A Coverage Analysis Tool for the Effectiveness of Software Testing

Michael R. Lyu
J. R. Horgan
Saul London
Bell Communications Research
445 South Street
Morristown, NJ 07960
<lyu@bellcore.com>
(201)829-3999

Keywords – N-Version Programming, Data Flow Testing, Test Effectiveness, Testing Coverage Tool, Software Metrics.

Reader Aids –

General purpose: Description of a testing coverage analysis tool and its experimental results.

Special math needed for explanations: None

Special math needed to use results: None

Results useful to: Software testers and software reliability engineers

Abstract & Conclusions – This paper describes a software testing and analysis tool, called **ATAC** *(***A***utomatic* **T***est* **A***nalysis for* **C**), developed as a research instrument at Bellcore to measure the effectiveness of testing data. It is also recommended as a tool to facilitate the design and evaluation of test cases during software development. To demonstrate the capability and 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 the ATAC tool to analyze and compare coverage of the testing conducted in 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 implicitly 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, namely, software with higher reliability. One would hope, for example, that code tested to 85% c-uses coverage would exhibit 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 rate of field faults (or that there is none) is the ultimate and critical test of the usefulness of data flow coverage testing. In fact, we demonstrated with 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, achieved 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 confidence.

The ultimate question that we hope ATAC can help us to answer is a typical question to all software reliability engineers: "When is a program considered acceptable?" This question presents a tremendous challenge to every researcher in the software engineering field. Software reliability modeling people have proposed several models to answer this question [1], [2], [3]. However, none of these models address the issues of program constructs, data flow testing, and testing coverages, which are deemed important to testing people. We intend to investigate the relationship between the quality of data flow testing and the subsequent detection of field 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 and practitioners to establish the relationship in between structure-based testing schemes and software reliability measurement techniques.

1. Introduction

ATAC (**A**utomatic **T**est **A**nalysis for **C**) is a software tool which supports data flow coverage testing for C programs [4],[5]. Coverage testing helps the tester create a thorough set of tests and gives a measure of test completeness. Each of the different structural coverage criteria proposed in the literature [6],[7],[8] attempts to capture some important aspect of a program's behavior. Rapps and Weyuker [8] define a family of data flow coverage criteria for an idealized programming language. Frankl and 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 standard 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 standard 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, however, the multiple program versions obtained from the project as an abundant resource for the study of testing coverage and quality metrics. ATAC facilitates this study. Preliminary results have shown 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* (i.e., whether a program is easy to test or not), becomes visible through the resulting measures. As will be seen 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 to the confidence of testing upon these programs.

Section 2 presents ATAC in terms of its purpose, its implementation, and its uses. Section 3

describes an industrial project conducted recently to obtain 12 program versions for a critical flight software system. Section 4 discusses some experience and results obtained in applying ATAC to the final program versions obtained in the project. Some conclusions and future study plans are given in Section 5.

2. The 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.

To use ATAC one prepares a program for testing with a preprocess-compile-link phase. This creates an instrumented object module and data flow tables used during run-time. Next, tests are run and trace and coverage data are collected by ATAC's run-time routine. Then the programmer executes an analysis phase which provides feedback on the tests that have been run. Finally, the programmer uses 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 may request coverage values on the preceding test for any of the data flow coverage measures, and may 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 may be performed for each C function, for each test, or some combination of tests and C functions. Multiple source files may 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 may be constrained by available memory and disk space, and 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 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.

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, counts 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, 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 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');"`).

2.1 The Purposes of ATAC

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

(1) measuring test set or test session completeness,

(2) displaying non-covered code to aid in test creation,

(3) selecting effective randomly generated tests, and

(4) reducing regression test set size by eliminating redundant tests.

The first purpose, measuring test completeness, gives an objective measure of how completely a program or routine has been tested. This measure is useful in evaluating the quality of the testing procedure being used, and in establishing a level of confidence 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 confidence that the program, in passing the tests, works correctly.

The second purpose, displaying code not covered, is a programmer's 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 particular 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.

The third purpose of ATAC is to select effective, randomly generated tests. For many applications it is possible to automatically generate tests (e.g. [12],[13],[14]). However, there must be a mechanism for determining an effective, small subset of the large number of automatically generated tests, if the generation process is to be most fully useful. ATAC coverage measures, or any coverage measures, provide a basis for such a mechanism. While the number and complexity of data flow objects associated with a program may 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.

