D AN / AMES

Center for Reliable Computing

A METHODOLOGY FOR TESTING FAULT-TOLERANT SOFTWARE

Dorothy M. Andrews, Aamer Mahmood, and Edward J. McCluskey

CRC Technical Report No. 85-22
(CSL TN No. 85-282)

November 1985

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Depts. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305 USA

This work was supported in part by NASA-AMES Research Center under Grant No. NAG 2-246.

(NASA-CR-176713) A METHODOLOGY FOR TESTING FAULT-TOLERANT SOFTWARE (Stanford Univ.) 26 p HC A03/MF A01 CSCL 09B

N86-23323

Unclas
G3/61 16615

A METHODOLOGY FOR TESTING FAULT-TOLERANT SOFTWARE

Dorothy M. Andrews, Aamer Mahmood, and Edward J. McCluskey

CRC Technical Report No. 85-22
(CSL TN No. 85-282)

November 1985

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Depts. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305 USA

ABSTRACT

A methodology for testing fault-tolerant software is presented in this paper. There are problems associated with testing fault-tolerant software because many errors are masked or corrected by voters, limiters, automatic channel synchronization, etc. This methodology illustrates how the same strategies used for testing fault-tolerant hardware can be applied to testing fault-tolerant software. For example, one strategy used in testing fault-tolerant hardware is to disable the redundancy during testing. A similar testing strategy is proposed for software, namely, to move the major emphasis on testing earlier in the development cycle (before the redundancy is in place) thus reducing the possibility that undetected errors will be masked when limiters and voters are added.

TABLE OF CONTENTS

1. INTRODUCTION

This paper presents a methodology for testing fault-tolerant software. The reason for such a specific methodology is that testing fault-tolerant software presents problems not encountered in testing other types of software. These problems were discovered during an experiment to assess the use of assertions in dynamic testing of digital flight control system software. The experiment demonstrated that assertion testing can be very effective at detecting errors in flight software, however, it also uncovered a unique set of problems in testing fault-tolerant software. Unfortunately, these problems are sometimes caused by the very method used for implementation of fault tolerance, that is, the fault masking and fault secure techniques of duplication, voters, limiters, etc. Solutions for the problems have been incorporated into this methodology for testing fault-tolerant software.

Factors taken into account during development of this testing methodology are discussed in Sec. 2. The methodology for testing fault- tolerant software is presented in Sec. 3. Section 4 contains various scenarios or levels of the use of assertion testing of fault-tolerant systems. A summary is in Sec. 5.

2. DEVELOPMENT OF THE METHODOLOGY

There were many factors contributing to the development of this testing methodology, including consideration of the desirable attributes of a testing methodology, a thorough study of various software testing methods, and an analysis of the problems encountered in testing fault-tolerant software. These factors are discussed in this section.

2.1 IMPORTANT ATTRIBUTES OF A TESTING METHODOLOGY

Before a testing methodology can be formulated, it is important to determine the attributes or characteristics that a testing methodology must have. Some of these attributes are general and apply to any software testing methodology, while others are specific to fault-tolerant software testing. They are as follows:

* **Good error detection capability** - the basic testing strategy should have a high probability of finding errors.

* **Adaptable to automation** - including automation of test evaluation as well as test case generation.

* **Cost effective** - should help in reduction of man power and time for testing.

* **Appropriate to the software** - some types of software, such as fault-tolerant software, have special problems that need to be considered when a test plan is developed.

* **On-line error detection** - in order to provide recovery, fault-tolerant software requires error detection during system operation (deployment).

* **Not affect performance of software** - on-line testing must not have high overhead requirements.

2.2 ASPECTS OF SOFTWARE TESTING

There are many aspects to testing software, such as: when the software is tested, where the testing takes place, and how the testing is done. Software is usually subjected to some form of "testing" throughout the entire software development cycle - from the debugging efforts of the programmer to the acceptance testing conducted when the software is delivered and installed on site. In addition to this, some sort of testing or error detection scheme is incorporated in fault-tolerant software to provide masking of errors or recovery from errors when it is in operation. Testing of the software usually takes place on the computer where it is developed. Fault-tolerant software most often operates in a real-time environment in which the outputs cause direct action. Such software is usually also tested in a simulated environment and is then embedded for testing in the system for which it is ultimately intended. The methodology presented in this paper is not just testing for one phase but is broad in scope because it encompasses testing throughout the entire software development cycle and testing in all of the environments - including the on-line error detection required for fault-tolerant applications.

