A Coverage Analysis Tool for the Effectiveness of Software Testing

Michael R. Lyu
Bell Communications Research, Morristown
J. R. Horgan
Bell Communications Research, Morristown
Saul London
Bell Communications Research, Morristown

Key Words — **Data-flow testing, Test effectiveness, Testing coverage tool, Software metric, *N*-Version programming.**

Reader Aids —
General purpose: Describe an analytic tool and its experimental results
Special math needed for explanations: None
Special math needed to use results: None
Results useful to: Software testers & reliability-engineers

Summary & Conclusions — **This paper describes the software testing & analysis tool, "ATAC (Automatic Test Analysis for C)", developed as a research instrument at Bellcore to measure the effectiveness of testing data. It is also a tool to facilitate the design & evaluation of test cases during software development. To demonstrate the capability & applicability of ATAC, we obtained 12 program versions of a critical industrial application developed in a recent university/industry *N*-Version Software project, and used ATAC to analyze & compare coverage of the testing on the program versions. Preliminary results from this investigation show that ATAC is a powerful testing tool to provide testing metrics and quality control guidance for the certification of high quality software components or systems.**

In using ATAC to derive high quality test data, we assume that a good test has a high data-flow coverage score. This hypothesis requires that we show that good data-flow testing implies good software, *viz*, software with higher reliability. One would hope, for example, that code tested to 85% *c*-uses coverage would have a lower field-failure rate than similar code tested to 20% *c*-uses coverage. The establishment of a correlation between good data-flow testing and a low (or zero) rate of field failures is the ultimate & critical test of the usefulness of data-flow coverage testing. We demonstrated by ATAC that the 12 program versions obtained from the U. of Iowa & Rockwell NVS project (a project that has been subjected to a stringent design, implementation, and testing procedure) had very high testing coverage scores of blocks, decisions, *c*-uses, and *p*-uses. Results from the field testing (in which only one fault was found) confirmed this belief.

The ultimate question that we hope ATAC can help us answer is a typical question for all software reliability engineers: "When is a program considered acceptable?" Software reliability analysts have proposed several models to answer this question. However, none of these models address the issues of program structure or testing coverages, which are important in understanding software quality.

1. INTRODUCTION

Software reliability analysts have traditionally proposed several models [1 - 3] to help decide when a software program is ready to be released. However, these methods have typically not considered the program structure. ATAC (Automatic Test Analysis for *C*) is a software tool which considers program structure and supports data-flow coverage testing for *C* programs [4,5,20]. Coverage testing, 1) helps the tester create a thorough set of tests, and 2) gives a measure of test completeness. Each of the structural coverage criteria proposed in the literature [6 - 8] attempts to capture some important aspect of a program's behavior. Rapps & Weyuker [8] define a family of data-flow coverage criteria for an idealized programming language. Frankl & Weyuker [9] extend these definitions to a subset of PASCAL and describe a tool, ASSET, to check for test completeness based on the data-flow coverage criteria. We have adapted these data-flow coverage definitions to define realistic data-flow coverage measures for *C* programs.

The concepts of coverage testing are well-described in the literature, but there are few tools that actually implement these concepts for common programming languages [9,10]. Even less evidence is found of the application of these concepts to realistic projects in obtaining meaningful results. ATAC is a data-flow testing tool which, to our best knowledge, incorporates the most complete set of coverage measures for any common language. To investigate ATAC in a realistic project, we apply the tool to the 12 program versions developed by a recent university/industry joint project [11]. This project started as an *N*-version programming investigation on a critical automatic flight control application. We consider the multiple program versions obtained from the project as an abundant resource for studying testing-coverage and quality-metrics. ATAC facilitates this study. Preliminary results showed that, by using ATAC to analyze coverage of programs during testing, various program execution aspects are revealed easily. Not only is an indicator of testing quality revealed, but the nature of program structure, or its testability (whether a program is easy to test or not), becomes visible through the resulting measures. As shown by the comparisons among the multiple program versions tested with the same set of data, the structure of different programs can have a major impact upon the faith in testing upon these programs.

Section 2 presents ATAC in terms of its purpose, its implementation, and its uses. Section 3 describes a recent industrial project to obtain 12 program versions for a critical flight software system. Section 4 discusses some experience & results obtained in applying ATAC to the final program versions obtained in the project.

