Achieving Software Quality with Testing Coverage Measures

Joseph R. Horgan, Saul London, and Michael R. Lyu, Bellcore

Coverage testing helps the tester create a thorough set of tests and gives a measure of test completeness. The concepts of coverage testing are well-described in the literature.[1,2] However, there are few tools that actually implement these concepts for standard programming languages, and their realistic use on large-scale projects is rare.

In this article, we describe the uses of a dataflow coverage-testing tool for C programs — called ATAC for Automatic Test Analysis for C[3] — in measuring, controlling, and understanding the testing process. We present case studies of two real-world software projects using ATAC. The first study involves 12 program versions developed by a university/industry fault-tolerant software project for a critical automatic-flight-control system.[4,5] The second study involves a Bellcore project of 33 program modules.

These studies indicate that coverage analysis of programs during testing not only gives a clear measure of testing quality but also reveals important aspects of software structure. Understanding the structure of a program, as revealed in coverage testing, can be a significant component in confident assessment of overall software quality.

When is a program considered acceptable? Investigating the relationship between the quality of dataflow testing and the subsequent detection of field faults may lead to new criteria.

Metrics in dataflow testing

The purpose of software testing is to detect errors in a program and, in the absence of errors, impart confidence in the program's correctness. Just as an adequate test of a used car consists of a satisfactory test drive and a complete test of the car's components by a mechanic, thorough software testing requires both functional and coverage testing.[1,6] *Functional testing* assures that a program meets its specifications by exercising the features described in the specification. This kind of testing depends only on program specifications and is independent of encoding. *Coverage testing* identifies constructs in program encoding that have not been exercised during testing. It guides the testing of important software constructs and gives a clear checklist of test completeness.

Each coverage criteria proposed in the literature[1,7] captures some important aspect of a program's structure. Rapps and Weyuker[7] define a family of dataflow coverage criteria for an idealized programming language. Frankl and Weyuker[2] extend these definitions to a subset of Pascal and describe a tool to check for test completeness

0018-9162/94/$4.00 © 1994 IEEE

based on the dataflow coverage criteria. We have adapted these dataflow coverage definitions to define realistic dataflow coverage measures for C programs. A coverage measure associates a value with a set of tests for a given program. This value indicates the completeness of the set of tests for that program. We define the following dataflow coverage measures for C programs based on Rapps and Weyuker's[7] definitions: block, decision, c-use, p-use, all-uses, path, and du-path.

Precisely defining these concepts for the C language requires some care, but the basic ideas can be illustrated by the example in Figure 1. We define the measures to be intraprocedural, so they apply equally well to individual procedures (functions), sets of procedures, or whole programs.

Figure 1. Sum.c computes the sum and product of numbers from 0 to *N*.

```
#include <stdio.h>
main()
{
        int n, i, k, sum, prod;

        printf("Enter an integer and 0 for +, 1 for *: ");
        scanf("%d %d", &n, &k);
        sum = 0;
        prod = 1;
        i = 1;
        while (i <= n)
        {
            sum += i;
            prod *= i;
            i++;
        }
        if(k == 0)
            printf("n = %d, sum = %d\n", n, sum);
        if(k == 1)
            printf("n = %d, prod = %d\n", n, prod);
}
```

A definition of variable *i*

Uses of variable *i*

A basic block

A decision predicate involving variable *k*

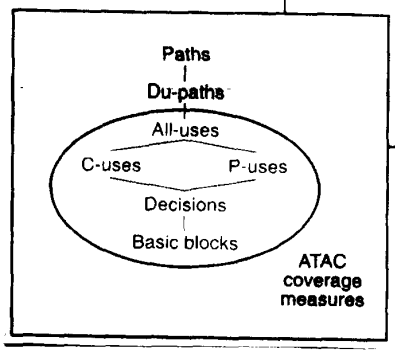


Figure 2. A hierarchy of control and dataflow coverage measures.

Block. The simplest example of a coverage measure is basic block coverage. The body of a C procedure may be considered as a sequence of basic blocks. These are portions of code that normally execute together, that is, consecutive code fragments without branches. The block coverage of a set of tests is the ratio of the basic blocks executed (covered) to the total number in the program. Thus, the block coverage measure indicates the fraction of basic program blocks executed by the tests. Block coverage is similar to statement coverage but more sensitive to program structure.

Decision. Although essential, basic block coverage is not a sufficient measure of test completeness. A decision exists for each possible value of a branch predicate. For instance, in Figure 1, the predicate $k == 1$, which may be true or false, has two decisions associated with it, and a case predicate may have many associated decisions. The decision coverage of a set of tests is the ratio of the number of decisions covered by the tests to the total number of decisions in the program.

C-use, p-use, and all-uses. Dataflow coverage testing directs the tester to construct test cases that cover all definition-use pairs. A *definition* is a statement, like $i = 1$ in Figure 1, that assigns a value to a variable, and the occurrence of i in the statement "sum += i" is a *use* of i. These two occurrences of i constitute a def-use pair. If, as in our example, the use appears in a computational expression, the pair is a *c-use*. If the use appears inside a predicate, for example, i in $i \leq n$, then the pair is a *p-use*. An *all-uses* is either a c-use or a p-use.