The most important aspect of testing is how the testing will be done, in other words, the testing approach or method. A program can be tested by static analysis or dynamically by executing the software. Static analysis is usually done by a "software tool" that examines the data flow within the program and looks for inconsistencies in the program structure or variables [Adrion 82], [Andrews 83]. Static analysis is useful in the early phases of software development but is not a substitute for dynamic program testing. This methodology is for dynamic testing of software.

There are two classifications of dynamic testing of software. One can be termed external because it looks primarily at the external results rather than at the program itself. The other can be thought of as internal testing, since it is most concerned with what is happening within the program and there is less emphasis on the actual values of the output variables. It is important to note that these types are not mutually exclusive. Figure 1 shows the basic classification of testing strategies.

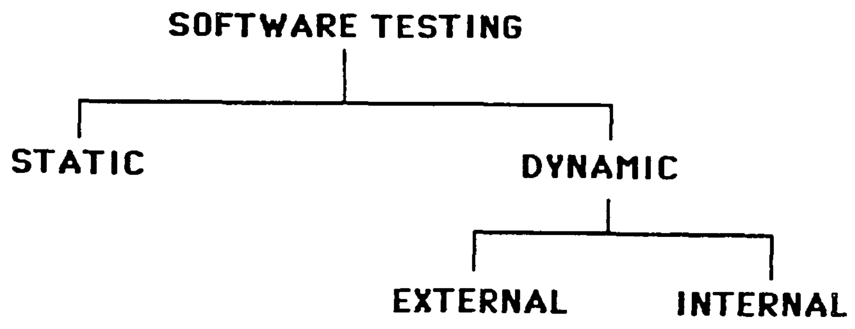

Figure 1. Basic Testing Strategies

External testing is sometimes referred to as "black box" testing because of the lack of emphasis on the program structure, etc. The program is tested by perturbation (changing) of the values of the input variables or by perturbation of the program itself. In the first type of testing, all possible values of the input variables may be tested (called exhaustive testing) or the values of the input variables may be changed according to some algorithm, as in random testing [Duran 84], [Ntafos 85], in grid testing [Andrews 85], in functional testing [Howden 80], or in adaptive testing [Andrews 81,85]. Perturbation of the program is done by mutation analysis [De Millo 78], [Budd 80] or by error seeding. Figure 2 illustrates the types of external dynamic testing.

