ATAC Tutorial

Abstract

ATAC is a test coverage analysis tool developed at Bellcore by Bob Horgan and Saul London to test programs written in the C language. ATAC measures how thoroughly a program is tested by a set of tests using data flow coverage techniques, identifies areas that are not well tested, identifies overlap among tests, and finds minimal covering test sets. This document is a tutorial introduction to the use of ATAC. It contains examples and explanations of the most significant features of ATAC. Manual pages are included as an attachment.

1. Introduction to ATAC

ATAC is a test coverage analysis tool developed at Bellcore to test programs written in the C language. ATAC was originally described in [Hor90]. ATAC measures how thoroughly a program is tested by a set of tests using data flow coverage techniques, identifies areas that are not well tested, and identifies overlap among tests. ATAC is used by software developers and testers to identify areas in a program that require further testing. ATAC provides coverage measures used in conjunction with the C Risk Browser [Agr93], *CRB* to estimate software risk and reliability. These measures may be used for project tracking to indicate testing progress and as acceptance criteria to subsequent development stages. Regression testers also may use ATAC to identify a subset of tests that achieve a high level of coverage at limited cost.

Structural coverage testing identifies program constructs that may be exercised during program execution and determines which of these constructs are in fact exercised by a set of tests. We call these constructs *testable attributes*. Testable attributes may be *blocks* of consecutive statements, branch *decisions*, and various combinations of assignments and uses of variables. A set of tests is considered *adequate* with respect to a given coverage criterion if all of the testable attributes identified by this criterion are exercised, at least once, by the set of tests. In practice, it is quite difficult to achieve completely adequate coverage so ATAC presents a coverage *measure* – the percentage of testable attributes that are exercised for a set of tests.

ATAC reports coverage measures for a number of coverage criteria. These are described in more detail in the "Coverage Measures" section. ATAC displays source code associated with each testable attribute that has not been exercised so that tests can be developed to exercise them.

ATAC has been designed to be used in the standard UNIX® C development environment. For developers and testers familiar with this environment, ATAC is very easy to learn and use. Although ATAC provides a large number of options for varying the form and content of the information reported, defaults have been chosen so that few options are required to use the basic features. Complete details of the ATAC commands and options appear in the attached manual pages.

Section 2 of this tutorial is a complete demonstration of the use of ATAC on a small program. Section 3 covers ATAC features for managing information collected for multiple tests. The remaining sections present additional features, options, and trouble shooting suggestions.

2. Getting Started

ATAC is easy to use. The basic features of ATAC are presented in this section using a small word counting program as an example of a program being tested.

Before attempting to use ATAC for the first time check that ATAC is installed on your system by entering:

```
$ atac -v
```

The output should look something like this*:

UNIX is a registered trade mark of UNIX Systems Laboratories, Inc.

* This documentation applies to ATAC version 3.0. Many of the features described here are not available with older versions. Newer versions have features not described here. (Very old versions will not even recognize the *-v* option to display the version number.)

```
ATAC release 3.0 Nov 06, 1992.
Copyright @ 1992 Bell Communications Research, Inc. (Bellcore)
Send comments or questions to atac@bellcore.com.
```

This tutorial uses a word counting program to illustrate the features of ATAC. The word counting program takes as arguments an optional list of files and an optional combination of *-l*, *-w*, and *-c* to count lines, words, or characters. By default all input comes from standard input and characters, words and lines are counted. The source code and sample input for the word counting program consists of files *makefile*, *main.c*, *wc.c*, *input1*, *input2*, and *input3*. These files are installed with ATAC. To copy the installed files into your current directory enter:

```
$ cp `atacLib`/tutorial/*  .
```

Before using ATAC, check that the word counting program compiles and that it runs on some simple inputs:

```
$ make
```

The output from *make* should look like this:

```
cc -g -c wc.c
cc -g -c main.c
cc -g -o wordcount wc.o main.o
```

If you don't like *make* you can enter the *cc* lines yourself.

The executable program is called *wordcount*. Run the program:

```
$ wordcount input1                    ← test t.1
```

(The arrow notation to the right of the command is for reference later in this document. It is not part of the command.) The output should look like this:

```
       1       4      19 input1
       1       4      19 total
```

Run the program again. This time with options:

```
$ wordcount -lwc input1               ← test t.2
```

The output should look the same.

These two tests are the initial functional tests for the *wordcount* example. Functional tests are designed to demonstrate that all required features have been implemented. Functional tests typically exercise 30% to 70% of the statements in a program. ATAC is used to help create a more thorough set of tests after the initial functional tests are passed.

To use ATAC, remove previously created object files and executable files by entering:

```
$ make cleanup
```

(This is unnecessary if you are not using *make*). Compile *wordcount* program with ATAC by entering:

```
$ make CC=atacCC
```

The output should look like this:

```
atacCC -g -c wc.c
atacCC -g -c main.c
atacCC -g -o wordcount wc.o main.o
```

As you can see, ATAC is easily integrated with existing makefiles. The options for *atacCC* are the same as the options for *cc*. Again, if you don't like *make* you can enter the *atacCC* lines yourself. If errors are encountered during compilation refer to the "Compilation and Link Errors" section.

Notice that in addition to creating the *.o* files, *main.o* and *wc.o*, and the executable program, *wordcount*, *atacCC* has created *main.atac* and *wc.atac*. *AtacCC* creates a *.atac* file for each *.c* file it compiles. These files contain static coverage information used in test analysis. (It is possible to restrict collection of coverage information to selected source files. This is discussed in the "Selective Coverage" section.)

Now run the functional tests for *wordcount*. The output from each of these tests is the same as that given when compiled without ATAC.

```
$ wordcount input1                ← test t.1
$ wordcount -lwc input1           ← test t.2
```

If these tests do not produce the same output as produced without ATAC, refer to the "Runtime Errors" section.

Notice that in addition to the expected output, executing these tests has created a file called *wordcount.trace*. This file contains dynamic coverage information used in test analysis. Now we use ATAC to see how well our functional tests have done. To obtain a summary by function (C subroutine) of the coverage obtained by these two tests enter:

```
$ atac -s -f wordcount.trace main.atac wc.atac
```

The output looks like this:

```
% blocks      % decisions   % C-Uses      % P-Uses      function
------------- ------------- ------------- ------------- -------------
74(23/31)     65(11/17)     60(38/63)     60(6/10)      main
100(8)        50(3/6)       100(4)        50(3/6)       print
100(14)       100(12)       75(12/16)     93(14/15)     count
85(45/53)     74(26/35)     65(54/83)     74(23/31)     == total ==
```

The *-s* option specifies that a summary is desired rather than a source code display. The *-f* option specifies that the summary should include information for each function. (The *-f* option may be omitted to include only the *total* line. For programs with many source files, *-g* may be substituted for *-f* to include information by source file instead of by function.) The rightmost column gives the name of a function in the *wordcount* program. The leftmost column gives the percentage of blocks covered by the two tests. A block is a code sequence that is always executed sequentially, i.e. it has no internal branching constructs. The numbers in parentheses following the percentage indicate the number of blocks covered and the total number of blocks in the function. The last line gives a total for all functions. The remaining columns will be discussed later.

In this example, 45 out of 53 blocks, or 85 percent, have been covered by two tests. Although complete block coverage does not guarantee that a set of tests will reveal all errors, testing is certainly incomplete if there are blocks of code that are not exercised by any test. In order to determine what additional test cases are needed to cover the remaining seven blocks, use the ATAC block display feature:

```
$ atac -mb wordcount.trace main.atac wc.atac
```

The *-mb* option specifies that the block coverage measure should be applied. The output is shown in Figure 1. The output is paged by *more* (or *pg*). to move to the next page enter space (or return). Blocks of code not covered by the tests are highlighted in reverse video. (The `TERM` environment variable must be set appropriately to view highlighted code.) In this example, highlighted code indicates that tests are needed for:

— Invalid options.