Path and du-path. Any sequence of statements defined by program control flow is a *path*. A path from a variable's definition to its use, which contains no redefinition of the variable, is called a *du-path*. In Figure 1, an example of a du-path would be execution of a sequence that starts at $i = 1$, loops once in the body of the while statement, and then continues. However, paths with two or more loops from $i = 1$ to sum += i do not constitute a du-path because i++ redefines i.

Any of these control or dataflow constructs are covered if they execute during the test.

Hierarchy. Each basic block, decision, definition, and path in a program is an attribute that may contain a fault that could cause program failure. Therefore, as many attributes as possible should be tested to improve the chances of detecting any remaining faults. (Note that we use "fault" to refer to a coding error and reserve "error" to describe incorrect program behavior, presumably due to one or more faults in the code.)

Figure 2 suggests an ordering of the coverage criteria. In this hierarchy, block coverage is weaker than decision coverage, which in turn is dominated by p-use coverage. C-use coverage dominates both block and decision coverage but is independent of p-use coverage; both c-use and p-use coverage fall below all-uses. Rapps and Weyuker[7] originally described this hierarchy as a containment relationship among the various criteria. Although containment fails for the C language, the relationships described by the hierarchy are useful. We suggest, for instance, exhausting block testing before undertaking decision testing. Decision testing is more effective, but it also takes more time and effort. Therefore, working up the hierarchy is a prudent strategy for realizing the benefits of coverage testing.

ATAC software coverage tool

ATAC evaluates test-set completeness using dataflow coverage measures. It computes dataflow coverage adequacy, using data collected from static analysis of source code and dynamic analysis of execution paths, and incorporates techniques for improving software quality, using the data originally collected to compute coverage adequacy. The program constructs measured by ATAC include blocks, decisions, c-uses, p-uses, and all-uses. (For a simple example, see the sidebar at right) Analysis can be performed

for each test case, source file, and C function or for various combinations. Multiple source files can be tested together or one at a time. There are no explicit limits on the size of programs tested with ATAC.

The ATAC preprocessor analyzes C source code (according to the ANSI standard or Kernighan and Ritchie definition) and produces a file containing dataflow information for use in the analysis phase. The preprocessor also creates a modified version of the source code instrumented with calls to the ATAC runtime routine. The modified source code is automatically compiled and linked as appropriate, to produce an executable program.

During testing, the ATAC runtime routine, invoked from the modified program, records dataflow coverage execution slices for use in the analysis phase. In the analysis phase, the tester can request coverage values for any of the dataflow coverage measures, display source code constructs not covered by the test cases, or obtain various other analyses of the coverage data. Blocks not covered are displayed in a context of surrounding source code. Other constructs are also displayed by highlighting the constructs not covered in their context.

ATAC uses. ATAC can be used in several ways in the software improvement process.

Measure test-set or session completeness. ATAC's measures of test completeness give an objective measure of how completely a program or routine has been tested. This measure is useful in evaluating test quality and program correctness. A low coverage score indicates that the test cases do not effectively exercise the program; a high score establishes a degree of confidence that the program, by passing the tests, works correctly.

Assist in generating and creating new test cases. ATAC can be used to create new test cases in two ways. The first way is to use ATAC as a source code browser to highlight code or dataflow associations that have not been executed. With this aid, the programmer can examine the code and create test cases for these, as yet uncovered, constructs. After running the additional test cases, the programmer can see which constructs are newly covered and examine the remaining uncovered constructs. This is a simple — though possibly time consuming — exercise for block and decision coverage. (It's

Using ATAC

We illustrate the use of ATAC on the sample program sum.c of Figure 1. *Sum* computes the sum or product of integers from 1 to *n*, depending on the inputs. We compile and link a program using ATAC in place of the standard compiler and linker.

```
>atacCC –o sum sum.c
```

ATAC creates an instrumented, executable program in sum and dataflow tables in sum.atac. During program execution, ATAC's runtime routine collects execution path information into sum.trace without interfering with the program's usual behavior. We invoke sum to input the values 5 and 0.

```
>sum
Enter an integer and 0 for +, 1 for *: 5 0
n = 5, sum = 15
```

The correct output for $n = 5$ has been calculated, and we examine the coverage achieved on sum.c by this first test case.

```
>atac –s sum.trace sum.atac
% blocks    % decisions    % C-Uses    % P-Uses
90(9/10)    80(3/5)        50(3/6)     75(3/4)     == total ==
```

We see that nine of 10 blocks have been covered, and we can ask ATAC to display the uncovered block with > atac –mb sum.trace sum.atac. The result is in Figure A.

Now, we test the functionality of sum on other inputs in the hope of achieving fuller coverage.

```
>sum
Enter an integer and 0 for +, 1 for *: 5 1
n = 5, prod = 120
>sum
Enter an integer and 0 for +, 1 for *: 0 0
n = 0, sum = 0
```

The results are in accord with our functional expectations. Now we ask ATAC how we are doing in coverage.

```
>atac –s sum.trace sum.atac
% blocks    % decisions    % C-Uses    % P-Uses
100(10)     100(5)         83(5/6)     100(4)      == total ==
```

We see that we have not covered one of the six c-uses in sum.c. We can ask ATAC to display that uncovered c-use.

```
>atac -mc sum.trace sum.atac
```

The result is in Figure B.