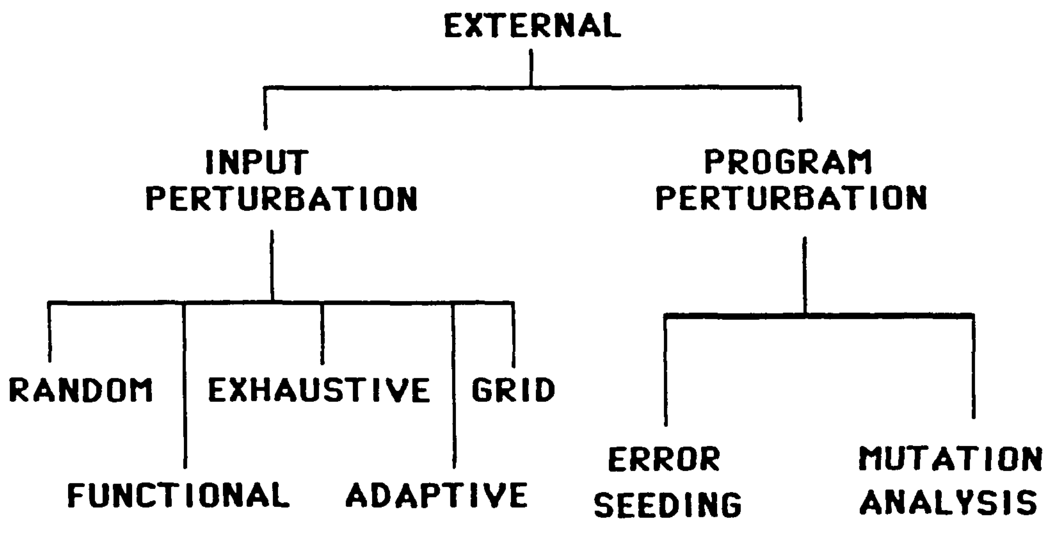

Figure 2. External Testing Strategies

Internal testing also has two general types within its classification. One type is interested mainly in the structure of the program, that is, determining what statements have been executed or what paths have been traversed [Gannon 79], [Miller 75]. For this reason, it is usually referred to as "structural" testing. The other is interested in whether or not certain conditions or specifications are valid at given points in the program. These conditions are stated in the form of logical predicates. This type of testing is called assertion testing. Assertions that are made "executable" have been used in for testing in software development environments and have been the subject of several research studies [Andrews 79-85], [Mahmood 84a,b,c]. Figure 3 is a diagram of types of internal testing.

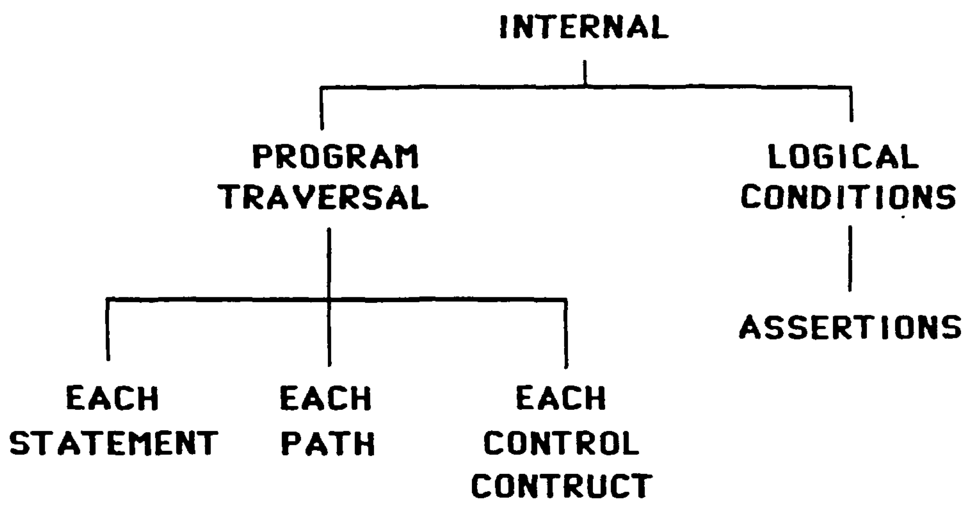

Figure 3. Internal Testing Strategies

Assertion testing is a technique for dynamically testing software by embedding additional statements, called assertions, in the software. An assertion states a condition or specification in the form of a logical expression. During execution of the program, the assertion is evaluated as true or false. If it is false, the presence of an error is indicated. Notification of the error is most often made in an output message, such as, "Assertion in procedure <xxxx> at statement # <nn> is false." However, when the testing process is fully automated, information on the assertion violations is used as input by the software program that generates test cases using adaptive or artificial intelligence techniques.

Assertion testing has distinct advantages over other testing methods and therefore was chosen as the basic strategy for a testing methodology for fault-tolerant software. Among the most important are the following:

* It has proved effective in detecting errors in typical real-time, fault-tolerant software.

* Determining correctness of the output is remarkably simplified because of the automatic notification of an error when an assertion is violated.

* Because of this simplification and cohsequent reduction of time required for assessment of test results, the generation of a larger set of input data becomes possible.

* Automation of the process of adaptively generating test data becomes easier to implement using optimization and artificial intelligence techniques [Andrews 81,85], [Cooper 76].

* Assertions left in the code during operation can provide on-line testing that is comparable to the "built-in self test" used in hardware testing.

* When combined with recovery blocks, assertions embedded in the software provide a convenient and effective way to implement on-line fault tolerance for hardware faults, as well as for software errors.

* Assertions can be made conditionally compilable, so they can be turned into comment statements and easily stripped out of the code after testing is finished.