— Default input (from standard input).

— An input file that cannot be opened.

The following tests cover these cases:

```
$ wordcount -x                    ← test t.3
$ wordcount <input1               ← test t.4
$ wordcount nosuchfile            ← test t.5
```

Two of these tests cover error cases so the appropriate error messages are expected.

Test *t.3*, *t.4*, and *t.5* were constructed by examining the uncovered source code displayed by ATAC to determine what inputs will cause that source code to be executed. ATAC displays uncovered source code but does not construct tests or determine what inputs are needed to cover this code. Constructing these tests is the role of the tester.

```
------------------> main.c:main 8 of 31 blocks not covered <------------------
                        case 'w':
                                doword = 1;
                                break;
                        case 'c':
                                dochar = 1;
                                break;
                        default:
                                fprintf(stderr, "invalid option: -%c\n",
                                        *p);
                        case '?':
                                fputs("usage: wc [-lwc] [files]\n",stderr);
                                return 1;
                        }
                argv += 2;
        }
        else {
                ++argv;
                doline = doword = dochar = 1;
        }

        do {
                if (!*argv) {
                        count(stdin, &linect, &wordct, &charct);
                        print(doline, doword, dochar, linect, wordct, charct,
                              "");
                        return;
                }
                else {
                        file = fopen(*argv, "r");
                        if (file == NULL) {
                                perror(*argv);
                                return 1;
                        }
                        count(file, &linect, &wordct, &charct);
                        fclose(file);
                        print(doline, doword, dochar, linect, wordct, charct,
                              *argv);
                }
                tlinect += linect;
```

Figure 1. ATAC Display of Blocks Not Covered

Check the coverage summary for all tests so far:

```
$ atac -s -f wordcount.trace main.atac wc.atac
```

The output should look like this:

```
% blocks      % decisions   % C-Uses      % P-Uses      function
------------- ------------- ------------- ------------- -------------
100(31)       88(15/17)     70(44/63)     90(9/10)      main
100(8)        50(3/6)       100(4)        50(3/6)       print
100(14)       100(12)       75(12/16)     93(14/15)     count
100(53)       86(30/35)     72(60/83)     84(26/31)     == total ==
```

All 53 blocks are now covered by the five tests we have run. Even so, they are not a complete set of tests. There may be errors not revealed by these tests that will be revealed when different combinations of statements are executed or statements are executed in a different order. The remaining coverage measures are designed to help create tests that will reveal these errors.

The second column in the summary display indicates decisions that have not been exercised. A decision is a conditional branch from one block to another. As can be seen from the ATAC summary in this example, it is possible that a set of tests will cover all blocks in a program without covering some of the decisions. In this example 30 of 35 decisions are covered. In order to determine what additional test cases are needed to cover the remaining five decisions use the ATAC decision display feature:

```
$ atac -md wordcount.trace main.atac wc.atac
```

The *-md* option specifies that the decision coverage measure should be applied. The output is shown in Figure 2. The value in the margin, `´?´====>`, `TRUE===>`, or `FALSE==>` indicates the required value of the highlighted conditional to cover the decision. For example, `´?´====>` pointing at `switch(*p)` indicates that a test case is needed in which `*p` has the character value `´?´` at the `switch` statement. For *wordcount* this indicates that a test case is needed for: the *-?* option.

```
---------------> main.c:main decision not covered at line 18 <---------------
                FILE    *file;

                if (argc > 1 && argv[1][0] == '-') {
                        for (p = argv[1] + 1; *p; ++p)
'?'====>                        switch(*p) {
                                case 'l':
                                        doline = 1;
                                        break;
                                case 'w':

---------------> main.c:main decision not covered at line 63 <---------------
                        }
                        tlinect += linect;
                        twordct += wordct;
                        tcharct += charct;
TRUE===>        } while(*++argv);

                print(doline, doword, dochar, tlinect, twordct, tcharct, "total"
);
                return 0;
        }

---------------> main.c:print decision not covered at line 74 <---------------
                int     doline, doword, dochar;
                int     linect, wordct, charct;
                char    *file;
        {
FALSE==>        if (doline)
                        printf(" %7ld", linect);
                if (doword)
                        printf(" %7ld", wordct);
                if (dochar)

---------------> main.c:print decision not covered at line 76 <---------------
                char    *file;
        {
                if (doline)
                        printf(" %7ld", linect);
FALSE==>        if (doword)
                        printf(" %7ld", wordct);
                if (dochar)
                        printf(" %7ld", charct);
                printf(" %s\n", file);

---------------> main.c:print decision not covered at line 78 <---------------
                if (doline)
                        printf(" %7ld", linect);
                if (doword)
                        printf(" %7ld", wordct);
FALSE==>        if (dochar)
                        printf(" %7ld", charct);
                printf(" %s\n", file);
        }
```

Figure 2. ATAC Display of Decisions Not Covered

Note that the conditional expression highlighted in the display *was* evaluated by execution of some of the previous test cases. (Otherwise it would have appear highlighted in the block coverage display.) However, in none of those test cases did it evaluate to the required value. By default, ATAC will not display a decision as not covered if the conditional expression was never evaluated or the statement that will follow its evaluation was never exercised. In these cases, the appropriate code fragments are highlighted in the block coverage display.

The display in Figure 2 indicates that the following test cases are needed:

— *-?* option.

— Multiple passes through the file processing loop (multiple input files).

— Various combinations of *-l*, *-w*, and *-c* options.

The following tests cover these cases:

```
$ wordcount -? input1                ← test t.6
$ wordcount input1 input1            ← test t.7
$ wordcount -wc input1               ← test t.8
$ wordcount -c input1                ← test t.9
$ wordcount -l input1                ← test t.10
```

Again, it was the testers role to understand the ATAC display of decisions not covered and to create test *t.6* through *t.10* to cover them.

Check the coverage summary for all tests so far:

```
$ atac -s -f wordcount.trace main.atac wc.atac
```

The output should look like this:

```
% blocks      % decisions   % C-Uses      % P-Uses      function
------------- ------------- ------------- ------------- -------------
100(31)       100(17)       86(54/63)     100(10)       main
100(8)        100(6)        100(4)        100(6)        print
100(14)       100(12)       75(12/16)     93(14/15)     count
100(53)       100(35)       84(70/83)     97(30/31)     == total ==
```

All blocks and decisions have now been covered.

The third column in the summary display indicates computational variable uses, or c-uses, that have not been exercised. A c-use, or computational variable use, is a combination of an assignment to a variable and a use of a the variable in a computation that is not part of a conditional expression. The fourth column in the summary display indicates p-use coverage. A p-use is a combination of an assignment to a variable, a use of the variable in a conditional expression, and all branches based on the value of the conditional expression. Use the c-use/p-use display feature to reveal the remaining c-uses and p-uses.

```
$ atac -mc wordcount.trace main.atac wc.atac
$ atac -mp wordcount.trace main.atac wc.atac
```

The *-mc* or *-mp* option specifies that c-use or p-use coverage measures should be applied. The output is included as Attachments 2 and 3. The symbol `def====>` in the margin indicates the assignment (*definition*) of a variable. The symbol `use====>` indicates the use of the variable as assigned at that assignment. If both the assignment and use of a variable are on the same line, the symbol `d/use====>` is used, if the assignment comes first, or `u/def====>` is used, if the use comes first.