We test sum again on yet another input to cover the c-use.

```
>sum
Enter an integer and 0 for +, 1 for *: 0 1
n = 0, prod = 1
>atac –s sum.trace sum.atac
% blocks    % decisions    % C-Uses    % P-Uses
100(10)     100(5)         100(6)      100(4)      == total ==
```

We have now achieved complete coverage of sum.c. The mixture of functional and coverage testing has revealed that the four tests completely cover sum.c on ATAC's measures and that sum.c is correct on the tests so far. Notice that this testing tells us nothing about whether sum.c was supposed to work for negative integers. An additional test shows that it does not.

```
>sum
Enter an integer and 0 for +, 1 for *: -5 0
n = -5, sum = 0
```

This underscores the importance of functional testing in conjunction with ATAC coverage testing. Our coverage testing has thoroughly examined the circuitry of sum.c. Such complete coverage testing is easy for such trivial programs as sum.c. Full coverage is much more difficult for large and complex programs.

```
>atac -mb sum.trace sum.atac
------------------> sum.c:main 1 of 10 blocks not covered <-----

                prod *= i;
                i++;
    }

        if(k == 0)
        printf("n = %d, sum = %d\n", n, sum);
        if(k == 1)
        [redacted];
}
>
```

Figure A. The block not yet covered in sum.c.

```
---------------> C-USE of [redacted] in main; sum.c line 7 <----------

            printf("Enter an integer and 0 for +, 1 for*: ");
            scanf("%d %d", &n, &k);

            sum = 0;
[redacted]  [redacted];
            i = 1;
            while (i <= n)
            {

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

            if(k == 0)
            printf(" = %d, sum = %d\n", n, sum);
            if(k == 1)
[redacted]  printf("n = %d, prod = %d\n", n, [redacted]);

    }
>
```

Figure B. The missing c-use in sum.c.

somewhat more challenging for c-use and p-use coverage.) Since a thorough job of unit testing can vastly reduce the overall cost of testing a software system, this use of ATAC is well worth its cost. We find that the visual feedback motivates the programmer to pursue higher levels of unit testing coverage.

The second way to use ATAC is for effective selection of randomly generated tests. For many applications, test cases can be generated automatically. However, practical use of these tests requires either a correctness oracle or a mechanism for selecting an effective, small subset of the many test cases generated. ATAC coverage measures, or any coverage measures that can be computed automatically, provide a basis for such a test selection mechanism. Using coverage measures computed by ATAC, automatically generated test cases can be selected on the basis of coverage improvement. Unselected test cases don't have to be evaluated for correctness (a costly business), and the final number of test cases is usually much smaller than the total number generated.

For example, we used ATAC as a coverage oracle to cull good coverage tests for a Unix sort program. By seeding the test generator with functional tests and an input syntax for sort.c, we generated sets of 1,000, 10,000, and 100,000 tests. We culled the duplicate and useless tests, finding 27 of the first 1,000 and seven of the next 10,000 that improved coverage, but only two of the last 100,000 that improved coverage. The resultant 36 tests gave reasonable coverage of the approximately 900-line sort program. The process ran overnight unattended; thus, the real cost in human and computer time was small.

Assist in manual detection of faults via code inspection. ATAC coverage displays are effective aids in fault detection via manual inspection of source code. With them, programmers can focus on poorly covered sections of code that may be difficult to reach in the unit test environment. Often, while using ATAC to create additional test cases, programmers notice an unexpected pattern of coverage that leads directly to detection of a program fault.

The data collected by ATAC can be used to locate a fault responsible for an error detected by one or more test cases. The code executed by a test can be thought of as its execution slice. When a test fails, the fault causing the failure

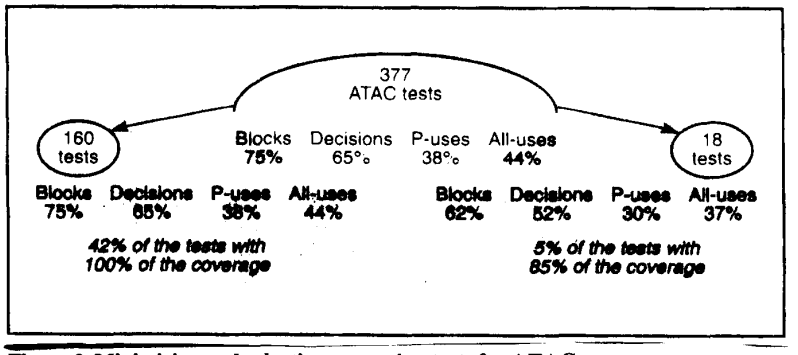


Figure 3. Minimizing and selecting regression tests for ATAC.

must be somewhere in that slice. If many tests fail, apparently due to the same fault, the fault causing the failure is probably in the intersection of the execution slices of those test cases. When a program containing a fault passes a test, the execution slice for that test may or may not contain the fault. (The faulty code may have executed in such a way that it does not adversely affect program behavior for this test.) When the execution slices for successful tests are subtracted from the intersection of slices for failed tests, the remainder is apt to contain the fault. By computing and displaying the resulting code fragments, ATAC helps programmers locate the code fault. The same technique, using appropriate weighting of successes and failures, can be used in the face of multiple code faults.