2.3 PROBLEMS IN TESTING FAULT-TOLERANT SOFTWARE

During the development of this methodology, an experiment was conducted to assess the use of assertion testing for digital flight control system software. The software used in the experiment was the autopilot code for a wide-bodied, commercial airplane [DFCR-96 80]. The code is representative of real-time, fault-tolerant software and uses typical techniques for provision of fault tolerance.

The experiment uncovered a unique set of problems in testing fault-tolerant software [Andrews 85b], [Mahmood 84a,b]. These problems were due to the testing environment itself, as well as the basic characteristic of fault-tolerant software, that is, the redundancy used to mask or correct errors. When the software was tested in a real-time environment, which simulated actual flight conditions, the following problems were noted:

* There were many crashes of the flight computer due to the sensitivity of the simulated environment.

* "Bugs" were hard to detect, because little indication was given as to where the program had failed.

* It was difficult to determine whether failures were due to a software error or the presence of a hardware fault.

* Executing software on a simulated real-time, fault-tolerant environment is very expensive because it is time comsuming and often requires extra personnel to keep the simulator running.

In addition to the problems due to the simulated real-time environment, the following characteristics of fault-tolerant flight software that contribute to the problems in testing fault-tolerant software were identified:

* **USE OF LIMITERS** - In the autopilot code, there is frequent use of limiters which reset certain variables whose values are not within certain limits. This is done, not only to control possible errors, but also to keep the values of those variables within the limits of passenger comfort and within the stress limits of the airplane structure, etc. However, this use of limiters throughout the program interferes with detection of errors during testing

because errors can be corrected by a limiter and therefore masked.

* **USE OF VOTERS** - The values of input data, as well as the values of variables from computations, are continually voted upon. If one of the values does not agree with the others, the majority vote prevails. Therefore, errors can be masked and difficult to detect, since propagation of errors is halted.

* **AUTOMATIC CHANNEL SYNCHRONIZATION** - The autopilot flight computers have a dual-dual redundancy architecture with automatic synchronization of the channels provided by the software. Under these conditions, assertions which monitor timing do not catch errors because timing problems are immediately corrected.

These observations clearly showed that testing a software system with built-in redundancy, that is, a fault-tolerant system, is not possible using normal testing techniques. The same problems encountered in testing fault-tolerant hardware systems (fault masking, etc) exist for testing fault-tolerant software systems. The following three changes to the development and testing cycle for fault-tolerant software need to be made:

1) "Design for testability" features (such as the use of assertions for testing) should be incorporated into fault-tolerant software design specifications.

2) Because of the problems and cost of testing in a real-time environment (such as on a simulator), testing in the earlier phases should be expanded so most errors are found before real-time tests. In this way, significant reductions in time and cost could be effected.

3) The redundancy and automatic channel synchronization have to be removed to be able to test fault-tolerant software effectively without error masking. This same method, the disabling of redundancy, has been proposed for testing fault-tolerant hardware. This is another reason for emphasizing testing before the redundancy is in place. It is also a consideration during the maintenance period. When changes are made to the code, testing must done without the redundancy in order to be most effective.

3 A TESTING METHODOLOGY FOR FAULT-TOLERANT SOFTWARE

The testing methodology presented here differs from traditional testing strategies in three ways: First, it is based on the use of assertions throughout the entire software cycle from the design of the system, through all phases of testing, maintenance, and for on-line" checking during deployment. Second, the emphasis on testing is moved forward, so that more thorough testing is performed before the system is tested by real-time simulation or as an embedded system. Third, the methodology has been adapted to take into account the particular problems encountered in testing fault-tolerant software.

The typical procedure for testing flight control software (and which is probably similar for most other real-time, fault-tolerant software) is as follows: Get the program to compile, test it on a flight simulator, and then test it in the airplane.

Since the first objective is to get the program up and running, "bugs" found during compilation are removed until the program will compile without errors. Then the program is downloaded into the flight computers installed on a pallet. There is a direct connection from the pallet to another computer that provides real-time simulation of the movement of the airplane on which the flight computers will eventually be installed. Data, generated by simulation on this computer is passed to the flight computers as input. (Normally, this input data would be read from sensors.) Once the program executes without failures on the flight simulator, it is downloaded into flight computers that are installed in airplanes for actual flight tests. At this time, a flight engineer goes along and makes recommendations for changes to the code that will make the flight more smooth. These changes are usually in the form of "limiters" such as those that prevent the plane from banking too quickly and causing stress to the structure of the plane or to the passengers.