The tests run over the life of a program are often collected together to form 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 behavior of the program. At some point a regression test set may grow large enough that it is not practical to run the whole set of tests after small program modifications. Hence, the fourth purpose of coverage testing uses the coverage measure to select a subset of the regression tests which together achieve a high level of coverage. This technique may identify tests that add no coverage at all to the regression tests, and are therefore candidates for deletion.

2.2 The Design and Implementation of ATAC

ATAC is currently 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. The first prototype analyzed C properly but only did block coverage. The second version was rather complete but consisted of over 50,000 lines of poorly engineered code. The present version is well-engineered and designed to accommodate changes and extensions. ATAC is running in a variety of UNIX environments (Sun 3, Sun SPARC, Dec 3100, Vax 8650, Pyramid, and others) in several Bellcore divisions and at Purdue University. It is reasonably easy to port and install. ATAC has been run successfully on programs up to 100,000 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, where *n* is the number of test runs. Execution time may increase significantly, 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 megabyte system.)

- *ATAC Preprocessor*

The ATAC preprocessor is the heart of ATAC. The preprocessor parses and analyzes C source code and outputs an instrumented version of the source code and a file containing static data flow information. The C parser used in ATAC was originally part of a language-based editor for C [15].

A parser generated by the YACC parser generator tool [16] 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 de-parsed 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 pointers to the DEF/USE tables, information about constructs already covered and 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.

- *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.

- *ATAC 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 and 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 and use are recorded in the trace file. Multiple occurrences of the same definition/use pair are not recorded. Because a single procedure may 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.

- *ATAC Analysis*

The ATAC analysis program reads the static data flow information for each source file being tested and reads 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, the total number of constructs in the function and the number of constructs executed by the tests. The analysis results could be either broken down by test files, or broken down by program modules.

2.3 The 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 5,000 line Spiff program [17], the functional tests presented the following coverage profile:

% blocks	% decisions	% c-uses	% p-uses
62(1009/1622)	54(471/869)	46(1023/2242)	42(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 ATAC is of no further use. If the tester chooses to improve the coverage ATAC can aid in the selection of new tests.