Locate faults and features based on execution slicing. In maintaining large software systems, sometimes it's necessary to locate, in unfamiliar code, the sections that implement a particular application feature. ATAC's fault-location technique can also be used to locate other code features. For example, given an association of each test with the application features it exercises (that is, what it does), ATAC can combine execution slices to determine which code sections implement each feature.

Prioritize test cases for regression testing. The test cases run over the life of a program are often collected as a regression test set. For each program modification, the regression test set is rerun to verify that the modifications have not adversely affected program behavior. At some point, it becomes impractical to run the entire set of tests for small program modifications. ATAC uses the execution slice of each test to determine a minimum set of regression test cases to achieve a given level of coverage. If the cost of executing test cases varies, costs can be assigned to each test so that ATAC will provide a minimum-cost set of test cases or a cost-effective ordering of test cases. This technique can identify tests that add no coverage to the regression tests and are therefore candidates for deletion. Tests that must be retained can be assigned a cost of zero so that they always remain cost-effective.

Figure 3 shows the application of these techniques to some ATAC regression tests. In building the regression suite, we found that 377 tests could be minimized to 160 tests with no loss in coverage and to just 18 tests with some loss in coverage. Such information is useful when we must retest under time pressure.

Provide data for performance, risk, and reliability analysis. In addition to coverage information, ATAC collects the number of executions of a covered code construct. This data can be viewed in the source code browser, with a color spectrum indicating frequency of execution. ATAC also uses this data to assign a risk measure to source code fragments. The risk measure, or the likelihood of faults in a section of code, is based on local code complexity, coverage, and execution frequency.

Software reliability analysis uses statistical techniques, based on the pattern of failures from previous tests, to predict the number of error-causing faults remaining in software. The analysis assumes uniform variability in the test sequence; in particular, repeated execution of the same test should not appear the same as execution of many different tests. Execution slices collected by ATAC for each test can be used to obtain a difference measure over the tests. Incorporating this measure in the software reliability analysis improves prediction accuracy.

Case study 1: U of Iowa/Rockwell joint project

Our first case study in software testing-coverage measurement comes from a real-world airplane landing system, or so-called *autopilot*, developed by 15 programming teams at the University of Iowa and the Rockwell/Collins Avionics Division. Guided by an *N*-version programming design paradigm,[8] 40 students (33 from ECE and CS Departments at the University of Iowa and seven from Rockwell International) participated in this project to independently design, code, and test the autopilot application.

Project overview. The selected application, a fault-tolerant software project, is part of a specification some aerospace companies use for computer-controlled landing of commercial airliners. The specification can be used to develop flight-control computer software for real aircraft, since it's adjustable to the performance parameters of a specific aircraft. All algorithms and control laws are specified in diagrams certified by the Federal Aviation Administration. The *pitch-control* portion of the autopilot problem — that is, the control of the aircraft's vertical motion — was selected for this project.

The software development cycle was conducted in several software engineering phases, including the Initial Design, Detailed Design, Coding, Unit Testing, Integration Testing, Acceptance Testing, and Operational Phases. Software testing was a major activity. In the Unit Testing (UT) Phase, each team received sample test data sets for each module to check its basic functionality. A total of 133 data files (roughly equivalent to one execution of the completely integrated program) was provided in this phase. In the Integration Testing (IT) Phase, four sets of partial flight-simulation test data, representing 960 complete program executions, were provided to each programming team. This phase of testing was intended to guarantee the software's suitability for a flight simulation environment in an integrated system.

Finally, in the Acceptance Testing (AT) Phase, programmers formally submitted their programs for an acceptance test. In the acceptance test, each program was run in a test harness of flight simulation profiles for both nominal and difficult flight

Table 1. Fault distribution of each program by phases.

Test Phase	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Total
Unit Testing (UT)	2	2	3	1	3	3	5	3	2	1	2	2	29
Integration Testing (IT)	4	3	4	4	1	0	3	2	2	2	3	1	29
Acceptance Testing (AT)	2	2	3	4	3	4	1	3	5	2	5	3	37
Operational Testing (OT)	1	0	0	0	0	0	0	0	0	0	0	0	1
Total	9	7	10	9	7	7	9	8	9	5	10	6	96
Original fault density	2.2	5.7	11.2	9.7	4.7	5.9	7.2	3.2	7.7	4.7	5.9	4.4	5.1
Fault density after AT	0.2	0	0	0	0	0	0	0	0	0	0	0	0.05

conditions. When a program failed a test, it was returned to the programmers for debugging and resubmission, along with the input case on which it failed.

More than 21,000 different program executions were imposed on these programs before final acceptance. Twelve of the 15 programs passed the acceptance test and went to the Operational Testing (OT) Phase for further evaluations. Program size ranged from 900 to 4,000 uncommented lines of code, with an average of 1,550 lines.

Program metrics and statistics. A total of 96 faults were found and reported during the project's life cycle. Table 1 shows the distribution of the detected software faults in the 12 accepted programs (identified by Greek letters) with respect to each test phase. The fault densities (per thousand lines of uncommented code) of the original version and the accepted version for each program are also shown.