We intend later to investigate the relationship between the quality of data-flow testing and the subsequent detection of field

0018-9529/94/$4.00 ©1994 IEEE

faults, and hopefully, a unified technique combining testing methodology and reliability theory could emerge to address the program acceptance problem. We believe that ATAC can facilitate software reliability researchers & practitioners to establish the relationship in between structure-based testing schemes and software reliability measurement techniques.

Acronyms & Abbreviations

ATAC	Automatic Test Analysis for *C*
c-use	computational-variable use
p-use	predicate-variable use
NVP	*N*-version programming
U/Iowa	University of Iowa.

2. ATAC SOFTWARE-COVERAGE TOOL

ATAC is a tool for evaluating test set completeness based on data-flow coverage measures. ATAC allows the programmer to create new tests intended to improve coverage by examining code not covered. The steps to use ATAC are:

- Prepare a program for testing with a preprocess-compile-link phase. This creates an instrumented object module and data-flow tables used during run-time.
- Run tests and collect trace & coverage data with the ATAC run-time routine.
- Execute an analysis phase which provides feedback on the tests that have been run.
- Use ATAC to browse the code not covered. This allows the programmer to understand the incompleteness of the tests and to design new tests that enhance coverage.

The ATAC preprocessor analyzes *C* source code and produces a file containing data-flow information about the source program for use in the analysis phase. The preprocessor also creates a modified version of the source code instrumented with calls to the ATAC run-time routine.

During testing, the ATAC run-time routine, invoked from the modified program, maintains a compact coverage trace for use in the analysis phase. In the analysis phase, the tester can:

- request coverage values on the preceding test for any of the data-flow coverage measures,
- display source code constructs not covered by the tests.

Blocks not covered are displayed in a context of surrounding source code. Other constructs are also displayed by highlighting the constructs not covered in the context of their surrounding code.

Coverage analysis can be performed for each *C* function, for each test, or some combination of tests and *C* functions. 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. However, for very large programs, testing can be constrained by, 1) available memory and disk space, and 2) test execution time. The program-constructs measured by ATAC include blocks, decisions, *c*-uses, and *p*-uses.

- *Block coverage* counts the branch free executable code fragments that are exercised at least once. A block can be more than one *C* statement if there is no branching between statements. A statement can contain multiple blocks if there is branching inside the statement. An expression can also contain multiple blocks if branching is implied in the expression (*eg*, a conditional expression or logical-and or logical-or expression). If block coverage is less than 100%, then some statements are not exercised by any test.
- *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 can not be revealed. Completely adequate decision coverage implies completely adequate block coverage except for functions with no branches.
- *c*-use coverage counts the number of combinations of an assignment to a variable and a use of the variable in a computation that is not part of a conditional expression. Since functions & statements need not use or assign any variables, *c*-use coverage is not comparable to most of the other measures.
- *p*-use coverage counts 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 & *p*-use coverage is that when a variable can be assigned a value in more than one way, a good test set insures 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 (*eg*, "`while (getchar() != '\\n');`").

2.1 Purposes of ATAC

ATAC can achieve the following objectives in the software testing process:

1. objectively measure test-set or test-session completeness,
2. display non-covered code to aid in test creation,
3. select effective randomly-generated tests,
4. reduce regression test-set size by eliminating redundant tests.

Objective #1 is useful in evaluating the quality of the testing procedure, and in establishing a level of assurance in the quality of tested programs. A low coverage-score indicates that the tests do not effectively exercise the program. A high coverage-score establishes faith that the program, in passing the tests, works correctly.

Objective #2 is a programmer aid for unit testing. Since a thoroughly-done unit-testing job can vastly reduce the overall cost of testing a software system, a programmer can use the coverage displays to reveal code constructs that have not been covered by unit testing. By examining the code, the programmer can discover tests that will cause these, as yet not covered, constructs to be covered. After running these additional tests, the programmer can check which constructs are newly covered, and examine the remaining non-covered constructs.

Objective #3 provides a mechanism for determining an effective, small subset of the many automatically generated tests. For many applications it is possible to generate tests automatically [12 - 14]. Coverage-measures provide a basis for such a mechanism. While the number & complexity of data-flow objects associated with a program can pose a problem to the programmer trying to devise tests for the program, we see no similar problem in our use of automatic test-selection.

Objective #4 relates to the tests run over the life of a program that are collected into a regression test set. The regression test set is re-run each time the program is modified, to verify that the modifications have not adversely affected the program. A regression test set can grow so large that it is not practical to run the whole set of tests after small program modifications. Hence, this objective uses the coverage measure to select a subset of the regression tests which can achieve a high level of coverage. This technique can identify tests that add no coverage at all to the regression tests, and are therefore candidates for deletion.

