TANDEM

Uses and Abuses of Statement Coverage

Bruce W. Bailey

Technical Report 87.2
April 22, 1987
Part Number: 83053

USES AND ABUSES OF STATEMENT COVERAGE

Bruce W. Bailey

22 April 1987

Tandem Technical Report 87.2

Uses and Abuses of Statement Coverage

Bruce W. Bailey

22 April 1987

ABSTRACT

While coverage analyzers have been discussed in the literature for years, they are still rarely used by software projects. The author has developed a tool for the software developers and software quality assurance personnel at Tandem Computers that provides a measure of testing effectiveness. This tool shows which statements within a program have (and have not) been executed during a test run. The author is not aware of any other coverage facility that is fully integrated with compilers, debugging facilities, and the operating system. The improved operating characteristics and ease of use have contributed to the acceptance of this tool.

Several practical applications of a statement coverage analyzer are outlined. Additionally, some of the pitfalls of using such a facility are examined, based on experiences using the tool.

Uses and Abuses of Statement Coverage

Table of Contents

1.0 Introduction

Commercially available coverage tools are, at present, limited to specific programming languages. They generally require a pre-compilation pass of the source code to insert procedure calls that track execution paths and write the results to a file upon program termination.

Advances in coverage analyzer design have primarily concentrated on adaptations to new languages and higher orders of coverage; for example, cycle coverage.

The author has developed a statement coverage facility which is compatible with all languages that execute on Tandem NonStop Systems. It requires no pre-processing of the source code and does not noticeably degrade execution performance. In addition, it provides reporting facilities such as statistical reports, program listing mark-up, sorted displays showing the largest unexecuted program segments, and a listing which suggests an optimal order of test execution.

From the experience gained in using this tool on both the operating system and typical data-processing applications, several suggestions are made regarding the suitability of various uses for coverage data.

1.1 Background

A coverage tool was developed at Tandem specifically to test the microcode that runs in one of our intelligent terminals, the 6530. This tool was based on a hardware facility that was attached to the micro-processor being measured. After the QA organization built what was considered a comprehensive test library for the terminal, the tool was applied to determine which areas of code were not exercised. Results indicated that 45% of the code was not tested. The coverage tool was used to help direct testing until 96% coverage was achieved. Each iteration of test improvement uncovered new problems, clearly demonstrating the utility of a coverage facility.

Interest in developing a coverage facility for our mainframe processor line grew. Since Tandem supplies an entire range of architecturally equivalent processors, a software solution to the problem was particularly attractive. Experience has shown this approach to have several other benefits.

A coverage prototype that did not require special hardware support was written by the author as a research vehicle. While not suitable for general use, it helped to identify not only the feasibility of several approaches but also the relative development costs.

Subsequently, the author developed a production quality product (**COVER**) that is fully integrated into the Tandem GUARDIAN 90 operating system and functions on all Tandem NonStop systems, thereby satisfying the needs of the majority of the developers in the Software Development organization.

2.0 What COVER Provides

Basically, **COVER** measures STATEMENT COVERAGE; that is, a measure of which executable source code statements were executed during a particular run [BAIL87]. The requirements for such a facility were first outlined in [BAIL85].

The primary output from **COVER** is the "bitmap file." A bitmap file is the output from a given measurement or collection of measurements. This includes information on the object file which was measured, as well as the actual "bitmap" indicating which statements were executed. Roughly speaking, there is a bit assigned for each statement of a program. A "0" bit indicates that a statement has not been executed, a "1" bit that it has.

Given these measurement results and the original object code file, **COVER** is able to produce a variety of reports. These include the ability to mark the original program listing to highlight unexecuted statements, as well as the ability to provide statistical data, on a procedure by procedure basis, such as the total number of statements and total number of statements that were not executed.

3.0 Legitimate Uses of Statement Coverage Analysis

It is important to be clear about *why* and *when* a coverage tool should be used. Coverage tools are very powerful and provide the user with information that would not likely be made available through manual techniques. Some of the merits of coverage that might not be obvious are treated here.

3.1 Metric

Coverage is not the only metric that applies to the quality of a test library, but it is one of the more useful ones. The relationship between the metric and quality of test is clear, and it is a fairly simple metric to produce. It appears that *statement* coverage is as useful as a *metric* as any other coverage metric [WEIS85]. Note that 100% coverage of a program does not mean that it has been completely tested. However, if there is only partial coverage you can be certain that the program is either not fully tested or that it has "dead" code. Only as 100% statement coverage is approached do other measures of test coverage become necessary. A good compromise between effort in testing and the identification of errors is typically 85% [MILL79], although a coverage metric alone is not a reasonable termination criterion.