Later in the operational testing phase, we conducted more than 1,000 flight simulations — over five million program executions. Only one operational fault (in the β version) was found. This implies that the program quality obtained from this project was very high. For the 12 accepted programs, the average fault density was 0.05 faults per thousand lines of code. This number is close to the best current effort in the software industry. (For a detailed report on this project, see Lyu.[4])

Testing metrics measured by ATAC. Facilitated by the ATAC tool, we further investigated the application of testing coverage metrics as a quality control mechanism. Table 2 shows the coverage obtained during each testing phase for the four coverage metrics (block, decision, c-use, p-use). It also gives the average value and the range, from highest to lowest, among the 12 programs.

Table 2 shows a number of interesting results. First, there were no strong correlations among the four program con-

Table 2. Testing-related coverage metrics measured by ATAC (Automatic Test Analysis for C).

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Average	Range
Blocks	511	711	531	554	679	537	367	1,132	542	473	457	483	581.4	3.08
Percent in														
UT	65	59	62	70	44	64	56	60	68	68	71	57	62.0	1.61
IT	85	71	77	83	74	86	79	76	80	88	86	80	80.4	1.24
AT	95	78	88	95	88	98	91	91	90	97	97	94	91.8	1.24
Decisions	216	250	320	297	520	284	286	357	264	237	231	262	293.7	2.41
Percent in														
UT	36	37	37	43	27	28	33	29	42	41	42	32	35.6	1.59
IT	71	73	63	67	60	72	69	62	63	78	72	66	68.0	1.30
AT	88	87	78	82	77	90	82	78	79	92	89	86	83.9	1.19
C-uses	935	755	395	696	1027	636	710	965	727	537	803	665	737.6	2.60
Percent in														
UT	60	57	56	50	45	57	44	69	56	55	56	55	55.0	1.57
IT	83	76	80	67	70	81	72	84	74	81	78	82	77.3	1.25
AT	96	90	96	84	87	95	87	96	86	96	93	94	91.7	1.14
P-uses	413	340	349	520	611	463	459	419	355	310	279	392	409.2	2.19
Percent in														
UT	30	34	38	32	26	22	23	37	42	36	38	29	32.3	1.91
IT	66	60	63	47	58	59	49	61	59	68	64	61	59.6	1.45
AT	84	72	78	58	74	72	57	71	72	85	80	79	73.5	1.49

structs. For example, program β's block and p-use values were average, but it had the smallest decision value and a very high c-use value. We also noticed that the equivalence of one complete program execution in UT exposed a variety of effects on different program constructs of different program versions, which contained a fairly large range of coverage in blocks (44-71 percent), decisions (27-43 percent), c-uses (44-69 percent), and p-uses (22-38 percent). Moreover, the coverage of blocks and c-uses is higher than the coverage of decisions and p-uses.

We further observed that the programs tested with fairly high quality. In particular, some programs achieved acceptance test coverages as high as 98 percent of blocks, 92 percent of decisions, 96 percent of c-uses, and 85 percent of p-uses. Although some programs had consistent scores, others did not. For example, version ν had very high values in all measures, and version ζ had the lowest value in both c-uses and p-uses. Version θ, on the other hand, had the highest block value and very high decision and c-use values, but it had a relatively low percentage of p-uses.

As the number of program executions increased, test quality improved, and the range of coverage percentages decreased. Nevertheless, considering that these coverage results were obtained from programs of *the same application* tested by *the same data*, the differences in these measures seemed significant (for example, version θ obtained 98 percent block coverage while version γ obtained only 78 percent). On the other hand, we also noticed a diminishing return on coverage after the acceptance test, and the operational test data (five million program executions) did not increase this coverage significantly. This meant that the 22 percent of uncovered code in version γ was probably not even executed during the operational phase.

Figure 4 summarizes the increase of software coverage metrics, measured by averaging coverage data from the 12 programs, achieved as testing progressed from UT (one program execution) to IT (960 executions) and AT (21,000 executions). As expected, the first execution hit a large area of the programs, but the coverage measures increased monotonically with the number of test cases. The amount of the increase, however, declined with the addition of more test cases and finally reached a plateau.

Table 3 summarizes the effectiveness of the three testing phases. In particular, it lists the percentage of known faults detected up to a certain test phase. In this table, we see that the coverage obtained from AT was satisfactory, but that obtained from UT and IT was not. Achieving a higher level of coverage (for example, from 80 to 90 percent block coverage) proved to be a crucial step toward quality and reliability (from 60 to 99 percent fault detection) among the investigated programs.

We suspect there's a correlation between the number of faults detected in a version and the coverage of its program constructs. In theory, the better a program is covered during testing, the more faults will be detected. However, we did not see strong correlations between the total faults detected in the program versions (Table 1) and their coverage measures during various testing conditions (Table 2). It may be that, because each version has a different fault distribution to begin with, the coverage measures are not good predictors of the absolute number of program faults. Besides, the number of faults detected in each version is not very large, which may reduce the statistical significance of the analysis.

ATAC's ability to highlight noncovered program code permits detailed examination of construct coverage, thereby revealing programming style and program testability. In version γ, for example, we noticed that an untested error-handling function accounted for 10 percent of the total blocks; in most other versions, the same function accounted for only 1 to 2 percent of block coverage. Further examination showed that version γ used numerous function calls to pass parameters, and each function call was counted as an uncovered block. This clearly indicated the need for an extra test case to increase version γ's block coverage.