2.2 Design & Implementation of ATAC

ATAC is implemented as 5 *C* programs consisting of about 36K lines of source code, several shell scripts, and a run-time routine. ATAC is in its third incarnation. Version #1 analyzed *C* properly but only did block-coverage. Version #2 was rather complete but consisted of over 50K lines of poorly engineered code. Version #3 (current) is well-engineered and is designed to accommodate changes & extensions. ATAC is running in a variety of UNIX environments (Sun 3, Sun SPARC, Dec 3100, Vax 8650, Pyramid, *etc*) in several Bellcore divisions and at Purdue University. It is reasonably easy to port & install. ATAC has been run successfully on programs up to 100K lines. Disk-space utilization is, in our experience, less than $(3+n)$ times the space needed for a "debug" (-g) version of the test program (n = number of test runs). Execution time can increase appreciably (in one case by a factor of 36), but usually less than a factor of 2 and commonly 20-30%. ATAC has never exceeded available memory on an 8 MB system.

2.2.1 Preprocessor

The ATAC preprocessor is the heart of ATAC. The preprocessor parses & analyzes *C* source code, and outputs, 1) an instrumented version of the source code, and 2) a file containing static data-flow information. The *C* parser was originally part of a language-based editor for *C* [15]. A parser generated by the YACC [16] parser generator tool creates an abstract syntax tree in memory for each *C* function. A data-flow graph is created for this syntax tree using well known techniques. A table of DEF/USE information is generated from the data-flow graph to be included with the instrumented source code. In order to instrument the source code, a mark is placed in the syntax tree for each node in the flow graph. The syntax tree is then deparsed to create the instrumented source code. Marks in the syntax tree are translated to calls to the ATAC run-time routine. Each call to the ATAC run-time routine contains a block number and a pointer to the current context. The context contains, 1) pointers to the DEF/USE tables, 2) information about constructs already covered, and 3) dynamic function call level.

To create the static data-flow information file, the data-flow graph is searched for data-flow coverage constructs that might be covered during testing. These constructs are saved in a file with their original source-code positions.

2.2.2 Source Code Positions

In order to display non-covered constructs in the source code, it is necessary to store original source-code positions in the static data-flow information file. This is complicated by the presence of *C* preprocessor macros and include files which are expanded before the ATAC preprocessor reads the code. The standard *C* preprocessor inserts line directives in the preprocessed code to reveal the original source-file name and line number of included code. However, this does not help with macros which expand within a given line. To handle this problem we have modified a *C* preprocessor to insert an escape sequence (into the code) indicating which text is part of a macro expansion and the size of the original text. The ATAC preprocessor decodes these escape sequences and the line directives to get original source-code positions which are saved in the static data-flow information file.

2.2.3 Run-time Routine

The ATAC run-time routine recognizes data-flow constructs during execution of a test, and notes the first occurrence of each construct in the trace file. The mechanism for this is simple. For DEF/USE constructs it proceeds as follows.

- Tables generated during source-code instrumentation indicate which variables are defined & used in a given block.
- The run-time routine keeps track of each variable that has been defined and the block at which it was defined.
- When a block that uses a defined variable is encountered, the definition & use are recorded in the trace file. Multiple occurrences of the same DEF/USE pair are not recorded.
- Because a single procedure can be invoked recursively, the run-time routine maintains a separate list of defined variables for every active procedure being tested. When the final block of a procedure is executed, the list of defined variables for that procedure is freed.
- The methods for recording the other constructs (decisions, *c*-uses, *p*-uses) are similar.

2.2.4 Analysis

The ATAC analysis program reads, 1) the static data-flow information for each source file being tested, and 2) the trace file from the execution of the tests. Constructs in the trace file are matched with constructs in the data-flow information file to determine, for each function, 1) the total number of constructs in the function, and 2) the number of constructs executed by the tests. The analysis results can be broken down by test files or by program modules.

2.3 Uses of ATAC

ATAC is a coverage-testing tool and does not directly aid in functional testing. Therefore, the first step in testing a program is for the tester to create tests which are intended to ascertain that the program meets the functional characteristics of the specifications. ATAC can then be used to measure the coverage of those functional tests. For example, for the 5K line Spiff program [17], the functional tests presented the following coverage profile:

blocks	decisions	*c*-uses	*p*-uses
62%	54%	46%	42%
1009/1622	471/869	1023/2242	691/1664

This means that 62% of the blocks, 54% of the decisions, 46% of the *c*-uses, and 42% of *p*-uses were covered by the tests which were deemed adequate to gauge that the program implemented the functions required in the specifications. If the tester is satisfied with this coverage, then ATAC is of no further use. Otherwise ATAC can aid in the selection of new tests.