The following tests cover the remaining cases:

```
$ wordcount -c nosuchfile          ← test t.11
$ wordcount input1 nosuchfile      ← test t.12
$ wordcount -cX                    ← test t.13
$ wordcount -wc <input1            ← test t.14
$ wordcount -c <input1             ← test t.15
$ wordcount -l <input1             ← test t.16
$ wordcount </dev/null             ← test t.17
$ wordcount <input2                ← test t.18
$ wordcount <input3                ← test t.19
```

Now check the coverage summary for all tests so far:

```
$ atac -s -f wordcount.trace main.atac wc.atac
```

The output should look like this:

```
% blocks      % decisions   % C-Uses      % P-Uses      function
------------- ------------- ------------- ------------- -------------
100(31)       100(17)       100(63)       100(10)       main
100(8)        100(6)        100(4)        100(6)        print
100(14)       100(12)       100(16)       100(15)       count
100(53)       100(35)       100(83)       100(31)       == total ==
```

All coverage criteria measured by ATAC are now covered. From ATAC's point of view, these 19 tests are a completely adequate test of *wordcount*. Of course, all we have done is create a set of tests that will thoroughly test the program. You must check that the program actually passed the tests. This may be done while using ATAC or after recompiling with the standard compiler.

There is no guarantee that a program which has passed a completely adequate set of tests has no errors*. However, in addition producing test sets that reveal errors, the use of ATAC to achieve high coverage places the source code under intensive scrutiny which also tends to reveal errors. A complete test set combined with the effort to create such a test set is very effective at revealing errors.

3. Trace Management – Coordinating the ATAC

When a program that has been compiled with ATAC is executed, information regarding the testable attributes exercised is recorded in a *trace* file. The default name of the trace file is *prog*.trace, where *prog* is the name of the executable file. (See the "Trace File Names" section for more information.) Subsequent executions of the program append information to the same file. This trace file is used by the *atac* command to produce coverage displays and reports. (The trace file is not in human readable form.)

By default ATAC uses trace information for all tests executed so far. If, during testing, a source file is modified, trace information from tests executed before modification must be discarded. If the old trace information is not discarded ATAC displays a message indicating which test are out of date. For example, if the source file main.c were edited and compiled after execution of test t.1, the following message would be printed by *atac*.

```
wordcount.trace t.1: test is older than "main.c".
```

Information collected during this test would be omitted from the results of the *atac* command.

One way to discard obsolete trace information is to remove the trace file. This will discard information for all tests executed so far. An individual test may be deleted from the *.trace* file using the ATAC trace manager command, *atactm*. For example, a test named *t.1* may be deleted from *wordcount.trace* by entering:

```
$ atactm -d -n t.1 wordcount.trace
```

A list of all the tests represented in *wordcount.trace* is obtained using the *-L* option of *atactm.* For example:

```
$ atactm -L wordcount.trace
```

The output after test *t.1* is deleted is shown in Figure 3.

```
11/10/92-11:51:56   3.0  100 ----- t.2
11/10/92-11:53:03   3.0  100 ----- t.3
11/10/92-11:53:09   3.0  100 ----- t.4
11/10/92-11:53:14   3.0  100 ----- t.5
11/10/92-13:53:40   3.0  100 ----- t.6
11/10/92-13:53:49   3.0  100 ----- t.7
11/10/92-13:53:51   3.0  100 ----- t.8
11/10/92-13:53:53   3.0  100 ----- t.9
11/10/92-13:53:55   3.0  100 ----- t.10
11/10/92-13:56:28   3.0  100 ----- t.11
11/10/92-13:56:35   3.0  100 ----- t.12
11/10/92-14:08:05   3.0  100 ----- t.13
11/10/92-14:09:42   3.0  100 ----- t.14
11/10/92-14:09:48   3.0  100 ----- t.15
11/10/92-14:09:57   3.0  100 ----- t.16
11/10/92-14:10:03   3.0  100 ----- t.17
11/10/92-14:10:05   3.0  100 ----- t.18
11/10/92-14:10:06   3.0  100 ----- t.19
```

Figure 3. Atactm -L – Full List of Test Cases

The test list contains five columns for each test giving the date-time the test was executed, the version of ATAC

* In general, only when all possible inputs have been tested does passing the tests imply the program is error free. For most programs this is impossible.

used (*3.0*), a user assigned test cost (default 100), test trace attributes (`-----` for none), and the test name. Test cost is described in the "Test Set Minimization" section. Test trace attributes are described in later sections where appropriate. To obtain a list of test names only, use *-l* in place of *-L*. For example:

```
$ atactm -l wordcount.trace
```

gives:

```
t.2    t.4    t.6    t.8    t.10   t.12   t.14   t.16   t.18
t.3    t.5    t.7    t.9    t.11   t.13   t.15   t.17   t.19
```

By default tests are named t.*n* where *n* is a number starting from 1. Two tests never have the same name in a given trace file. A test may be named explicitly by setting the `ATAC_TEST` environment variable to the test name before the test is executed. A numeric suffix is added to the name automatically to make it unique. (A numeric suffix should not be specified in `ATAC_TEST`). A test may be renamed using *atactm* For example, a test named *t.2* in *wordcount.trace* may be renamed *lwc_test.1* by entering:

```
$ atactm -r lwc_test -n t.2 wordcount.trace
```

The *-r* option specifies the new name. The *-n* option identifies the test to be renamed. `atactm -L wordcount.trace` will now show *lwc_test.1* in place of *t.2*. A group of tests may be renamed using wild cards or multiple *-n* options for the old test names. A different *.n* suffix is added to the new test name for each test. For example, to rename *t.9* through *t.15* to *new.1* through *new.11* in *wordcount.trace* enter:

```
$ atactm -r new -n t.9 -n 't.1*' wordcount.trace
```

Wild cards are the same as those used for file names by the UNIX shell (*, ?, and [...]). Quote marks may be needed to prevent the shell from expanding the test name as file names.

Test names and wild cards may also be used with test coverage analysis. For example, to obtain the coverage for tests *new.1* through *new.11* in *wordcount.trace* enter:

```
$ atac -s -n 'new.*' wordcount.trace main.atac wc.atac
```

It may sometimes be useful to see the coverage of each test individually. Rather than enter the `atac -s -n ...` command for each test, use the per test option, *-p*. For example:

```
$ atac -s -p wordcount.trace main.atac wc.atac
```

The output (after the deletion and renaming examples above) is shown in Figure 4. The *-f* option was omitted to simplify the output. Note that the previous deletion of test *t.1* did not reduce the total coverage. This test was apparently not needed to achieve complete coverage. (The "Regression Testing" section discusses options for detecting redundant test cases or test cases that are not cost effective.)

4. Test Comparison

It is often useful to compare the coverage of one set of tests with the coverage of another set to determine where and to what extent they overlap. The *-c* option of atac provides this facility. For example, to obtain a summary of testable attributes covered by tests *new.1* through *new.11* but not covered by test *lwc_test.1*, enter:

```
$ atac -s -n 'new.*' -c lwc_test.1 wordcount.trace main.atac wc.atac
```

The output looks like this:

```
% blocks      % decisions   % C-Uses      % P-Uses
------------- ------------- ------------- -------------
17(9/53)      26(9/35)      40(33/83)     29(9/31)      == total ==
```

To display blocks covered by tests *new.1* through *new.11* but not covered by test *lwc_test.1*, enter:

```
$ atac -mb -n 'new.*' -c lwc_test.1 wordcount.trace main.atac wc.atac
```