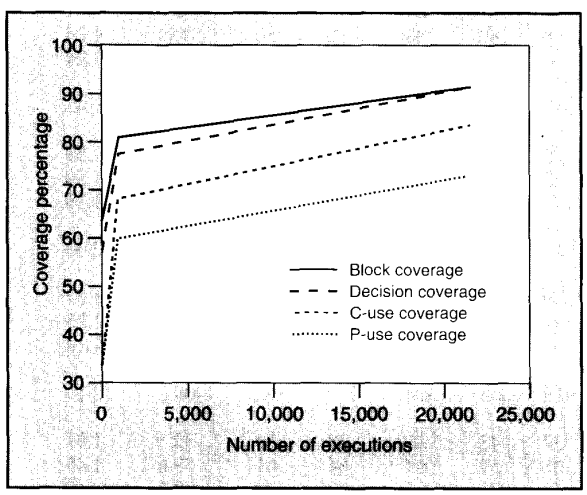


Figure 4. Increase in coverage as testing progresses.

Case study 2: A Bellcore project

A central question of coverage testing is whether there is an exact "dose-response" relationship between the percentage of coverage and the number of faults in a software system. The analogy is to pharmacology, which attempts to calibrate patient response to a particular quantity of medicine. Ideally, the testing manager would have tables relating "bug killing" capacity to coverage testing level.

Two years ago, in a retrospective study of Bellcore production software, we addressed the dose-response question. The idea was simple. First, we would find a production system that had carefully preserved versions of codes, tests, and failure reports (called modification requests or MRs), and we would retrieve them for each phase. Then, test coverage would be assessed using ATAC, faults would be assigned to modules, and the relationship between percent of coverage and number of faults could be determined.

The system we studied consisted of approximately 60,000 lines of code in 60 modules. We soon encountered significant difficulties in conducting the study. The versioning system and the MR system had not been designed to facilitate a retrospective study. Therefore, assigning faults described in MRs to the correct module required great care and consid-

erable knowledge of the system. We determined that an analysis could be performed on only half of the system. Ultimately, we performed the study on 33 modules, their unit tests, and 35 system test MRs. Because several MRs could not be accurately assigned, several modules might have been assigned one of these ambiguous MRs or no MRs at all. These modules and MRs are eliminated from the data we report here.

Table 3. Testing coverage measures and known faults detected during testing.

Testing Phase	Number of Tests	Percent Block	Percent Decision	Percent C-Uses	Percent P-Uses	Percent Detected
Unit Testing	1	62.0	35.6	55.0	32.3	30.2
Integration Testing	960	80.4	68.0	79.3	59.6	60.4
Acceptance Testing	21,000	91.8	83.9	91.7	73.5	98.9

Statistics and analysis. Figure 5a displays the achieved block coverage of unit tests for the 33 modules compared to the number of system test faults found for each module. Figure 5b plots the achieved uses coverage versus faults found in system tests.

Figure 5a plots the modules by percentage of block coverage on the *y* axis and number of system test modification requests (MRs — equivalent to a fault) on the *x* axis. We had preserved the unit modules, the tests done at unit test time, the system built for system test, and the MRs recorded during system test. With these artifacts, we were able to trace the MRs to the modules with the associated faults. For instance, we found 13 modules with no MRs and one module with 6 MRs.

From these data on this single experiment, we cannot conclude anything about a dose-response relationship. However, from the data presented in Figure 5a, we can safely observe that modules with high block coverage (70 percent and above) are free of MRs in system test. This simple observation is in accord with the report of Piwowarski, Ohba, and Caruso.[9] That study of several large IBM software systems found a precise relationship between fault density and statement coverage (virtually identical to block coverage). We believe that such results will be possible when coverage is the goal during testing. Our study assessed coverage after testing. The testers were unaware of the level of testing and had no coverage goals. Observations similar to those for block coverage can be made for all-uses coverage.

In this single study, there is a clear relationship between high statement coverage in unit testing and low system test faults, and we allow the reader to draw parallels between the different measures of coverage and MRs. The conclusion that MRs decrease with higher coverage seems sound if each module is regarded as a function point. However, if we adjust for "size" (for example, dividing MRs by the number of blocks in a module), this apparent result is suspect.

Nonetheless, it is commonly accepted that less than 70 percent block coverage does not assure good testing. We therefore prefer to view these data as weakly supporting the hypothesis that high coverage tends to reduce faults. Further experimentation on the dose/response relationship (if any) between coverage testing and fault elimination is underway in more controlled experiments. The final judgment on the value of coverage testing as a fault purgative will come only with use of coverage testing in standard software development.