The tester who wishes to improve coverage by hand-crafting tests can request that ATAC display the code while highlighting non-covered objects. For instance, blocks not covered are displayed in situ (as in figure 1).

```
---------> sort.c: merge 6 blocks not covered <----------

while (i > 0) {
   cp = ibuf[i-1] -> 1;
      if (!cflg && (uflg == 0 || mufl[illegible]
         do
            putc(*cp, os);
         while (*cp++ != '\n');
         if (ferror(os)) {
            error = 1;
            exit();
         }
      }
```

Figure 1. ATAC Highlighting of Non-Covered Blocks

The tester can use this display to attempt to understand why none of the tests touched the highlighted blocks. One can proceed through the code, analyzing the blocks not covered, and constructing tests which are designed to increase coverage. This is, of course, an interactive process. Tests are created & run, the coverage is checked, and the blocks not covered are redisplayed — until one is either satisfied with the coverage or convinced that no tests can be added that will cover the remaining blocks. The value of this approach, particularly in unit testing, is that hand-crafted tests can be created (by the programmer) which are aimed precisely at constructs not covered. This can lead to a very high-quality test set.

3. U/IOWA & ROCKWELL JOINT PROJECT

NVP achieves fault-tolerant software systems (*N*-version software systems) through development & use of design diversity [18]. This approach involves the statistically-independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. NVP was motivated by the "fundamental conjecture that the independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program." [18]

In late 1991, a real-world automatic (computerized) airplane landing system (autopilot) was developed & programmed by 15 programming teams at U/Iowa and the Rockwell/Collins Avionics Division [11]. 40 students[1] participated in this project to *s*-independently design, code, and test the computerized airplane landing system — as a major requirement of a graduate-level software engineering course.

3.1 Application Problem

The application in this NVP project is part of a specification used by some aerospace companies for the automatic (computer-controlled) landing of commercial airliners. The specification can be used to develop the software of a flight-control computer for a real aircraft, given that it is adjusted to the performance parameters of a specific aircraft. All algorithms and control laws are specified by diagrams which have been certified by the US Federal Aviation Administration. The pitch control part of the auto-landing problem (control of the vertical motion of the aircraft) was selected for the project.

Simulated flights begin with the initialization of the system in the Altitude Hold mode, at a point approximately 10 miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet/second). The Complementary Filters preprocess the raw data from the aircraft sensors. Pitch-mode entry & exit is determined by the Mode Logic equations, which use the filtered airplane-sensor data to switch the controlling equations at the correct point in the trajectory.

Pitch modes entered by the autopilot/airplane combination, during the landing process, are: Altitude Hold, Glide Slope Capture, Glide Slope Track, Flare, and Touchdown. The Control Law for each of them consists of two loops, outer & inner. The Altitude-Hold Control Law is responsible for maintaining the reference altitude. As soon as the edge of a glide slope beam is reached, the airplane enters the Glide-Slope Capture & Track mode and begins a pitching motion to acquire & hold the beam center. Controlled by the Glide Slope Capture and Track Control Law, the airplane maintains a constant speed along the glide slope beam. Flare logic equations determine the precise altitude (about 50 feet) at which the Flare mode is entered. In response to the Flare control law, the vehicle is forced along a path which targets a vertical speed of 2 feet/second at touchdown.

Besides computing the flight control command according to the above sequence, each program checks its final result (the pitch control command) against the results of other programs. Any disagreement is indicated by the Command Monitor output, so that a supervisory program can take appropriate action.

[1] 33 from ECE & CS departments at U/Iowa, 7 from the Rockwell International.

3.2 Software Development

There were 6 phases:

1. Initial design (4 weeks). The purpose was to allow the programmers to get familiar with the specified problem, so as to design a solution to the problem. At the end of this phase, each team delivered a preliminary design document, which followed specific guidelines & formats for documentation.

2. Detailed design (2 weeks). The purpose was to let each team obtain some feedback from the coordinator to adjust, consolidate, and complete their final design. Each team was also requested to conduct at least one design walk-through. At the end of this phase, each team delivered, 1) a detailed design document, and 2) a design walk-through report.