3.2 Guiding test case development

Probably the most common use of a coverage analyzer is to identify areas of code which are not tested in order to direct resources to those areas. Improving coverage often requires looking for the largest contiguous segments of statements which were not executed, then analyzing, with the aid of the developer, ways in which this code can be reached. However, see "Pitfalls."

3.3 Dead code identification

Unexecuted statements may, in fact, be "dead" code. That is, there may be *no* way to reach this code. Close inspection of the code should indicate whether this is the case and, if so, whether this represents a bug or whether the code can simply be removed. Some dead code may be found with static analyzers, but often it is dead because of an error in assumptions about possible data values at a conditional statement.

3.4 Ordering test execution

Testing is usually a cyclical activity involving QA test and Development fixes. The testing cycle continues until the product meets its release criteria. If the test suite takes a long

time to run with respect to this turn-around cycle, it would be desirable to run first those tests that are most likely to uncover bugs. Coverage information on individual tests can suggest a practical ordering: first, run the test case that exercises the most statements. Next, run the test case that adds the most coverage, and so forth. (See the appendix for a more thorough description of this algorithm.)

3.5 Identifying QA acceptance tests

In many organizations it is desirable to have the developer run a subset of the QA regression library before re-releasing his code to QA. Often a reasonable choice of tests can be made based on coverage.

3.6 Equivalency testing

A high level of coverage helps assure that a re-implimentation of a component is equivalent to the original in the absence of a complete External Specification. Too often QA is handed a specification which says: "program B does the same thing as the old program A, with these additions..." If before program B is developed, the test developer can achieve nearly 100% coverage of program A, then applying the same test suite to program B will give reasonable confidence that the old users of A will see no differences in external function when using only the original features.

3.7 Boolean operations on results

Given the bitmap files FileA and FileB corresponding to two test cases Test A and Test B, both for the same program; boolean operators prove valuable for answering the following questions:

- What is tested if we run *both* of these test cases? This can be answered by ORing (inclusive OR) FileA and FileB together. This is the most commonly used operation. In general, this is used to operate on all measurement results to provide a statistic on how much was tested by an entire test suite.

- What does Test B test that is common to Test A? This can be answered by ANDing FileA and FileB together.

- What does Test B test that has *not* already been tested by Test A? This can be answered by first ORing FileA and FileB into Temp, then XORing (exclusive OR) FileA and Temp. For example:

		Statement				
		1	2	3	4	5
	FileA	1	1	0	1	0
	FileB	0	1	0	1	1
Temp := FileA OR FileB		1	1	0	1	1
Temp := Temp XOR FileA		0	0	0	0	1

  Test B contributes an execution of statement 5.

4.0 Pitfalls of Using a Coverage Analyzer

A coverage tool is not a panacea for software bugs. Even if 100% coverage is achieved, it does *not* follow that adequate testing has been performed. Some of the pitfalls of using a coverage tool to direct test development are explored in this chapter.

Using a coverage analyzer to drive test development may produce inefficient testing. An inordinate number of test cases may be created simply to "fill in the holes." A more methodical approach may yield fewer test cases which will test the same function. A *good* test case is one which reduces *by more than a count of one*, the number of other test cases that must be developed. It should cover a large set of other possible test cases [BEIZ83].

4.1 Is the statement tested

Is the executed statement really *tested*? Testing means comparing the result of the execution with the expected result. A good test of a statement will cause it to manifest incorrect results if the statement is in error.

4.2 Boundary tests

Have the boundary conditions been tested? Coverage will not give you a clue. If a variable is to take on values [n:m] then test cases should be developed for such cases as n-1, n, n+1, and m-1, m, m+1.

4.3 Range of values

Have a wide range of data values been included? Mid-range values may influence other areas of code or may identify degenerative performance problems.

4.4 Table-driven code

If the procedure is table-driven, have *all* values of the table been used? Has every state of a state-machine been spanned? For example, in an assembler have all machine mnemonics been used?

4.5 Timing / concurrence

Have timing & concurrence conditions been adequately exercised? Are there race conditions which have not been found? Sometimes these are impossible to discover except non-deterministically. This is a major advantage of "stress testing."

4.6 Paths / cycles

Have sufficient paths been tested? For example, in the following program fragment

```
IF A THEN CALL B;
CALL C;
```

Execution of all statements does not tell you whether A was ever FALSE.