Secondary study. The difficulties encountered in this study led us to attempt

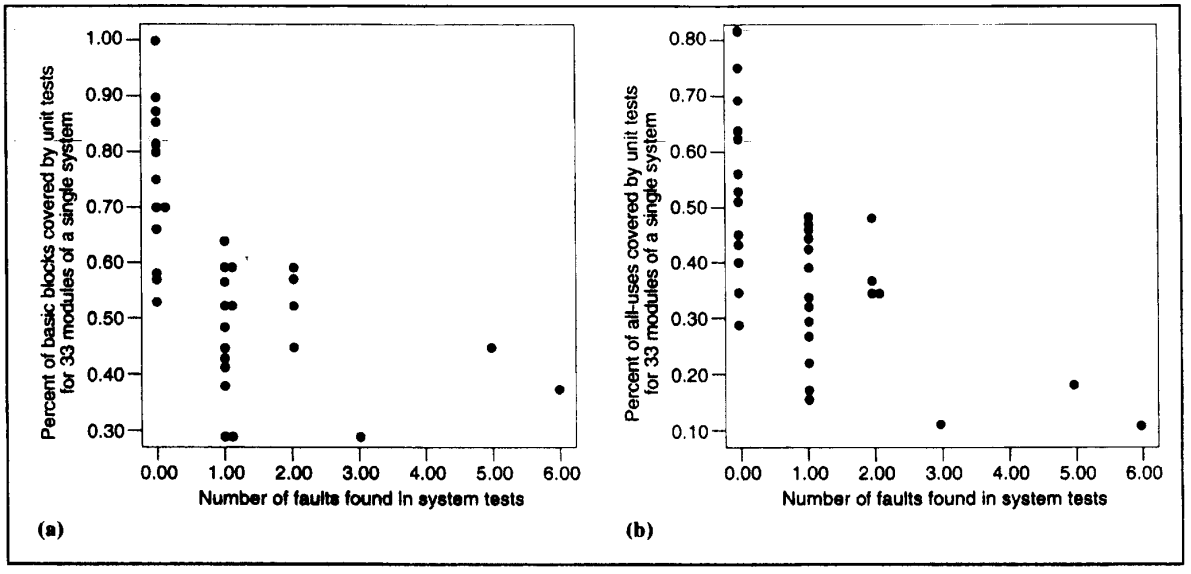


Figure 5. System testing faults versus block coverage (a) and all-uses coverage (b).

tion of a deliverable or salable product. In ... yields better ... suggest that

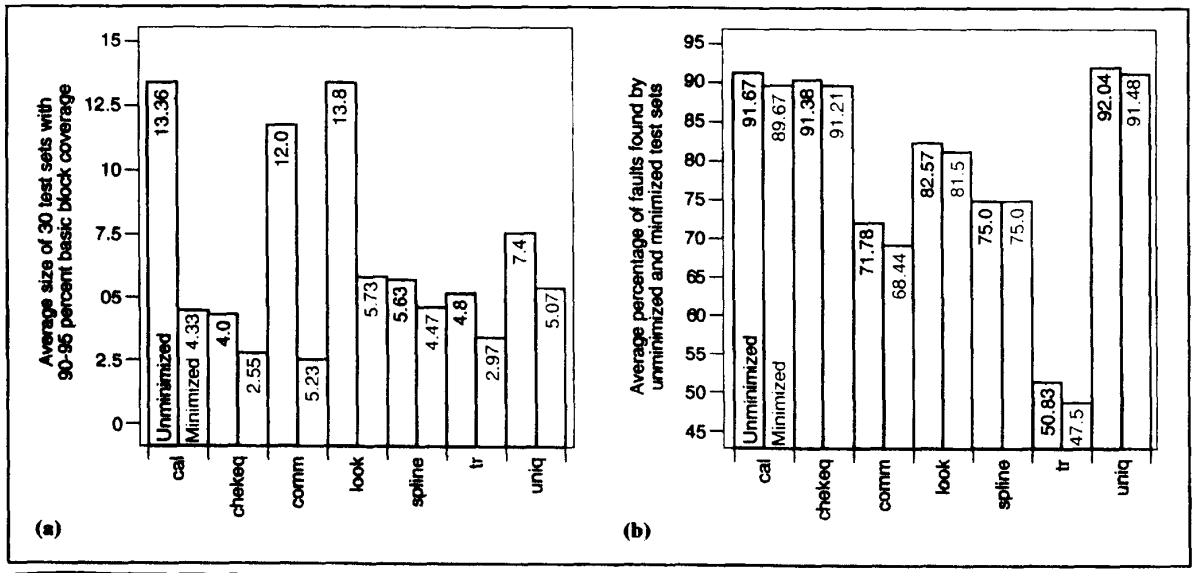


Figure 6. Unminimized and minimized tests: (a) average size; (b) average effectiveness.

a more controlled study on smaller programs with artificially seeded, but realistic, bugs.[10] Figure 6 summarizes some of our findings. For each of the seven standard Unix programs, cal, checkeq, comm, look, spline, tr, and uniq, we generated 30 test sets with 90 to 95 percent block coverage. We then used ATAC to select a minimal test set with the same coverage for each program.

The average test set sizes are represented in Figure 6a. Clearly the minimized test sets are, on average, substantially smaller. Figure 6b shows the average number of seeded bugs found by the test sets and by their minimized counterparts. While the minimized test sets are substantially smaller, they are only marginally less effective in finding bugs. This leads us to conclude that it's the coverage rather than the number of tests that is detecting the bugs. A fuller study might establish a dose-response between degree of coverage testing and bug detection.