The output looks similar to the usual display of uncovered blocks. Wild cards and multiple *-c* and *-n* options may be used.

```
% blocks      % decisions   % C-Uses      % P-Uses      test
------------- ------------- ------------- ------------- ------------
83(44/53)     71(25/35)     52(43/83)     71(22/31)     lwc_test.1
17(9/53)      11(4/35)       8(7/83)       6(2/31)      t.3
55(29/53)     49(17/35)     25(21/83)     61(19/31)     t.4
19(10/53)     11(4/35)      10(8/83)      13(4/31)      t.5
15(8/53)      11(4/35)       7(6/83)       6(2/31)      t.6
70(37/53)     60(21/35)     48(40/83)     74(23/31)     t.7
79(42/53)     69(24/35)     51(42/83)     71(22/31)     t.8
75(40/53)     66(23/35)     47(39/83)     71(22/31)     new.1
75(40/53)     66(23/35)     47(39/83)     71(22/31)     new.2
28(15/53)     20(7/35)      16(13/83)     13(4/31)      new.3
70(37/53)     60(21/35)     41(34/83)     74(23/31)     new.4
21(11/53)     14(5/35)      12(10/83)      6(2/31)      new.5
66(35/53)     63(22/35)     35(29/83)     65(20/31)     new.6
62(33/53)     60(21/35)     31(26/83)     65(20/31)     new.7
62(33/53)     60(21/35)     31(26/83)     65(20/31)     new.8
40(21/53)     17(6/35)      19(16/83)     29(9/31)      new.9
55(29/53)     49(17/35)     27(22/83)     61(19/31)     new.10
55(29/53)     46(16/35)     27(22/83)     61(19/31)     new.11
100(53)       100(35)       100(83)       100(31)       == all ==
```

Figure 4. ATAC Per Test Coverage Summary

5. Trace Compression

At the end of a test, ATAC trace files are automatically reformatted into a compressed format to reduce the file size. (The compressed trace file format is not the same as that used by general purpose file compression tools.) If a test exits abnormally, e.g. by receipt of signal, the trace file may not be compressed. To prevent the trace file from growing too large, *atactm* should be run to compress the trace file. For example, to compress *wordcount.trace* enter:

```
$ atactm wordcount.trace
```

The trace is automatically compressed when a test is renamed or deleted.

6. Selective Coverage

ATAC coverage analysis is limited to source files that have been compiled with *atacCC*. Coverage analysis can be further limited based on a combination of:

— Test cases

— Functions (C subroutines)

— Source files

Coverage analysis is restricted to a named set of tests by the `atac -n` *test-name* option as described in the "Trace Management" section, above. In this case, coverage is reported only for selected test cases.

Coverage analysis is restricted to a named set of functions by the `atac -F` *function-name* option. Wild cards or multiple *-F* options may be used to specify multiple function names. For example to obtain coverage in *wordcount.trace* for functions with names starting with *m* or *p* only, enter:

```
$ atac -s -f -F '[mp]*' wordcount.trace main.atac wc.atac
```

Coverage analysis is restricted to selected source files by including on the command line only *.atac* files corresponding to selected source files. For example to obtain coverage in *wordcount.trace* for functions in source file *wc.c* only, enter:

```
$ atac -s -f wordcount.trace wc.atac
```

If coverage information will be needed only for functions in selected source files, it is unnecessary to compile the remaining source files with ATAC. Since compilation with ATAC increases execution time, and memory use, and object code size, (see the "Performance and Timing Considerations" section) it may be advantageous to use ATAC only on source files for which coverage information is needed. The remaining source files may be compiled with the standard compiler. In any case, all object files must be linked with *atacCC*. In the *wordcount* example, if coverage information is needed for *wc.c* only, *make* may be used to compile *wordcount* as follows:

```
$ make cleanup
$ make main.o
$ make CC=atacCC
```

The first *make* will remove previously created *.o* and executable files. The second *make* will compile *main.c* with the standard compiler. The third *make* will compile *wc.c* with ATAC and link all object files with ATAC.

At times you may loose track of which object files or executable files were compiled with ATAC. The SCCS *what* command or the RCS *ident* command may be used to identify files compiled with ATAC. For example, for SCCS use:

```
what main.o wc.o wordcount | grep ATAC
```

For RCS use:

```
ident main.o wc.o wordcount | grep ATAC
```

If none of the files were compiled with ATAC, no lines are printed.

7. Trace File Names

By default, execution of a program compiled with ATAC creates a trace file in the current directory, with the name *prog*.trace, where *prog* is the original name of the executable file; that is, the name that was specified with the *-o* option to *atacCC*. If no *-o* option was given to *atacCC*, the trace file is created with the name *a.out.trace*. If the executable file is subsequently renamed, the default trace file name remains the same.

The trace file name may be overridden by setting the `ATAC_TRACE` environment variable to the trace file name before the test is executed. By convention the trace file name ends with *.trace*. The trace file name may not begin with a hyphen and may not end with *.atac* (which is reserved for *.atac* files). (Suffixes *.ata*, *.at*, and *.a*, are also reserved for *.atac* files when the filename has maximum permitted length for your system.) The value of `ATAC_TRACE` is also used by *atac*, and *atactm* if no trace file name is specified. (If `ATAC_TRACE` has no value, *a.out.trace* is used.)

The trace file may be written in a directory other than the current directory by setting the `ATAC_DIR` environment variable with a path to the desired directory. The value of `ATAC_DIR`, if set, is also used by *atac* and *atactm*.

8. Coverage Measures

ATAC implements a number of structural coverage measures of varying strength. Figure 5 shows an approximate partial ordering of the coverage measures from weak to strong.

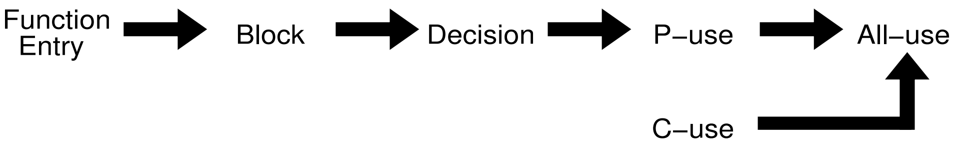

Figure 5. Approximate Partial Ordering of ATAC Coverage Measures

Coverage measure *x* is stronger than coverage measure *y* if, for any program, completely adequate coverage on measure *x* implies completely adequate coverage on measure *y* and the converse is not true. (If the converse is true, the measures are equivalent in strength.) For example, block coverage is stronger than function coverage because if all blocks are covered, all functions must have been entered. (Even an empty function is considered to have an implicit entry block.) Block coverage is weaker than decision coverage, assuming every function has at least one decision and every function called, returns, because if all decisions are covered, all blocks must be covered. We call this partial ordering *approximate* because it holds given certain assumptions which are not always true in C programs.

The weaker coverage measures usually require fewer test cases than the stronger measures to obtain completely adequate coverage. It is also easier to create test cases for the weaker coverage measures using the ATAC displays. However, completely adequate coverage for a weaker measure is less likely to reveal an error than completely adequate coverage for a stronger measure. The recommended approach is to create tests that achieve high coverage on each measure starting from the weakest and moving toward the strongest. Increasing the coverage of the weaker measures is easier and will usually increase the coverage for the stronger measures also. By default, ATAC does not display a testable attributes of a stronger measure that contains uncovered testable attributes of a weaker measure. For example, an uncovered block is not displayed if the function containing that block was never called. Similarly, an uncovered decision is not displayed if either the decision predicate or the consequent statement were never exercised. These will be displayed in the display for the weaker measure.

A completely adequate test set is not typically as easy to create as in the tutorial example in section 1. As the level of coverage increases for a measure, it becomes increasingly more difficult to create test cases that cover the remaining testable attributes. The appropriate coverage measures and target level of coverage for a given program depend on the reliability requirements for that program. In some cases it may be impossible or impractical to achieve complete coverage. When a testable attribute cannot be exercised by any test case it is called *infeasible.* ATAC attempts to identify infeasible testable attributes and omit them or count them as covered. Unfortunately, this is not possible in general. In cases where test cases cannot be created to achieve complete coverage, one should try to determine, using the ATAC display as an aid, whether the uncovered testable attributes are feasible, and if they are, under what conditions they would be exercised. If the risk of an error in the program under those conditions is acceptable, it may be unnecessary to pursue completely adequate coverage.