4.7 Conjunctives

Execution of both statements in the following program fragment

```
IF A OR B THEN CALL C;
```

will not indicate whether A (or B) was ever FALSE.

4.8 Initialization

Was initialization of variables performed properly? The most common example of initialization failures is when memory is cleared before program execution. A procedure may perform correctly on its first call when variables are initialized to zero, but it may fail on subsequent calls when there is "junk" left around from previous calls.

4.9 Contexts

Was a function performed in various contexts? This is especially important for operating system types of processes which run indefinitely; for example, teleprocessing monitors (Pathway) and database processes (DP2) which may perform differently based on earlier inputs.

5.0 Conclusions

The design features that contribute the most to **COVER**'s usability are

- it requires no special hardware,
- it requires no re-compilation of the source program, and
- it does not noticeably degrade execution performance.

The utility features which are particularly important are

- there is no need to modify the test suite,
- it will mark-up a program listing directly,
- it provides for a variety of statistical reports,
- it can identify the largest unexecuted segments of the program under test, and
- it can suggest an efficient order for test execution.

We expect the coverage analysis of our test libraries to help increase the level of testing of our software while reducing the over-all duration of testing. This can only be achieved if the user has a clear understanding of the value of statement coverage. The misuse of coverage information may otherwise result in less efficient testing.

6.0 Acknowledgements

Many people have provided me with encouragement and ideas. I'd like to thank those who took the time to review my material and who volunteered to use my early prototypes. In particular, I'd like to thank Alan Hibdon for being the first "guinea pig" and for comparing his measurements of the "C" compiler with those of a commercially available product, Ed Kit for providing background and demonstrating the utility of statement coverage, Chris Larson for logistic support and the original pseudo-code for the ORDER algorithm, Randy Shingai for help with final implementation details, and Keith Stobie for recommendations on how such a tool may be used.

7.0 References

[BAIL85] Bruce W. Bailey, "COVER Product Requirements", Tandem Computers Incorporated, April 29, 1985.

[BAIL87] Bruce W. Bailey, "COVER External Specification", Tandem Computers Incorporated, February 11, 1987.

[BEIZ83] Boris Beizer, "Software Testing Techniques", Van Nostrand Reinhold, 1983.

[MILL79] Edward F. Miller Jr., "Some Statistics from the Software Testing Service", ACM SIGSOFT, Software Engineering Notes, Vol. 4, No. 1, January 1979, pp. 8.

[WEIS85] Mark D. Weiser, John D. Gannon, and Paul R. McMullin, "Comparison of Structural Test Coverage Metrics", IEEE Software, Vol. 2, No. 2, March 1985, pp. 80-85.

8.0 Appendices

8.1 ORDER Algorithm

It is desirable to expose a product to very broad testing in as short a time as possible. Given an existing set of test cases which have been separately measured, it is useful to order those test cases so that the test with the highest coverage is run first. Once this is run, the next test case should be selected so that it adds the most to the total coverage, and so forth. This is a natural sort order for the measurement results.

For simplicity, assume that all test case executions require the same elapsed time. Assume there is a set of K test cases for a product and that the product comprises N program statements. We may represent the results of a statement coverage measurement on a product as a bit matrix X consisting of K rows and N columns, X(K,N), where X(k,n) is 1 if statement "n" was executed by test case "k"; otherwise X(k,n) is 0. These K rows will be considered to be the sort keys. In the following pseudo-code, only the keys will be sorted; however, in reality each record would also contain a field indicating the name of the corresponding test case.

The following pseudo-code should give a reasonably clear picture of the algorithm. Note that the selection of a test case depends on first selecting all the "better" test cases. This is what precludes the use of a partitioned sort algorithm. A consequence of this is that ORDER is an O(K**2) algorithm.

```
proc ORDER( BigK, BigN, X );
   integer BigK, BigN;              --Array dimensions.
   bit array X( BigK, BigN );       --Test measurement results.
   bit array V( BigN ); --Cumulative measurement result
-- Temporary variables
   integer SumJ, SumK, SumMax, j, k, Max;
   bit array swap( BigN );

-- The notation "*" as an array subscript means "all elements
-- along that coordinate taken one at a time."

-- The function "LOR" is the boolean inclusive OR.

-- The function "AddElements" computes the sum of the elements
-- of its vector argument.

V(*) := 0;

for k := 1 to BigK-1 do
   begin
   SumMax := SumK := AddElements( V(*) LOR X(k,*) );
   Max := k+1;
   --Find the test case which helps the most.
   for j := Max to BigK do
      begin
      SumJ := AddElements( V(*) LOR X(j,*) );
      if SumJ > SumMax then
         begin
         --Establish a new bound
         SumMax := SumJ;
         --Keep track of index for later swap
         Max := j;
         end;
      end; --j

   if SumMax > SumK then
      begin
      --Swap X(k,*) with X(Max,*)
      swap(*) := X(k,*);
      X(k,*) := X(Max,*);
      X(Max,*) := swap(*);
      end;

   V(*) := X(k,*) LOR V(*);

   end; --k
```

