculation, the main thread would block waiting for the worker thread to finish. This case was exercised in the test suite by having an input image for which the worker thread's calculations took a long time. A coverage case was associated with code that got executed only when the worker thread was still running when the main thread first needed the information.

A year later, this code was tested on a much faster machine. The machine was fast enough that the worker thread finished its calculations on the test image before the main thread needed them. All test cases passed, but the coverage system alerted us to the fact that the thread waiting code was no longer being exercised. The code was then updated to force an artificial delay under specified conditions triggered by the test suite (such as a special environment variable or command-line flag) thus guaranteeing that the condition would be tested on a machine of any speed. This turned out to be particularly important because the faster machine was using a newer implementation of the underlying threading system with which this application had not been previously tested.

11 Test Coverage Tips and Tricks

Here are a few tips and tricks that can be used with the test coverage system to further increase its usefulness.

Pick names for coverage cases that start with the name of the source file in which they appear. This way, the lexically sorted listing of missing or extra coverage cases reported by the coverage analyzer will be grouped together by source file.

Agree on naming conventions for certain kinds of cases. For example, put the word `ERR` after the source file name in all coverage cases that are associated with issuing error messages. This causes all coverage cases that should happen in error conditions to be grouped together within the cases for a given source file.

Add coverage calls to the code while it is being written—that's when you're best able to think of all the detailed clear-box test cases you couldn't have thought of during design. This makes it impossible to forget to exercise certain functionality in the test suite.

Make coverage calls themselves conditional upon the circumstances under which you want to ensure that the event occurs, even if this information has to be passed in explicitly by the test driver. For example, the test suite for a configuration file parser may want to make sure that a particular construct appears in a file that contains no errors. It may not be sufficient to simply avoid making the coverage call if no errors have yet been detected because an error may still appear in the configuration file *after* the construct has already been seen. Although the application may have no way to know that, the test suite knows in advance whether a specific run is supposed to have errors or not. It can pass this information to the program being tested through an environment variable or

command-line argument. The code being tested can make the coverage call conditional upon that information instead. In other words, don't be afraid to exploit knowledge that the test suite may have beyond what the application knows even if this means writing code in the application that exists solely to help it be tested.

Use numerical arguments not only for conditions that are expected to happen but also for conditions that are expected *not* to happen. Expected cases are numbered consecutively from 0 to n , where n is the number listed in the registry. Unexpected cases are numbered greater than n . That way, missing conditions generate missing test coverage cases, and extra conditions generate extra test coverage cases. For example, this coverage call:

```
TCOV::TC("image", "border placement",
        ((left && (! right)) ? 0 :
         (right && (! left)) ? 1 :
         2)); // 2 not expected
```

when accompanied by this registry line:

```
border placement 1
```

would cause generation of the extra coverage case "border placement 2" if `left` and `right` were either both true or both false.

12 Obtaining the Example Code

An electronic version of this paper, the accompanying presentation, and the `search` example, including a full implementation of an automated test driver that includes support for this coverage system, is available for download at <http://www.qi.org/pstt/testcov/>. The example code is written in C++. The accompanying test suite and coverage system are implemented in perl and require perl version 5.6 or higher. The package includes a makefile that must be processed with GNU make (called `GNUmakefile`). This example has been tested on recent Linux systems running various distributions, and on Windows using Cygwin 32. It is likely that the example code would work on any POSIX-compliant system. This code is also known to pass its test suite when compiled under Windows with Visual C++ .NET. For additional information, please see the `README.txt` file in the downloadable example.