The number of test cases required to achieve a given level of coverage depends on the nature of the program and its input. In some cases one large test case may completely cover a program. A number of smaller test cases may be more manageable and more helpful in locating the source of an error once the error is revealed. (See also [Bei89].)

The *-m* option of *atac* is used to select coverage options. Any combination of *e, b, d, c, p, u* may be specified respectively for *function-entry, block, decision, c-use, p-use,* and *all-use.*

Function entry coverage counts the functions defined in the program that are called at least once. Function coverage includes only functions coded in the source files compiled with ATAC. The function need not be called from a function compiled with ATAC.

Block coverage counts the branch free executable code fragments that are exercised at least once. A block may be more than one C statement if there is no branching between statements. A statement may contain multiple blocks if there is branching inside the statement. An expression may also contain multiple blocks if there is branching implied in the expression (e.g. a conditional expression or logical-and or logical-or expression). If block coverage is less than 100%, there are statements that are not exercised by any test. Completely adequate block coverage implies completely adequate function-entry coverage.

Decision coverage counts the number of branches that have been followed at least once. If a decision is not covered during testing, an error in the decision predicate may not be revealed. Completely adequate decision coverage implies completely adequate block coverage except for functions with no branches.

C-use, or computational variable use coverage count the number of combinations of an assignment to a variable and a use of a the variable in a computation that is not part of a conditional expression. Since functions and statements need not use or assign any variables, c-use coverage is not comparable to most of the other measures.

P-use, or predicate variable use coverage count the number of combinations of an assignment to a variable, a use of the variable in a conditional expression, and all branches based on the value of the conditional expression. The idea behind c-use and p-use coverage is that when a variable may be assigned a value in more than one way, a good test set will insure that the uses of that variable are exercised for each possible assignment. Completely adequate p-use coverage implies completely adequate decision coverage except when there are predicates that do not contain any variables (e.g. `while (getchar() != '\n');`).

All-use coverage is the sum of p-use and c-use coverage measures.

The structural coverage measures implemented by ATAC are introduced for PASCAL in [RAP85]. Numerous other structural coverage measures have been proposed. For more information see also [Cla89], [DeM79], [Fra88], and [How82].

9. Execution Counts – Counter ATAC

In addition to recording coverage information, ATAC records the number of times each testable attribute is covered during a given test. This information may be useful as a rough measure of machine independent execution time. (However, execution counts do not account for varying execution times for different statements nor do the account for code that is not compiled with ATAC.) The total number of times each construct was covered is displayed with the *-C* option. For example, the total number of blocks executed by each test of the *wordcount* program is displayed by:

```
$ atac -C -mb -p wordcount.trace main.atac wc.atac
```

The *-mb* option specifies information for blocks only. The *-p* option specifies information for each test. The output is shown in Figure 6.

```
# blocks      test
------------- -------------
201(53)       lwc_test.1
9(53)         t.3
171(53)       t.4
10(53)        t.5
8(53)         t.6
360(53)       t.7
195(53)       t.8
189(53)       new.1
189(53)       new.2
16(53)        new.3
183(53)       new.4
13(53)        new.5
181(53)       new.6
176(53)       new.7
176(53)       new.8
21(53)        new.9
321(53)       new.10
180(53)       new.11
2599(53)      == total ==
```

Figure 6. ATAC Execution Counts Per Test Case

The last line indicates that for the 53 total blocks there were a total of 2599 block executions during these tests.

The greatest number of executions of a single testable attribute may be displayed using the *-H* option. The threshold option, *-t* may then be used to display all testable attributes executed more than a given number of times. This may be useful for performance analysis. For example, the greatest number of executions of a block for all tests of the *wordcount* program is displayed by:

```
$ atac -H -mb wordcount.trace main.atac wc.atac
```

The output indicates that one (or more) block was executed 281 times by these tests:

```
# blocks
-----------
   281      == all ==
```

Based on this result, we may choose to display all blocks executed more than 250 times using:

```
$ atac -mb -r -a -t 250 wordcount.trace main.atac wc.atac
```

The *-r* option requests that covered blocks be displayed rather than non-covered blocks. The *-a* option requests that covered blocks be displayed even if they are in a non-covered function, i.e. a function that was not entered 250 times. The output is shown in Figure 7.

```
--------------------> wc.c:count 5 of 14 blocks covered <--------------------
FILE    *file;
int     *p_nl, *p_nw, *p_nc;
{
        int c, nl, nw, nc, state;

        state = OUT;
        nl = nw = nc = 0;
        while (EOF != (c = getc(file))) {
                ++nc;
                if (c == '\n')
                        ++nl;
                if (c == ' ' || c == '\n' || c == '\t')
                        state = OUT;
                else if (state == OUT) {
                        state = IN;
                        ++nw;
                }
        }
        *p_nl = nl;
```

Figure 7. ATAC Display of Blocks Executed 250 or More Times

If a test exits abnormally, execution count information is not available. The *atactm -L* command indicates tests for which frequency information is not available by displaying the letter *f* in place of the first hyphen of the test attributes. In the following example, frequency information is not avaiable for tests *t.2* and *t.4.*

```
$ atactm -L wordcount.trace
11/10/92-11:51:56   3.0  100 f---- t.2
11/10/92-11:53:03   3.0  100 ----- t.3
11/10/92-11:53:09   3.0  100 f---- t.4
11/10/92-11:53:14   3.0  100 ----- t.5
```

10. Regression Testing

Regression testing refers to testing a modified program or system to determine that recent changes have not adversely affected existing features. Regression testing may be performed on the whole system or on various subsystems or units. At every stage of testing it is a good idea to save tests to be run later as a regression test. Test developed with ATAC are particularly useful as regression tests because it is know how thoroughly they test the program. (See also [Agr92], [Ost88].)

As your test set grows larger you may wish to identify a subset of the tests that achieves high coverage yet is easier to set-up, execute, and evaluate than the complete set of tests. ATAC can do this in two ways. The ATAC minimization command, *atacMin*, will attempt to identify a minimal cost subset of all the tests. The following command lists the names of a minimal subset of the tests in the *wordcount* example that achieve the same block and decision coverage as all tests combined.

```
$ atacMin -mbd wordcount.trace main.atac wc.atac
```

The output looks like this:

```
new.11  new.2   new.4   new.5   t.6     t.8
```

The tests are listed in order of decreasing added coverage.* The *-mbd* option requests that minimization be based on blocks and decision coverage only. By default, block, decision, p-use and c-use coverage are considered.

---

* This ordering is implemented in ATAC version 3.1.

Minimization may be limited to selected tests or selected functions (subroutines) using the *-n* and *-F* options in the same way they are used with the *atac* command.

The *cost* of each test** is a non-negative integer assigned by the tester with the *-c* option of the *atactm* command.

```
$ atactm -n t.3 -c 500 wordcount.trace
```

The unit of cost is determined by the user. Cost should be assigned to reflect the relative difficulty and expense of executing the test case. Depending on the nature of the program this may a function of execution time, tester time, test setup time, or any of a number of other factors. ATAC assigns an initial cost of 100 to each test. The current cost assignments are displayed with the *atactm -L* command. (See the "Trace Management" section above.)

In practice, *atacMin* identifies a minimal covering subset fairly quickly; in fact, the algorithm used is faster than any published algorithm for the minimal set cover problem. However, since the problem is in the class of NP complete problems, there may be cases for which *atacMin* cannot find a minimal subset in a reasonable amount of time. *AtacMin* provides additional options, that will limit the search time by reporting a "minimal or near minimal" result (See the attached manual page for *atacMin*.)