The use of coverage testing in the software process can be twofold. First, coverage can be taken as a measure of testing quality. It is not uncommon to find that testing considered to be thorough and complete is not very complete from the coverage point of view. Coverage measurement allows the manager to set repeatable and objective targets for testing quality. Second, coverage is an excellent feedback mechanism for the software engineer. An examination of Figure 5a reveals that one module had six MRs and was block covered to less than 40 percent during unit testing. Such data can focus testing effort on faulty and poorly covered modules.

The ultimate question we hope to answer is central to software engineering: "When is a program considered acceptable?" Many software reliability models have been proposed to answer this question.[11] However, few researchers[9,12] address the relationship of reliability to program structure or testing coverage. Investigating the quality of dataflow testing and the subsequent detection of field faults may lead to a new understanding of this relationship — one that combines testing methodology and reliability theory to address the program acceptance problem. ■

Acknowledgments

Our colleagues H. Agrawal and E.W. Krauser have contributed substantially to the work reported here. A.O. Olagunju of Delaware State College, with the aid of Bellcore's T.K. Ramaprasad, L.W. Smith, and E.I. Yang, conducted the experiment that yielded the data in Figure 5.

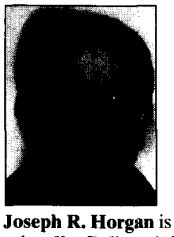
References

1. R.A. DeMillo et al., *Software Testing and Evaluation*, Benjamin, Menlo Park, Calif., 1987.
2. P.G. Frankl and E.J. Weyuker, "An Applicable Family of Dataflow Testing Criteria," *IEEE Trans. Software Eng.*, Vol. SE-14, No. 10, Oct. 1988, pp. 1,483-1,498.
3. J.R. Horgan and S.A. London, "A Dataflow Coverage Testing Tool for C," *Proc. Symp. on Assessment of Quality Software Development Tools*, IEEE CS Press, Los Alamitos, Calif., Order No. 2620, 1992, pp. 2-10.
4. M.R. Lyu and Y. He, "Improving the *N*-Version Programming Process Through the Evolution of a Design Paradigm," *IEEE Trans. Reliability*, Vol. 42, No. 2, June 1993, pp. 179-189.
5. M.R. Lyu, J.R. Horgan, and S. London, "A Coverage Analysis Tool for the Effectiveness of Software Testing," *Proc. Fourth Int'l Symp. Software Reliability Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 4010, 1993, pp. 25-34.
6. W.E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.
7. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Dataflow Information," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 4, Apr. 1985, pp. 367-375.
8. M.R. Lyu and A. Avizienis, "Assuring Design Diversity in *N*-Version Software: A Design Paradigm for *N*-Version Programming," *Dependable Computing and Fault-Tolerant Systems*, J.F. Meyer and R.D. Schlichting, eds., Springer-Verlag, New York, 1992, pp. 197-218.
9. P.M. Piwowarski, M. Ohba, and J. Caruso, "Coverage Measurement Experience During Function Test," *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 3700, 1993, pp. 287-301.

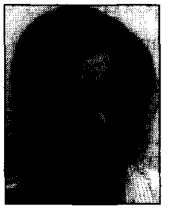
10. W.E. Wong et al., "Effect of Test Set Minimization on the Error Detection Effectiveness of the All-Uses Criterion," Tech. Report, Purdue University, West Lafayette, Ind., SERC-TR-152-P, 1994.

11. M.R. Lyu and A. Nikora, "Using Software Reliability Models More Effectively," *IEEE Software*, Vol. 9, No. 4, July 1992, pp. 43-52.

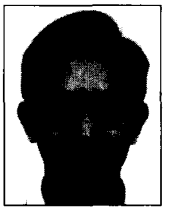
12. M. Chen et al., "Time/Structure-Based Model for Estimating Software Reliability," Tech. Report, Purdue University, SERC-TR-117-P, 1992.



Joseph R. Horgan is a member of the technical staff at Bellcore's Information Sciences and Technologies Research Laboratory. His research is in software analysis, testing, and reliability. Before joining Bellcore in 1983, he was with AT&T Bell Labs and on the computer science faculty at the University of Kansas. He has also worked at the University of Delaware and IBM. Horgan received a BA and MA in philosophy from the University of Delaware and a PhD in computer science from the Georgia Institute of Technology



Saul London is a member of the technical staff at Bellcore's Information Sciences and Technologies Research Laboratory. His research interests include software testing, programming languages, software reuse, and telecommunications software. London received his BA in mathematics from Columbia University in 1980 and his MS in computer science from New York University in 1982.



Michael R. Lyu has been a member of the technical staff in the Information Sciences and Technologies Research Laboratory at Bellcore since 1992. His research interests include software engineering, software reliability, and fault-tolerant computing. He is the editor of two books: *McGraw-Hill Software Reliability Engineering Handbook* (to be published in February 1995) and *Software Fault Tolerance* (to be published by Wiley in October 1994).

Lyu received his BSEE in 1981 from the National Taiwan University, his MS in electrical and computer engineering in 1984 from the University of California, Santa Barbara, and his PhD in computer science in 1988 from the University of California, Los Angeles.

Readers can contact the authors at Bellcore, 445 South St., Morristown, NJ 07960-6438. Lyu's e-mail address is lyu@bellcore.com.