8.2 Sample ORDER Output

The following listing shows how **COVER** reports the coverage of a set of test executions. There is one bitmap per test execution. These are then reported according to the above sort ordering. Note that even though the test corresponding to ZZCO0004 only produced 25.3% coverage, it added more than any other test once ZZCO0006 was run.

```
COVER - T9618C00 - (15JUL87)
  Program $SYSTEM.SYS10.COVERCOM compiled  7 Apr 1987, 18:40:53
  Process \PRUNEQC.04,031 run  8 Apr 1987,  8:55:07
  by user 101,147 (SOFTDEV.BWB)
Current OS is $SYSTEM.SYS10.OSIMAGE - Version M00 (15JUL87)
COVER is ENABLEd (NO PRIV).
Program file is $SYSTEM.BWB.BRANDY dated 26 Jan 1987, 15:47:38
Measurement of $SYSTEM.BWB.BRANDY which comprises 771 statements
Sequence   Filename    % Coverage    % Cumulative
       1   ZZCO0006         57.1%           57.1%
       2   ZZCO0004         25.3%           64.3%
       3   ZZCO0002         55.3%           69.9%
       4   ZZCO0005         53.7%           71.6%
       5   ZZCO0003         56.5%           71.7%
       6   ZZCO0001         49.8%           71.7%
```

8.3 Sample Listing Mark-up

The following text is the actual output of a marked up program listing. Some columns of extraneous information were removed so the listing would fit on this page. It demonstrates how a program listing itself is marked to indicate which statements were or were not executed. Statements which were NOT executed are marked with a "*". If more than one statement occurs on the same line, a "+" is used when some statement has been executed but another has not.

```
PAGE 1     $OVER.BWBDEMO.SAMP [1]


TAL - T9250B40 - (15SEP86)
Date - Time :  04FEB87 - 08:56:54


 6.  0000   int proc abs( arg ); !Function definition
 7.  0000   int          arg;
 8.  0000   begin
 9.  0000   if arg > 0 then
10. *0003     return arg
11.  0003   else
12.  0006     return -arg;
13.  0011   end;
14.  0000
15.  0000   proc mainstreet main;
16.  0000   begin
17.  0000   int          i,j;
18.  0000
19.  0001   call initializer;   !External system procedure
20.  0006   i := 0;
21.  0010
22.  0010   while i do
23. *0012     i := 0;           !Never executed
24.  0015
25. +0015   if i then j := -j;  !Second statement not executed
26.  0022
27.  0022   j := abs( i );      !Function call
28.  0026
29.  0026   i := 2; j := 3;     !Both statements executed
30.  0032
31. +0032   call stop; j := 0;  !Second statement not executed
32.  0041
33. *0041   end;                !Implicit call stop
```