The tester who wishes to improve coverage by hand-crafting tests may 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 || muflg || i == 1 ||
         (*compare)(ibuf[i-1] -> 1, ibuf[i-2] -> 1 ))) {
            do
                putc(*cp, os);
            while (*cp++ != '\n');
            if (ferror(os)) {
                error = 1;
                term();
            }
      }
```

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 and run, the coverage is checked, and the blocks not covered are re-displayed until one is 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. The U. of Iowa / Rockwell Joint Project

The N-Version Programming (NVP) approach achieves fault-tolerant software systems, called *N-Version Software* (NVS) systems, through the development and use of design diversity [18]. This approach involves the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. The NVP approach 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."

In Fall 1991, a real-world automatic (i.e., computerized) airplane landing system, or so-called *autopilot*, was developed and programmed by 15 programming teams at the University of Iowa and the Rockwell/Collins Avionics Division [11]. A total of 40 students (33 from ECE and CS departments at the University of Iowa, 7 from the Rockwell International) participated in this project to independently design, code, and test the computerized airplane landing system for the major requirement of a graduate-level software engineering course.

3.1 The Application Problem

The application used 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 (FCC) 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 Federal Aviation Administration (FAA). The *pitch control* part of the auto-landing problem, i.e., the 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 ten miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet per second). The Complementary Filters preprocess the raw data from the aircraft's 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 parts, the Outer Loop and the Inner Loop. 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 and Track mode and begins a pitching motion to acquire and 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 two feet per 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.

3.2 The Software Development Process

The development of this software project was scheduled and conducted in six phases:

(1) Initial design phase (4 weeks):

The purpose of this phase 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 four-week phase, each team delivered a preliminary design document, which followed specific guidelines and formats for documentation.

(2) Detailed design phase (2 weeks):

The purpose of this phase was to let each team obtain some feedbacks from the coordinator to adjust, consolidate, and complete their final design. Each team was also requested to conduct one or several design walkthroughs. At the end of this two-week phase, each team delivered a detailed design

document and a design walkthrough report.

(3) Coding phase (3 weeks):

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

(4) Unit testing phase (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. A total of 133 data files was provided to the programmers.

(5) Integration testing phase (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 intended to guarantee that the software was suitable for a flight simulation environment in an integrated system.

(6) Acceptance testing phase (2 weeks):

Programmers formally submitted their programs for a two-step acceptance test. In the first step (AT1), each program was run in a test harness of four nominal flight simulation profiles. For the second step (AT2), one extra simulation profile, representing an extremely difficult flight situation, was imposed. In total there were *23930* executions imposed on these programs before they were accepted and subjected to the final evaluation in the following stage. By the end of this two week phase, 12 of the 15 programs passed the acceptance test and were subject to further evaluations.

3.3 Program Metrics and Statistics

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

The following (static) metrics are considered in Table 1: (1) the number of lines of code, including comments and blank lines (LINES); (2) the number of lines excluding comments and blank lines (LN-CM); (3) the number of executable statements, such as assignment, control, I/O, or arithmetic statements (STMTS); (4) the number of programming modules (subroutines, functions, procedures, etc.) used (MODS); (5) the mean number of statements per module (STM/M); (6) the number of calls to programming modules (CALLS); (7) the number of global variables (GBVAR); and (8) the number of local variables (LCVAR). The last column, max-min, is the ratio of the highest value to the lowest value for each metric.

A total of 96 faults was found and reported during the whole life cycle of the project. Classification of faults according to fault types is shown in Table 2. This category considers the following type of faults [19]: (1) typographical (a cosmetic mistake made in typing the program); (2) error of omission (a piece of required code was missing); (3) incorrect algorithm (a deficient implementation of an algorithm); (4) specification misinterpretation (a misinterpretation of the specification); and (5) specification ambiguity (an unclear or inadequate specification which led to a deficient implementation). In this category, items (1) through (3) are implementation-related faults, while items (4) and (5) are specification-related faults. It is also noted that "incorrect algorithm" of item (3) is the most frequent fault type, which includes miscomputation, logic fault, initialization fault, and boundary fault.

Table 3 shows the test phases during which the faults were detected, and the fault density (as per thousand lines of uncommented code, abbreviated as F.D.) of the original version and the accepted version. "F.D. after AT1" represents the fault density of the program versions after passing the Acceptance Test Step 1.

It is interesting to note that there were *only two* incidences of identical faults committed by two programs during the whole life cycle. The first fault, committed by θ version and μ version, was due to an incorrect initialization of a variable. Unit test data detected this fault very early when both programs were initially tested. The second fault, committed by γ and λ version, was an incorrect condi-

tion for a switch variable (a Boolean variable) for a late flight mode. This fault did not manifested itself until the Acceptance Test Step 1 where a complete flight simulation was first exercised.

Later in the operational testing phase, 1000 flight simulations, or over five million program executions, were conducted. Only one fault (in the β version) was found. This indicates that the program quality obtained from this project is very high. For the 12 accepted programs, the average F.D. was 0.05 faults per KLOC (thousand lines of code). This number is close to the best current industrial software engineering practice. A detailed report on the U. of Iowa / Rockwell Project could be found in [11].

4. Program Analysis by ATAC

Upon the completion of the U. of Iowa / Rockwell Project, its product, namely, the 12 accepted and fully operated program versions, is available for various investigations. Our particular interest here is the investigation of testing coverage metrics as a quality control mechanism to evaluate and analyze these programs. The ATAC tool facilitates the generation of some interesting results, which are summarized in the following tables.