3. Coding (3 weeks). By the end of this phase, programmers had finished coding, conducted a code walk-through, and delivered the initial, compilable code. Each team was required to use the RCS revision control tool for configuration management of their program modules.

4. Unit testing (1 week). Each team was supplied with sample test data sets for each module to check the basic functionality of that module. They were also required to build their own test harness for this testing purpose. 133 data files were provided to the programmers.

5. Integration testing (2 weeks). Four sets of partial flight-simulation test data, together with an automatic testing routine, were provided to each programming team for integration testing. This testing phase was to guarantee that the software was suitable for a flight simulation environment in an integrated system.

6. Acceptance testing (2 weeks). Programmers formally submitted their programs for a 2-step acceptance test. In step #1 (AT1), each program was run in a test harness of 4 nominal flight simulation profiles. In step #2 (AT2), one extra simulation profile, representing an extremely difficult flight situation, was imposed. In total there were 23 930 executions imposed on these programs before they were accepted and then subjected to the final evaluation in the following stage. By the end of this phase, 12 of the 15 programs passed the acceptance test and were further evaluated.

3.3 Program Metrics & Statistics

Table 1 gives several comparisons of the 12 accepted versions (identified by a Greek letter) with respect to some common software static metrics. The objective of software metrics is to evaluate the quality of the product in a quality assurance environment. For this project, however, we compare these program versions and observe their differences.

Notation

LINES number of lines of code, including comments and blank lines
LN-CM number of lines, excluding comments and blank lines
STMTS number of executable statements, such as assignment, control, I/O, or arithmetic statements
MODS number of programming modules (subroutines, functions, procedures, *etc*) used
STM/M mean number of statements per module
CALLS number of calls to programming modules
GBVAR number of global variables
LCVAR number of local variables.

96 Faults were found & reported during the life of the project. Table 2 classifies the faults by fault type.

Notation (Fault Classification)

Implementation Related

Typo typographical (cosmetic mistake made in typing the program)
Omiss error of omission (piece of required code was missing)
IncAlg incorrect algorithm (deficient implementation of an algorithm) — most frequent fault type, *eg*, miscomputation, logic fault, initialization fault, boundary fault

Specification Related

SpecMis specification misinterpretation
SpecAmb specification ambiguity (unclear or inadequate specification which led to a deficient implementation).

TABLE 1
Software Metrics for the 12 Accepted Programs
[versions are designated by lower-case Greek letters
range ≡ [highest value]/[lowest value] for each metric]

Metric	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	range
LINES	8769	2129	1176	1197	1777	1500	1360	5139	1778	1612	2443	1815	7.46
LN-CM	4006	1229	895	932	1477	1182	1251	2520	1168	1070	1683	1353	4.30
STMTS	2663	708	706	720	1208	753	640	1366	759	810	932	858	4.16
MODS	53	11	6	15	6	47	17	17	21	24	17	11	8.83
STM/M	179	64	101	439	201	406	38	80	36	35	67	78	12.5
CALLS	84	123	16	23	37	76	31	626	100	106	30	66	39.1
GBVAR	0	55	101	180	86	406	7	0	354	423	421	26	—
LCVAR	1326	179	86	309	553	532	376	402	294	258	328	329	15.4

TABLE 2
Fault Classification by Fault Types [19]
[see notes on table 1]

Class	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	Total
Typo	0	0	0	1	2	2	0	0	0	0	0	0	5
Omiss	0	0	4	0	1	0	3	1	0	0	1	0	10
IncAlg	7	1	3	6	2	1	3	3	4	3	6	2	41
SpecMis	2	2	0	1	1	4	3	3	4	2	2	4	28
SpecAmb	0	4	3	0	0	0	0	1	0	0	1	0	9
Other	0	0	0	1	1	0	0	0	1	0	0	0	3
Total	9	7	10	9	7	7	9	8	9	5	10	6	96

Table 3 shows the test phases during which the faults were detected, and the fault density of the original version and the accepted version.

Notation

Coding	Coding and Unit-Test
Integr	Integration
ATi	Acceptance Test i, $i = 1,2$
Operat	Operational
FD	fault density (per 1000 lines of uncommented code)
Orig FD	original FD
ATi FD	after passing ATi.

There were only two incidences of identical faults committed by two programs during the whole life cycle.

1. Committed by θ & μ versions — due to an incorrect initialization of a variable. Unit test data detected this fault very early when both programs were initially tested.