In some cases, a cost effective test set need not be minimal. You may wish to make the cost versus effectiveness trade-off decision yourself. ATAC provides the *-SKq* options*** to sort the tests in order of decreasing added cost effectiveness for this purpose. For example, using our tests for *wordcount*, cost effectiveness is given with the command:

```
$ atac -s -SKq wordcount.trace main.atac wc.atac
```

The *-S* option requests that the tests be sorted in order of increasing additional coverage. The *-K* option requests an output column listing the cost of each test. The *-q* option requests that the cost and coverage values be cumulative; i.e. the line for each test contains the total cost and coverage for all tests up to and including itself. The output using the default cost of 100 per test is shown in Figure 8.

```
cost   % blocks      % decisions   % C-Uses      % P-Uses      test
(cum)  (cumulative)  (cumulative)  (cumulative)  (cumulative)
------ ------------- ------------- ------------- ------------- -------------
100    83(44/53)     71(25/35)     52(43/83)     71(22/31)     lwc_test.1
200    89(47/53)     80(28/35)     66(55/83)     81(25/31)     new.4
300    94(50/53)     86(30/35)     73(61/83)     87(27/31)     new.9
400    94(50/53)     91(32/35)     78(65/83)     94(29/31)     new.1
500    100(53)       94(33/35)     80(66/83)     94(29/31)     t.3
600    100(53)       94(33/35)     86(71/83)     94(29/31)     t.7
700    100(53)       97(34/35)     89(74/83)     97(30/31)     new.8
800    100(53)       97(34/35)     93(77/83)     97(30/31)     new.6
900    100(53)       97(34/35)     95(79/83)     97(30/31)     new.2
1000   100(53)       97(34/35)     96(80/83)     100(31)       new.11
1100   100(53)       97(34/35)     98(81/83)     100(31)       t.5
1200   100(53)       100(35)       98(81/83)     100(31)       t.6
1300   100(53)       100(35)       99(82/83)     100(31)       new.3
1400   100(53)       100(35)       100(83)       100(31)       new.5
1500   100(53)       100(35)       100(83)       100(31)       t.4
1600   100(53)       100(35)       100(83)       100(31)       t.8
1700   100(53)       100(35)       100(83)       100(31)       new.7
1800   100(53)       100(35)       100(83)       100(31)       new.10
```

Figure 8. ATAC Cost Effectiveness Summary

** The cost feature is implemented in ATAC version 3.1. In version 3.0, the *atacMin* command assumes equal cost for each test regardless of the cost assigned.

*** These option are implemented in ATAC version 3.1.

From this summary it is easy to identify a subset of the tests that achieve a given level of coverage at reduced cost. For example, all tests together have cost 1800. Tests *lwc_test.1* through *t.3* achieve at least 80% coverage on all measures at a total cost of 500. At a total cost of 1000 we could achieve at least 96% coverage on all measures using tests *lwc_test.1* through *new.11*.

11. Code Preprocessors

The standard C preprocessor, *cpp*, is invoked automatically by the standard C compiler, *cc*. ATAC analyses the program after preprocessing is complete. However, source code displays show the original code. To accomplish this ATAC uses its own preprocessor which behaves like *cpp* except that it embeds column position information in the generated code which ATAC uses to display code fragments accurately. When uncovered testable attributes exist inside a preprocessor macro expansion, the ATAC display will highlight the macro name and arguments. The *putc* and *getc* macros are treated as an exception because they are used frequently and their internal structure is of no interest to the general tester. These two macros are replaced with calls to *fputc* and *fgetc*.

In addition, to the standard C preprocessor, C code is often generated by special purpose preprocessors such as *yacc*, *sql*, *m4*, or *cfront*. ATAC may be used with these preprocessors by calling atacCC after the source code has been appropriately preprocessed. Most preprocessors emit `#line` directives that indicate the original source code files and line numbers. ATAC will use these directives if they are present. If these directives are not present or if the preprocessor has changed column positions in the source code, ATAC's source code display may be inaccurate.

12. Overriding CC

By default, ATAC uses the C compiler called *cc* that is found first in $PATH. This option may be overridden by setting the environment variable `ATAC_CC` to the name or complete path of an alternate C compiler.

Some programs may contain conditional compilation directives (`#if` or `#ifdef`) based on predefined preprocessor symbols (e.g. #ifdef unix). By default, ATAC defines the symbols that it finds predefined by the C preprocessor used at installation. If an alternate C compiler is specified by the ATAC_CC environment variable, ATAC defines the symbols that it finds predefined by the preprocessor invoked by that compiler. In either case, the list of predefined symbols is revealed by *atacCC* with the *-E* option and no other arguments:

```
$ atacCC -E
```

On a Sun workstation the output looks like this:

```
sparc 1
sun 1
unix 1
```

Unfortunately, ATAC cannot always determine all the symbols defined by a given preprocessor.

13. Performance and Timing Considerations

Programs compiled with *atacCC* will execute somewhat more slowly than normally. Execution times vary among programs depending on the nature of the computation and the size of the trace file. This affect is only present during testing with ATAC. After testing, the program should be recompiled without ATAC to obtain maximum execution efficiency. This affect is insignificant in most testing environments. However, in some time dependent applications the change in timing behavior may affect the execution of the program.

Execution time may be reduced, at a cost of some disk space, by removing the trace compression step. Normally, ATAC attempts to compress the trace file at the end of each test execution. If execution time is critical during program exit, trace compression may be suppressed by setting the ATAC_NOCOMPRESS environment variable before the program is executed. In this case, it is advisable to run *atactm* on the trace occasionally, between tests, to reduce the size of the trace file. To resume automatic trace compression, unset the ATAC_NOCOMPRESS environment variable.

The impact of ATAC on execution time may also be reduced by limiting the scope of testing to subsets of the source files as described in the "Selective Coverage" section above. In this case, execution time is only affected while executing code in the source files compiled with ATAC. Overall coverage information is obtained by executing the same tests on each selected subset.

Execution time may also be reduced when testing in a file-server environment, by directing the trace file to a file on a local disk. (See the "Trace File Names" section above.) This will eliminate network file access time while writing to the trace file.

14. Compilation and Link Errors

When a program fails to compile or link with *atacCC* check that it compiles and links without errors using the standard *cc*. A program that compiles and links without errors using *cc* should compile and link without errors using *atacCC* with the same arguments, with the following exceptions:

1. *AtacCC* considers `const, enum, signed, void,` and `volatile` to be keywords as required by the ANSI C standard*. *AtacCC* does not support the use of these keywords as identifier names.
2. *AtacCC* does not support reuse of `typedef` names in a given source file. ANSI C and some other compilers allow reuse of `typedef` names in a nested scope or outside the scope of the `typedef.`
3. *AtacCC* does not support the ANSI C empty `struct/union` declaration to clear the definition of a struct/union tag. These declarations are ignored.
4. *AtacCC* does not support anachronistic C constructs that may be present in programs written before 1977. These constructs, such as `=+` instead of `+=` and `int x 6;` instead of `int x=6;`, are still supported by some compilers. Spaces inside assignment operators such as `= +` instead of `+=` are not supported by *atacCC*. *AtacCC* does not support struct/union member names used in association with a struct/union in which they were not declared.
5. *AtacCC* may not support extensions to the C language such as in-line assembler code sections, syntax variations, etc.
6. Some C compilers do not consider identifier names significant beyond the 6th character position. *AtacCC* considers the whole name significant. As a result *AtacCC* may detect misspelled variable names that are not detected by these compilers.
7. The preprocessor for *atacCC* searches for system include files in */usr/include*. If other directories must be searched they should be specified with the *-I* option.
8. *AtacCC* requires write permission in */usr/tmp* for temporary files.