Table 4 shows some more static program metrics of the 12 programs which were missing in Table 1. These new metrics, including blocks, decisions, c-uses, 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. The highest value to the lowest value for each metric is given in the last column (max-min). It is interesting to note that all the metrics in Table 4 have a tighter max-min range than all those metrics in Table 1. We also note that there are no strong correlations among these four program constructs. For example, the β version has an average value of blocks and 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.

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	max-min
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 1: Software Metrics for the 12 Accepted Programs

Fault Class	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Total
(1) Typo	0	0	0	1	2	2	0	0	0	0	0	0	5
(2) Omission	0	0	4	0	1	0	3	1	0	0	1	0	10
(3) Incorrect Algorithm	7	1	3	6	2	1	3	3	4	3	6	2	41
(4) Spec. Misinterpretation	2	2	0	1	1	4	3	3	4	2	2	4	28
(5) Spec. Ambiguity	0	4	3	0	0	0	0	1	0	0	1	0	9
(6) 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 2: Fault Classification by Fault Types

Test Phase	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Total
Coding/Unit Test	2	2	3	1	3	3	5	3	2	1	2	2	29
Integration Test	4	3	4	4	1	0	3	2	2	2	3	1	29
Acceptance Test 1	1	2	3	4	1	2	1	2	3	2	5	3	29
Acceptance Test 2	1	0	0	0	2	2	0	1	2	0	0	0	8
Operational Test	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 F.D.	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
F.D. after AT1	0.5	0	0	0	1.4	1.7	0	0.4	1.7	0	0	0	0.48
F.D. after AT2	0.2	0	0	0	0	0	0	0	0	0	0	0	0.05

Table 3: Fault Classification by Phases and Other Attributes

This test case thus serves as a baseline to observe the testing quality improvement when more test cases are executed. For Table 5 we notice that a simple, common test case has a variety of effects on different program constructs of different program versions. It shows a fairly large range of coverage

percentages of blocks (44% - 71%), decisions (27% - 43%), c-uses (44% - 69%), and p-uses (22% - 38%). Moreover, the coverage of blocks and c-uses could be better achieved comparing with that of decisions and p-uses.

Tables 6 and 7 give the testing coverage measures for the Integration Test data and the Acceptance Test data, respectively. The Integration Test data contains four test data files for a total of 960 program executions. The Acceptance Test data, a super set of the Integration Test data, also contains four test data files (each represent a complete flight simulation) for a total of about 21000 program executions. Both test data include the test data used in Table 5.

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	max-min
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 4: Testing-related Program Metrics Measured By ATAC

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	average	max-min
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 5: Single Execution Testing Coverage Measured By ATAC

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	average	max-min
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	1.3
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 6: Integration Testing Coverage Measured By ATAC

Metrics	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	average	max-min
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

Table 7: Acceptance Testing Coverage Measured By ATAC

From Tables 6 and 7 we clearly see 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. Notice that even though some programs have consistent scores in these measures (e.g., 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 (e.g., θ version has the highest % blocks, very high % decisions and % c-uses, but relatively low % p-uses).

It is also noted from Tables 5 through 7 that as the number of program execution increases, the quality of test increases, and the max-min range of coverage percentages reduces. 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 considered significant (e.g., the θ version obtained 98% block coverage while the γ version only obtained 78%). On the other

hand, we also noticed that there was a diminishing return on the coverage after the acceptance test, and the operational test data (five million program executions) did not increase this coverage significantly. This means that the 22% uncovered code in the γ version was probably not even executed during the operational phase.

One may suspect that there could be a correlation between the number of faults detected in a version and the coverage of the program constructs of the version, since it is hypothesized that 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 3) and their coverage measures during various testing conditions (Tables 5, 6 and 7). This may 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 may reduce the statistical significance in the analysis.

Finally, in using ATAC's capability for 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 the 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. It was observed that the γ version used a large amount of function calls to pass each parameter in the calling routine of the error handling function, and each of the 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 significantly.

References

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

[2] Dalal, S.R., Mallows, C.L., "When Should One Stop Testing Software?" *Journal of American Statistics Association*, pp. 872-879, 1983.

[3] Lyu, M.R., A. Nikora, "Using Software Reliability Models More Effectively," *IEEE Software*, pp. 43-52, July 1992.

[4] Horgan, J.R., S.A. London, "A Data Flow Coverage Testing Tool for C," *Proceedings of Symposium on Assessment of Quality Software Development Tools*, New Orleans, LA., pp. 2-10, 1992.

[5] Lyu, M.R., J.R. Horgan, S. London, "A Coverage Analysis Tool for the Effectiveness of Software Testing," *Proceedings of ISSRE'93*, Denver, CO, pp. 25-34, November 1993.

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

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

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

[9] Frankl, P.G., E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, Vol. SE-14, No.10, October 1988.

[10] DeMillo, R.A., D.S. Gundi, K.N. King, W.M. McCracken, and A.J. Offutt, "An extended overview of the Mothra software testing environment," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, IEEE Computer Society, 1988.

[11] Lyu, M.R., "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm," *IEEE Transactions on Reliability*, vol. 42, No. 2, pp. 179-189, June 1993.

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

[13] Ince, D.C., "The Automatic Generation of Test Data," *The Computer Journal*, vol. 30, no. 1, 1987.

[14] Offutt, A.J., "Automatic Test Data Generation," Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Ga., 1988.

[15] Horgan, J.R., D.J. Moore, "Methods for Improving Language-Based Editors," *Proceedings of the Sigplan/Sigsoft Conference on Programming Environments*, Pittsburgh, PA. 1984.

[16] Johnson, S.C., "Yacc: Yet Another Compiler-Compiler," AT&T Bell Laboratories internal memorandum, 1975.

[17] Nachbar, D.W., "SPIFF -- A Program for Making Controlled Approximate Comparison of Files," Bellcore internal memorandum, 1988.

[18] Avizienis, A., "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, Vol. SE-11, no. 12, pp. 1491-1501, December 1985.

[19] Lyu, M.R., 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, R.D. Schlichting eds., Springer-Verlag, Wien, New York, pp. 197-218, 1992.