2. Committed by γ & λ versions — an incorrect condition for a switch variable (Boolean variable) for a late flight mode. This fault was not detected until in the AT1 (step 1) when a complete flight simulation was first exercised.

Later in the operational testing phase, 1k flight simulations, or over 5M program executions, were conducted. Only 1 fault (in β version) was found. This indicates that the program quality obtained from this project is very high. For the 12 accepted programs, the average FD = 0.05 faults/(1k lines of code). This number is close to the best current industrial software engineering practice. A detailed report on the U/Iowa & Rockwell Project is in [11].

4. PROGRAM ANALYSIS BY ATAC

Upon the completion of the U/Iowa & Rockwell Project, its product (12 accepted and fully operational program versions) was available for investigation. Our particular interest here is the investigation of testing coverage metrics as a quality control mechanism to evaluate & analyze these programs. The ATAC tool facilitates the generation of interesting results (summarized in tables 4 - 7).

Table 4 shows some more static program metrics of the 12 programs which were not in table 1. These new metrics, including blocks, decisions, *c*-uses, and *p*-uses, are program constructs related to the quality of testing. ATAC can automatically measure these program constructs which reveal the testing-related program complexity. All the metrics in table

TABLE 3
Fault Classification by Phases and Other Attributes
[see notes on table 1]

Test Phase	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	Total
Coding	2	2	3	1	3	3	5	3	2	1	2	2	29
Integr	4	3	4	4	1	0	3	2	2	2	3	1	29
AT1	1	2	3	4	1	2	1	2	3	2	5	3	29
AT2	1	0	0	0	2	2	0	1	2	0	0	0	8
Operat	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
Orig FD	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
AT1 FD	0.5	0	0	0	1.4	1.7	0	0.4	1.7	0	0	0	0.48
AT2 FD	0.2	0	0	0	0	0	0	0	0	0	0	0	0.05

TABLE 4
Testing-Related Program Metrics Measured By ATAC
[see notes on table 1]

Metrics	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	range
blocks	511	711	531	554	679	537	367	1132	542	473	457	483	3.08
decisions	216	250	320	297	520	284	286	357	264	237	231	262	2.41
c-uses	935	755	395	696	1027	636	710	965	727	537	803	665	2.60
p-uses	413	340	349	520	611	463	459	419	355	310	279	392	2.19

TABLE 5
Single Execution Testing Coverage Measured By ATAC
[see notes on table 1]

Metrics	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	average	range
blocks	332	417	329	389	302	341	205	675	370	321	325	277	356.9	3.29
%	65	59	62	70	44	64	56	60	68	68	71	57	62.0	1.61
decisions	77	92	119	127	138	80	95	103	110	97	97	84	101.6	1.79
%	36	37	37	43	27	28	33	29	42	41	42	32	35.6	1.59
c-uses	557	431	220	347	460	364	310	670	405	295	446	368	406.1	3.05
%	60	57	56	50	45	57	44	69	56	55	56	55	55.0	1.57
p-uses	124	117	134	168	159	101	105	153	149	111	105	114	128.3	1.66
%	30	34	38	32	26	22	23	37	42	36	38	29	32.3	1.91

TABLE 6
Integration Testing Coverage Measured By ATAC
[see notes on table 1]

Metrics	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	average	range
blocks	433	506	408	462	503	464	290	859	434	417	394	385	462.9	2.96
%	85	71	77	83	74	86	79	76	80	88	86	80	80.4	1.24
decisions	153	183	200	198	313	205	197	220	167	185	167	172	196.7	2.05
%	71	73	63	67	60	72	69	62	63	78	72	66	68.0	1.30
c-uses	778	573	315	468	716	515	508	811	538	435	625	544	568.8	2.57
%	83	76	80	67	70	81	72	84	74	81	78	82	77.3	1.25
p-uses	274	205	221	244	353	271	223	254	210	212	179	239	240.4	1.97
%	66	60	63	47	58	59	49	61	59	68	64	61	59.6	1.45

TABLE 7
Acceptance Testing Coverage Measured By ATAC
[see notes on table 1]

Metrics	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	average	range
blocks	488	553	469	529	598	524	335	1033	487	461	443	453	531.1	3.08
%	95	78	88	95	88	98	91	91	90	97	97	94	91.8	1.24
decisions	191	217	249	245	399	255	234	280	208	218	206	223	243.8	2.09
%	88	87	78	82	77	90	82	78	79	92	89	85	83.9	1.19
c-uses	893	676	378	585	898	603	618	928	624	513	744	625	673.8	2.46
%	96	90	96	84	87	95	87	96	86	96	93	94	91.7	1.14
p-uses	345	245	271	300	454	334	263	297	256	262	223	311	296.8	2.04
%	84	72	78	58	74	72	57	71	72	85	80	79	73.5	1.49

4 have a tighter range than all the metrics in table 1. There are no strong correlations among these 4 program constructs. For example, the β version has an average value of blocks & *p*-uses, the smallest number of decisions, but a very high value of *c*-uses.