In most of these cases *atacCC* will issue an `atacCC: parse failed` message and fail to compile the program.

After preprocessing and data-flow instrumentation of a source program, *atacCC* passes the modified program to the standard C compiler. (See the "Overriding CC" section above.) Errors from the standard C compiler may indicate an error in ATAC or in the standard C compiler.

A program compiled with *atacCC* must be linked with *atacCC.* If linked with the standard compiler the linker will issue a message indicating an undefined external symbol named *aTaC* or *_aTaC.*

15. Runtime Errors

A program compiled with *atacCC* should exhibit the same behavior as the same program compiled with *cc* with the following exceptions:

1. The program compiled with *atacCC* will create or append to a trace file. (See the "Trace File Names" section, above.)
2. The program compiled with *atacCC* will execute somewhat more slowly than the original program. If the program is timing sensitive, this may cause other behavioral differences. (See the Performance and "Timing

---

* American National Standard for Information Systems – Programming Language C, Document Number X3J11.

Considerations" section, above.)

3. *AtacCC* attempts to define the same predefined symbols as the standard C compiler. (See the "Overriding CC" section, above.) If a symbol is used that is predefined by the standard C compiler predefines but not defined by *atacCC* the program may be compiled differently by *atacCC*.
4. *AtacCC* translates trigraph sequences as required by the ANSI C standard. The value of strings containing the sequence `??` may be different than the value given by a non ANSI compiler.
5. In accordance with the ANSI C standard, *atacCC* does not expand preprocessor macro arguments that occur inside a string. The value of strings inside preprocessor macros may be different than the value given by a non ANSI compiler.
6. The ATAC runtime routine uses `malloc` for memory allocation. Programs that allocate memory in a manner that is not compatible with `malloc` may not execute properly with ATAC.
7. ATAC redefines the standard library macros *putc* and *getc* to *fputc* and *fgetc*. User supplied redefinition of these macros is ignored.
8. Programs that appear to behave correctly sometimes contain memory violations which, by chance, do not interfere with program execution. (E.g. accessing unallocated memory locations.) When ATAC is used these violations may begin to interfere with program execution.

16. Trace Corruption

ATAC does not support parallel test execution. In particular, if two tests are executing at the same time with the same trace file, the trace file may become corrupted. When this happens, the *atac* and *atactm* commands may issue a message such as:

```
wordcount.trace: corrupted trace line: 58
```

The *atactm -L* command indicates tests that have corrupted trace information by displaying the letter *c* in place of the second hyphen of the test attributes. In the following example, tests *t.3* and *t.4* are corrupted.

```
$ atactm -L wordcount.trace
11/10/92-11:51:56   3.0  100 ----- t.2
11/10/92-11:53:03   3.0  100 -c--- t.3
11/10/92-11:53:09   3.0  100 -c--- t.4
11/10/92-11:53:14   3.0  100 ----- t.5
11/10/92-13:53:40   3.0  100 ----- t.6
```

The offending test cases may be removed using the *-d* option of *atactm*:

```
$ atactm -d -n t.3 -n t.4 wordcount.trace
```

Trace corruption can occur from a single test if interrupt handler and non-interrupt handler code are being tested at the same time. ATAC tries to avoid trace corruption in these cases by ignoring interrupt handler coverage at critical times. When trace corruption does occur, the test may be delete and rerun.

17. Availability

ATAC is available for most 32 bit UNIX-like systems. A windowed environment is not required.

References

[Agr92] H. Agrawal, J.R. Horgan, E.W. Krauser, S.A. London, "Incremental Regression Testing". *Proceedings of the 1993 IEEE Conference on Software Maintenance* (CSM'93), Montreal, Canada, September 27-30, 1993.

[Agr93] H. Agrawal, J.R. Horgan, E.W. Krauser, S.A. London, "A Testing Based Model and Risk Browser for C". *Proceedings of the Second IASTED International Conference on Reliability, Quality Control, and Risk Assessment*, Cambridge, Massachusetts, October 13-15, 1993.

[Bie89] J.M. Bieman, J.L. Schultz, "Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion". *Third Workshop on Software Testing, Verification, and Analysis*, Key West, IEEE Computer Society, 1989.

[Cla89] L. Clarke, A. Podgurski, D. Richardson, S. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria". IEEE Transaction on Software Engineering, SE-15(11):244-251, Nov 1989.

[DeM79] R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Program Mutation: A New Approach to Program Testing". *Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers*, Infotech International, 1979. pp. 107-126.

[Fra88] P.C. Frankl, E.J. Weyuker, "An applicable Family of Data Flow Testing Criteria *IEEE Transactions on Software Engineering*, SE-11, No.4, April 1985.

[Hor90] J.R. Horgan, S.A. London, "A Data Flow Coverage Testing Tool for C". *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, New Orleans, LA, 1992.

[Hor91] J.R. Horgan, S.A. London, "Data Flow Coverage and the C Language". *ACM SIGSOFT Symposium on Testing, Analysis, and Verification (TAV4)*, Victoria, B.C., 1991.

[How82] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, SE-8(4), July 1982. pp. 371-379.

[Las83] J.W. Laski, B. Korel, "A Data Flow Oriented Program Testing Strategy". *IEEE Transactions on Software Engineering*, SE-9(3), 1983.

[Ost88] T.J. Ostrand, E.J. Weyuker, "Using Data Flow Analysis for Regression Testing". *Sixth Pacific Northwest Software Quality Conference*, Portland, Oregon, 1988.

[Rap85] S. Rapps, E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information". *IEEE Transactions on Software Engineering*, SE-11, No.4, April 1985.

Uncovered C-use Display for Wordcount Program

```
----------------> C-USE of argv in main; main.c line 35 <----------------
                                case '?':
                                        fputs("usage: wc [-lwc] [files]\n",stder
r);
                                        return 1;
                                }
def====>                argv += 2;
                }
                else {
                        ++argv;
                        doline = doword = dochar = 1;

o o o o o o o o o o o o o (7 lines skipped) o o o o o o o o o o o o o o o o o o

                                return;
                        }
                        else {
                                file = fopen(*argv, "r");
                                if (file == NULL) {
use====>                                perror(*argv);
                                        return 1;
                                }
                                count(file, &linect, &wordct, &charct);
                                fclose(file);
^L----------------> C-USE of argv in main; main.c line 52 <----------------
                        }
                        else {
                                file = fopen(*argv, "r");
                                if (file == NULL) {
use====>                                perror(*argv);
                                        return 1;
                                }
                                count(file, &linect, &wordct, &charct);
                                fclose(file);

o o o o o o o o o o o o o (1 lines skipped) o o o o o o o o o o o o o o o o o o

                                           *argv);
                        }
                        tlinect += linect;
                        twordct += wordct;
                        tcharct += charct;
def====>        } while(*++argv);

                print(doline, doword, dochar, tlinect, twordct, tcharct, "total"
);
                return 0;
        }
^L------------------> C-USE of p in main; main.c line 17 <-----------------
--More--
```