The procedure to be followed in the testing methodology for fault-tolerant software that is proposed in this paper is as follows: Write assertions as soon as possible (some can be written before implementation of the code); use assertions during debugging; test thoroughly using automation of the testing process during module and system integration tests; then remove the least important assertions for testing by real-time simulation; and follow with testing as an embedded system. The testing methodology, adapted to the various phases for fault-tolerant software, is outlined next.

3.1 SPECIFICATION/REQUIREMENTS/DESIGN

During this early stage, assertions can be written for certain conditions, generally global conditions, that are true at various points during execution of the software. These assertions are best written by system designers. For flight control software, the designers are those people who understand flight control laws and operation of airplanes. Some of these laws can be stated in the form of the logical condition of assertions. The type of assertion that is written at this stage is likely to be a comparison between different variables, as well as assertions that set the tolerances for the values of a variable.

3.2 SOFTWARE IMPLEMENTATION

Prior to this stage, assertions can be written by the designer, but once implementation of the code has begun the nature of assertions changes somewhat. The type of assertion becomes language dependent and more specific to the code as well as the machine on which it will run. This is the stage where the programmer will be the one best able to write assertions that are local in nature, that is, specific at a particular point in the program. These assertions will most likely be those that test for maximal and minimal values of variables, conditions relating to computer calculations (overflow and underflow), etc. Often these assertions have little to do with the application, i.e. flight control laws, etc., but are checks on the operation of the program in the computer.

Some of the appropriate places in a program where it would be desirable to add assertions are:

* At each invocation of a module to check values of incoming parameters.
* At each control construct (IF, WHILE, DO, etc.), because there is a greater chance of error where paths divurge.
* After data has been read to ensure meaningful values were accessed.
* Following a call to a procedure or function to be sure that values being returned are within acceptable boundaries.
* Following complex calculations to prevent propagation of an error resulting from an incorrect formula.

Testing during implementation is usually referred to as debugging, because it is not a formalized testing procedure and often depends solely on the inclination and preference of the programmer. Symbolic debuggers are very helpful during this period, but there are some bugs that are not found by this method. Assertions should be spread liberally throughout the code, so they can can aid in finding difficult "bugs" at this time. After the code is tested, extra assertions can be removed. Suggestions for doing this will be in Section 3.4.

3.3 TESTING

The basis for this methodology for testing fault-tolerant software lies primarily in expansion of the module and software system testing phase, so that comprehensive testing is done before implementation of fault-tolerant techniques is in place. By doing this, error detection interference - from masking of errors, etc. - is minimal. Although the use of assertions throughout all phases is important, it is their use for comprehensive testing at module and system integration that allows greater coverage and more sophisticated error detection during testing. Assertion testing may be used with any test case generation technique. It works especially well with grid and adaptive test generation, because it allows tests to be both run and evaluated automatically [Andrews 85]. It is this automation of evaluation of test results that permits expansion of the number of tests that are run (and therefore of the comprehensiveness of the testing).

There is a second no less important reason for performing comprehensive testing at this stages, and that is because of the difficulty in testing in a simulated real-time environment (outlined in Sec. 2.3). If most errors are detected prior to execution of the code on a simulator, then fewer tests need be run. It is obvious that real-time code must be checked at some point in a real-time environment, but it is possible to write timing assertions that can locate some timing errors before code is run on a simulator [Mahmood 84a].

The recommended procedure for module and system testing of software is as follows:

* **individual module tests-** whenever possible generate test cases exhaustively. This is feasible in three cases: when most of the variables are Boolean (either 1 or 0), when there are a small number of input variables, or when the range of possible values for the input variables is small.

* **module integration tests-** generate test cases in a grid pattern to ensure uniform coverage. Choose for perturbation those variables having the most influence on the output.