Tables 5 - 7 analyze the quality of different tests conducted on the 12 program versions. Table 5 shows the testing coverage of these programs upon a simple test case which includes only one program execution.

This test case thus serves as a baseline to observe the testing quality improvement when more test cases are executed. In table 5, a simple, common test case has a variety of effects on different program constructs of different program versions. Table 5 shows a fairly large range of coverage of blocks (44% - 71%), decisions (27% - 43%), *c*-uses (44% - 69%), and *p*-uses (22% - 38%). Moreover, the coverage of blocks and *c*-uses is higher compared to that of decisions and *p*-uses.

Tables 6 & 7 give the testing coverage measures by the Integration Test data and the Acceptance Test data, respectively. The Integration Test data contains 4 test-data files for a total of 960 program executions. The Acceptance Test data, a super set of the Integration Test data, also contains 4 test-data files (each represents a complete flight simulation) for a total of about 21K program executions. Both test data sets include the test data in table 5.

Tables 6 & 7 show that the programs have been tested with fairly high quality. In particular, the Acceptance Test achieves coverages as high as 98% of blocks, 92% of decisions, 96% of *c*-uses, and 85% of *p*-uses in some programs. Even though some programs have consistent scores in these measures (*eg*,

ν version has very high values in all the measures; ζ version has both the lowest % c-uses and % p-uses), some programs do not (*eg*, θ version has the highest % blocks, very high % decisions and % c-uses, but relatively low % p-uses).

Tables 5 - 7 show that as the number of program execution increases, the quality of test increases, and the range of fraction-of-coverage decreases. Nevertheless, considering that these coverage results are obtained from the program versions of the same application tested through the same data, the differences in these measures are still important (*eg*, the θ version obtained 98% of block coverage while the γ version only obtained 78%). On the other hand, there was a diminishing return on the coverage after the acceptance test, and the operational test data (5M program executions) did not increase this coverage appreciably. This means that the 22% uncovered code in the γ version was probably not even executed during the operational phase.

There could be a correlation between the 'number of faults detected in a version' and the 'coverage of the program constructs of the version'; we hypothesize that the better a program is covered during testing, the more faults are detected. However, we did not see strong correlations between the total faults detected in the program versions (table 3) and their coverage measures during various testing conditions (tables 5 - 7). This could be due to the fact that each version has a different fault distribution to begin with, and therefore, the coverage measures would not be a good predictor for the absolute number of faults in the program. Besides, the number of faults detected in each version is not very large, which can reduce the statistical precision in the analysis.

Finally, in using ATAC capability in highlighting non-covered code in the program, we can reveal the programming style and the testability of a program easily by examining the coverage of program constructs in detail. In the γ version, for example, we noticed that an untested error-handling function accounts for 10% of the total blocks while the same function accounts for only 1-2% of block coverage in most other versions. The γ version used many function calls to pass each parameter in the calling routine of the error-handling function, and each function call was counted as an uncovered block. This clearly indicates the need for an extra test case to test this function, which can increase the block coverage of the γ version appreciably.

REFERENCES

[1] J.D. Musa, A. Iannino, K. Okumoto, *Software Reliability - Measurement, Prediction, Application*, 1987; McGraw-Hill.

[2] S.R. Dalal, C.L. Mallows, "When should one stop testing software?" *J. Amer. Statistical Assoc*, vol 83, 1988 Sept, pp 872-879.

[3] M.R. Lyu, A. Nikora, "Using software reliability models more effectively", IEEE Software, vol 9, 1992 Jul, pp 43-52.

[4] J.R. Horgan, S.A. London, "A data flow coverage testing tool for C", *Proc. Symp. Assessment of Quality Software Development Tools*, 1992 May, pp 2-10.

[5] M.R. Lyu, J.R. Horgan, S. London, "A coverage analysis tool for the effectiveness of software testing", *Proc. ISSRE'93*, 1993 Nov, pp 25-34.