Attachment 1 - page 1

Uncovered C-use Display for Wordcount Program

```
----------------> C-USE of argv in main; main.c line 35 <----------------
                                case '?':
                                        fputs("usage: wc [-lwc] [files]\n",stder
r);
                                        return 1;
                                }
def====>                argv += 2;
                }
                else {
                        ++argv;
                        doline = doword = dochar = 1;

o o o o o o o o o o o o o (7 lines skipped) o o o o o o o o o o o o o o o o o o

                                return;
                        }
                        else {
                                file = fopen(*argv, "r");
                                if (file == NULL) {
use====>                                perror(*argv);
                                        return 1;
                                }
                                count(file, &linect, &wordct, &charct);
                                fclose(file);
^L----------------> C-USE of argv in main; main.c line 52 <----------------
                        }
                        else {
                                file = fopen(*argv, "r");
                                if (file == NULL) {
use====>                                perror(*argv);
                                        return 1;
                                }
                                count(file, &linect, &wordct, &charct);
                                fclose(file);

o o o o o o o o o o o o o (1 lines skipped) o o o o o o o o o o o o o o o o o o

                                   *argv);
                        }
                        tlinect += linect;
                        twordct += wordct;
                        tcharct += charct;
def====>        } while(*++argv);

                print(doline, doword, dochar, tlinect, twordct, tcharct, "total"
);
                return 0;
        }
^L------------------> C-USE of p in main; main.c line 17 <-----------------
--More--
```

Attachment 1 - page 2

Uncovered C-use Display for Wordcount Program

```
^L------------------> C-USE of p in main; main.c line 17 <-----------------
-
                int     doline = 0, doword = 0, dochar = 0;
                FILE    *file;

                if (argc > 1 && argv[1][0] == '-') {
def====>                for (p = argv[1] + 1; *p; ++p)
                                switch(*p) {
                                case 'l':
                                        doline = 1;
                                        break;

o o o o o o o o o o o o o (3 lines skipped) o o o o o o o o o o o o o o o o o

                                case 'c':
                                        dochar = 1;
                                        break;
                                default:
                                        fprintf(stderr, "invalid option: -%c\n",
use====>                                        *p);
                                case '?':
                                        fputs("usage: wc [-lwc] [files]\n",stder
r);
                                        return 1;
                                }
^L---------------> C-USE of doline in main; main.c line 13 <---------------
        {
                char    *p;
                long    tlinect = 0, twordct = 0, tcharct = 0;
                int     linect, wordct, charct;
def====>        int     doline = 0, doword = 0, dochar = 0;
                FILE    *file;

                if (argc > 1 && argv[1][0] == '-') {
                        for (p = argv[1] + 1; *p; ++p)

o o o o o o o o o o o o o (22 lines skipped)  o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L---------------> C-USE of doline in main; main.c line 20 <---------------
--More--
```

Attachment 1 - page 3

Uncovered C-use Display for Wordcount Program

```
^L---------------> C-USE of doline in main; main.c line 20 <---------------
                if (argc > 1 && argv[1][0] == '-') {
                        for (p = argv[1] + 1; *p; ++p)
                                switch(*p) {
                                case 'l':
def====>                                doline = 1;
                                        break;
                                case 'w':
                                        doword = 1;
                                        break;

o o o o o o o o o o o o o (15 lines skipped)  o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L---------------> C-USE of doword in main; main.c line 13 <---------------
        {
                char    *p;
                long    tlinect = 0, twordct = 0, tcharct = 0;
                int     linect, wordct, charct;
def====>        int     doline = 0, doword = 0, dochar = 0;
                FILE    *file;

                if (argc > 1 && argv[1][0] == '-') {
                        for (p = argv[1] + 1; *p; ++p)

o o o o o o o o o o o o o (22 lines skipped)  o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L---------------> C-USE of doword in main; main.c line 23 <---------------
--More--
```

Attachment 1 - page 4

Uncovered C-use Display for Wordcount Program

```
^L---------------> C-USE of doword in main; main.c line 23 <---------------
                                case 'l':
                                        doline = 1;
                                        break;
                                case 'w':
def====>                                doword = 1;
                                        break;
                                case 'c':
                                        dochar = 1;
                                        break;

o o o o o o o o o o o o o (12 lines skipped)  o o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L---------------> C-USE of dochar in main; main.c line 13 <---------------
        {
                char    *p;
                long    tlinect = 0, twordct = 0, tcharct = 0;
                int     linect, wordct, charct;
def====>        int     doline = 0, doword = 0, dochar = 0;
                FILE    *file;

                if (argc > 1 && argv[1][0] == '-') {
                        for (p = argv[1] + 1; *p; ++p)

o o o o o o o o o o o o o (22 lines skipped)  o o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L---------------> C-USE of dochar in main; main.c line 26 <---------------
--More--
```

Attachment 1 - page 5

Uncovered C-use Display for Wordcount Program

```
^L--------------> C-USE of dochar in main; main.c line 26 <---------------
                                case 'w':
                                        doword = 1;
                                        break;
                                case 'c':
def====>                                dochar = 1;
                                        break;
                                default:
                                        fprintf(stderr, "invalid option: -%c\n",
                                                *p);

o o o o o o o o o o o o o (9 lines skipped) o o o o o o o o o o o o o o o o o o

                }

                do {
                        if (!*argv) {
                                count(stdin, &linect, &wordct, &charct);
use====>                        print(doline, doword, dochar, linect,
wordct, charct,
                                      "");
                                return;
                        }
                        else {
^L-----------------> C-USE of nl in count; wc.c line 15 <-----------------
-
        {
                int c, nl, nw, nc, state;

                state = OUT;
def====>        nl = nw = nc = 0;
                while (EOF != (c = getc(file))) {
                        ++nc;
                        if (c == '\n')
                                ++nl;

o o o o o o o o o o o o o (2 lines skipped) o o o o o o o o o o o o o o o o o o

                        else if (state == OUT) {
                                state = IN;
                                ++nw;
                        }
                }
use====>        *p_nl = nl;
                *p_nw = nw;
                *p_nc = nc;
        }
^L-----------------> C-USE of nl in count; wc.c line 19 <-----------------
--More--
```

Attachment 1 - page 6

Uncovered C-use Display for Wordcount Program

```
^L------------------> C-USE of nl in count; wc.c line 19 <-----------------
-
                nl = nw = nc = 0;
                while (EOF != (c = getc(file))) {
                        ++nc;
                        if (c == '\n')
d/use==>                        ++nl;
                        if (c == ' ' || c == '\n' || c == '\t')
                                state = OUT;
                        else if (state == OUT) {
                                state = IN;
^L------------------> C-USE of nw in count; wc.c line 15 <-----------------
-
        {
                int c, nl, nw, nc, state;

                state = OUT;
def====>        nl = nw = nc = 0;
                while (EOF != (c = getc(file))) {
                        ++nc;
                        if (c == '\n')
                                ++nl;

o o o o o o o o o o o o o (3 lines skipped) o o o o o o o o o o o o o o o o o o

                                state = IN;
                                ++nw;
                        }
                }
                *p_nl = nl;
use====>        *p_nw = nw;
                *p_nc = nc;
        }
^L------------------> C-USE of nc in count; wc.c line 15 <-----------------
-
        {
                int c, nl, nw, nc, state;

                state = OUT;
def====>        nl = nw = nc = 0;
                while (EOF != (c = getc(file))) {
                        ++nc;
                        if (c == '\n')
                                ++nl;

o o o o o o o o o o o o o (4 lines skipped) o o o o o o o o o o o o o o o o o o

                                ++nw;
                        }
                }
                *p_nl = nl;
                *p_nw = nw;
use====>        *p_nc = nc;
        }
```

Attachment 1 - page 7

Uncovered P-use Display for Wordcount Program

```
----------------> P-USE of state in count; wc.c line 14 <----------------
        int     *p_nl, *p_nw, *p_nc;
        {
                int c, nl, nw, nc, state;

def====>        state = OUT;
                nl = nw = nc = 0;
                while (EOF != (c = getc(file))) {
                        ++nc;
                        if (c == '\n')
                                ++nl;
                        if (c == ' ' || c == '\n' || c == '\t')
                                state = OUT;
TRUE===>                else if (state == OUT) {
                                state = IN;
                                ++nw;
                        }
                }
```

Attachment 2