8.4 Sample Display Output

The following text is a sample of the output display of a measurement of the program "AMP." In its most verbose form, it lists every program statement and indicates whether or not that statement was executed. Note that the entry point of "MAINSTREET" is considered to be a statement because it allocates local storage. For procedures with multiple entry points it is desirable to determine whether each entry point was called.

```
Execution report.  Statements marked "*" were NOT executed.
Bitmap is from $SYSTEM.BWBDEMO.ZZCOAMP
Program file is $SYSTEM.BWBDEMO.AMP dated  4 Feb 1987,  8:56:54

Spaceid 00 word offset 000004 is base of ABS.
  Entry point at word offset 000000 is ABS
Procedure header at line      6.    in $OVER.BWBDEMO.SAMP
Source file timestamp was  4 Feb 1987,  8:56:30.
  Word offset 000000 is line      9.    in $OVER.BWBDEMO.SAMP
* Word offset 000003 is line     10.    in $OVER.BWBDEMO.SAMP
  Word offset 000006 is line     12.    in $OVER.BWBDEMO.SAMP
Of      3 statements,      1 ( 33.3%) were not executed.

Spaceid 00 word offset 000015 is base of MAINSTREET.
  Entry point at word offset 000000 is MAINSTREET
Procedure header at line     15.    in $OVER.BWBDEMO.SAMP
Source file timestamp was  4 Feb 1987,  8:56:30.
  Word offset 000000 is line     15.    in $OVER.BWBDEMO.SAMP
  Word offset 000001 is line     19.    in $OVER.BWBDEMO.SAMP
  Word offset 000006 is line     20.    in $OVER.BWBDEMO.SAMP
  Word offset 000010 is line     22.    in $OVER.BWBDEMO.SAMP
* Word offset 000012 is line     23.    in $OVER.BWBDEMO.SAMP
  Word offset 000015 is line     25.    in $OVER.BWBDEMO.SAMP
* Word offset 000017 is line     25.    in $OVER.BWBDEMO.SAMP
  Word offset 000022 is line     27.    in $OVER.BWBDEMO.SAMP
  Word offset 000026 is line     29.    in $OVER.BWBDEMO.SAMP
  Word offset 000030 is line     29.    in $OVER.BWBDEMO.SAMP
  Word offset 000032 is line     31.    in $OVER.BWBDEMO.SAMP
* Word offset 000037 is line     31.    in $OVER.BWBDEMO.SAMP
* Word offset 000041 is line     33.    in $OVER.BWBDEMO.SAMP
Of     13 statements,      4 ( 30.8%) were not executed.


Of     16 statements,      5 ( 31.3%) were not executed.
```

The following text shows how the same data may be manipulated to display only summary information for the program with a breakdown by procedure.

```
Execution report.  Entry points marked "*" were NOT executed.
Bitmap is from $SYSTEM.BWBDEMO.ZZCOAMP
Program file is $SYSTEM.BWBDEMO.AMP dated  4 Feb 1987,  8:56:54

Spaceid 00 word offset 000004 is base of ABS.
  Entry point at word offset 000000 is ABS
Of      3 statements,      1 ( 33.3%) were not executed.

Spaceid 00 word offset 000015 is base of MAINSTREET.
  Entry point at word offset 000000 is MAINSTREET
Of     13 statements,      4 ( 30.8%) were not executed.


Of     16 statements,      5 ( 31.3%) were not executed.
```

8.5 Sample SORT Output

The following listing shows how **COVER** reports the largest unexecuted segments of contiguous statements. Each segment is reported in descending order of number of statements. Included on each detail line are: the procedure name, source file name, line number, and octal offset from the beginning of the procedure. This is followed by the number of statements which were not executed.

```
COVER - T9618C00 - (15JUL87)
  Program $SYSTEM.SYS10.COVERCOM compiled  7 Apr 1987, 17:23:42
  Process \PRUNEQC.04,031 run  7 Apr 1987, 18:14:10
  by user 101,147 (SOFTDEV.BWB)
Current OS is $SYSTEM.SYS10.OSIMAGE - Version M00 (15JUL87)
COVER is ENABLEd (NO PRIV).
Bitmap is from $SYSTEM.BWBSORT.ZZCO0006
Program file is $SYSTEM.BWBSORT.BRANDY dated 26 Jan 1987, 15:47:38
Procedure name             Source file Line number Offset Statmnts
CREATE^BACKUP               SBRANDY        696.    000000       55
SET^RUN^TIME^OPTIONS        SBRANDY        572.    000253       54
DOSHOW^MENU                 SBRANDY       1369.    000000       54
ANALYZE^CHECKPOINT^STATUS   SBRANDY        749.    000000       26
ANALYZE^SYSTEM^MESSAGE      SBRANDY        794.    000000       23
SET^RUN^TIME^OPTIONS        SBRANDY        547.    000001       18
CHECKRECEIVE                SBRANDY        832.    000000       15
OPEN^BACKUPS^FILES          SBRANDY        672.    000022       11
CHECK^VERSION               SBRANDY        656.    000021        9
GET^TOKEN                   SBRANDY        464.    000027        6
OPEN^PRIMARYS^FILES         SBRANDY        636.    000025        6
EXECUTE                     SBRANDY       1443.    000071        6
BRANDY                      SBRANDY       1469.    000033        5
GOTO^                       SBRANDY       1033.    000055        4
GOTO^                       SBRANDY       1051.    000125        3
XCAL^                       SBRANDY       1179.    000105        3
OUTPUT^                     SBRANDY        345.    000024        2
IF^                         SBRANDY       1201.    000042        2
BRANDY                      SBRANDY       1485.    000124        2
```

Distributed by
TANDEM
Corporate Information Center
10400 N. Tantau Ave., LOC 248-07
Cupertino, CA 95014-0708