[6] W.E. Howden, *Functional Program Testing and Analysis*, 1987; McGraw-Hill.

[7] R.A. DeMillo, W.M. McCracken, R.J. Martin, J.F. Passafiume, *Software Testing and Evaluation*, 1987; Benjamin/Cummings Publishing.

[8] S. Rapps, E.J. Weyuker, "Selecting software test data using data flow information", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Apr, pp 367-375.

[9] P.G. Frankl, E.J. Weyuker, "An applicable family of data flow testing criteria", *IEEE Trans. Software Engineering*, vol 14, 1988 Oct, pp 1483-1498.

[10] DeMillo, Gundi, King, McCracken, Offutt, "An extended overview of the Mothra software testing environment", *Proc. Second Workshop on Software Testing, Verification, and Analysis*, 1988; IEEE Computer Society.

[11] M.R. Lyu, Y. He, "Improving the N-version programming process through the evolution of a design paradigm", *IEEE Trans. Reliability*, vol 42, 1993 Jun, pp 179-189.

[12] P.M. Maurer, "Generating Test Data with Enhanced Context-Free Grammars", IEEE Software, vol 7, 1990 Jul, pp 50-55.

[13] D.C. Ince, "The automatic generation of test data", *The Computer J.*, vol 30, num 1, 1987, pp 63-69.

[14] A.J. Offutt, "Automatic test data generation", *PhD Thesis*, 1988; Georgia Institute of Technology.

[15] J.R. Horgan, D.J. Moore, "Methods for improving language-based editors", *Proc. Sigplan/Sigsoft Conf. Programming Environments*, 1984.

[16] S.C. Johnson, "YACC: Yet Another Compiler-Compiler", (internal memorandum), 1975; AT&T Bell Labs.

[17] D.W. Nachbar, "SPIFF % A program for making controlled approximate comparison of files", (internal memorandum), 1988; Bellcore.

[18] A. Avižienis, "The N-version approach to fault-tolerant software", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1491-1501.

[19] M.R. Lyu, A. Avižienis, "Assuring design diversity in N-version software: A design paradigm for N-version programming", *Dependable Computing and Fault-Tolerant Systems* (J.F. Meyer, R.D. Schlichting, Eds), 1992, pp 197-218; Springer-Verlag.

[20] J.R. Horgan, S. London, M.R. Lyu, "Achieving software quality with testing coverage measures", *IEEE Computer*, vol 27, 1994 Sep, pp 60-69.

AUTHORS

Dr. Michael R. Lyu; Bellcore; 445 South St; Morristown, New Jersey 07960 USA.

Internet (e-mail): lyu@bellcore.com

Michael R. Lyu received his BS (1981) in Electrical Engineering from National Taiwan University, MS (1984) in Electrical & Computer Engineering from the University of California, Santa Barbara, and PhD (1988) in Computer Science from the University of California, Los Angeles. He was with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, as a Member of the Technical Staff from 1988 to 1990. In 1990 he joined the Electrical & Computer Engineering Department at the University of Iowa as an Assistant Professor. Since 1992 June, he has been a Member of the Technical Staff in the Applied Research Area of Bellcore. His research interests include software engineering, software reliability, and fault-tolerant computing; he has published over 40 papers in these areas. He is the editor for two book volumes: *Software Fault Tolerance*, 1995 and *McGraw-Hill Software Reliability Engineering Handbook*, 1995.

Dr. J. R. Horgan; Bellcore; 445 South St; Morristown, New Jersey 07960 USA.

jrh@bellcore.com

Joseph R. Horgan received a BA & MA from the University of Delaware in Philosophy and a PhD from the Georgia Institute of Technology in Computer Science. Since 1983 he has been a member of the technical staff of the Bellcore Information Sciences & Technologies Research Laboratory. His research is in software analysis, testing, and reliability. Prior to his employment with Bellcore he was with Bell Labs and on the faculty of Computer Science at the University of Kansas. He has also worked at the University of Delaware and IBM.

Saul London; Bellcore; 445 South St; Morristown, New Jersey 07960 USA.
saul@bellcore.com

Saul London received his BA (1980) in Mathematics from Columbia University and MS (1982) in Computer Science from New York University. He is a member of technical staff in the Information Sciences & Technologies Research Laboratory at Bellcore. His research interests include software testing, programming languages, and software reuse, and telecommunication software.

Manuscript received 1994 May 15.

IEEE Log Number 94-06876 ◄TR►