* **system integration tests-** generate test cases using another test case generation methodology, such as random, functional, etc., or continue with a grid pattern to ensure uniform coverage. Choose for perturbation those variables having the most influence on the final system output.

Testing in the next phases, in a simulation environment or as an embedded system, is the same as usual except for the number of assertions. The number of assertions depends on the phase of testing. When used for module or system testing, assertions should be spread liberally throughout the software to make it easy to locate the errors. When the software is ready for testing in a simulation environment or as an embedded system, the number of assertions must be reduced so that memory space and execution time overhead are minimized. If assertions will be used during deployment, the procedure (outlined in the next section) for choosing which assertions to keep should be followed. If all the assertions will be removed during deployment, then the easiest plan for this part of the testing would be to remove the ones that detected the least number of errors.

3.4 DEPLOYMENT

When assertions are used for error detection in implementation of fault tolerance techniques, minimization of assertions (and the consequent overhead) is also important. A reasonable assumption would be that those assertions shown to be effective in error detection during the testing phase would be most able to detect intermittent and transient hardware faults, as well as any new software errors that might be introduced during maintainance.

One way of improving the selection process of assertions would be to subject the software to an error seeding process (as was done in the experiment in testing fault-tolerant software) and then retain a covering set of assertions, that is, the set detecting all seeded errors. In the research experiment, three assertions (out of the nineteen that were written) would have detected all the detectable errors. The implication of these results is that it may be possible to find a small subset of assertions capable of detecting a large number of errors, so space and time overhead can be minimal.

The placement of the assertions is also dependent on the testing phase. During the early debugging phase, it is most desirable to have many assertions to check incoming data, outgoing commands, data storage and retrieval, and the results of computations. The analysis showed that the effective and essential assertions were in the last part of the asserted code (the procedures that calculate the final commands to the ailerons). This is not surprising since assertions placed earlier in the code would not catch errors introduced later on. During deployment, therefore, assertions placed in the procedures that calculate the final commands will probably be the most effective for detecting errors.

To ensure that the greatest number of variables are directly or indirectly tested, the dependency factor for each variable should be calculated and the variables with the highest dependency number should be included in the assertions [Andrews 86]. This relationship between assertion effectiveness and the data dependency factor of the variables being tested should be of considerable help in writing good assertions for detection of software or hardware faults.

3.5 MAINTENANCE

During the maintenance phase, errors are corrected or enhancements are implemented. After any changes, the code must be retested. This phase is one of the most discouraging for programmers, primarily because it is often so difficult to find anyone who knows anything about the original code. It is much more of a problem to work on code written by someone else than to start over.

The use of assertions should not be overlooked during maintenance. Not only do they provide a form of documentation to help the programmer, but they also simplify the retesting procedure. Since assertions can be conditionally compilable, they are easily put back in the code when needed again. When a software system is very large, recompilation of the entire code is not feasible each time a change is made. In this case, a separate version containing all the original assertions could be maintained for use during retesting. The reason this is important is that there is always the chance of error masking when redundancy of a fault-tolerant system is still in place. Therefore, module and system testing must be performed with redundancy disabled to make sure errors are not covered up. Assertions used for testing in the earlier phases can help locate new errors, as well as notify of problems in other areas that may have been affected by updating the code.

4 VARIATIONS IN TESTING SCENARIOS

There are many levels in which the methodology, as outlined in Section 3, can be used for testing fault-tolerant software. These variations will be presented in order, from the minimal levels to the more complex. Presumably the more critical the application, the more intensive the testing scenario.

1) Use assertions vigorously during software implementation for debugging and then strip out of the code for remaining testing phases. This is done by making assertions conditionally compilable. In this way they can be used again during maintenance to make sure a change in code in one area does not affect another area.

2) Use assertions for automation of the testing process during module and system testing so a large number of test cases can be executed.

3) Use assertions to supplement other methods of error detection, including hardware redundancy and proofs of correctness. A 4-5% overhead can provide detection of errors that is worthwhile [Andrews 78].

4) Use assertions with recovery blocks to provide fault tolerance for hardware and software errors. A 10 - 15% overhead can provide detection as well as good recovery [Andrews 79].

5) Use assertions liberally and use a watchdog processor [Mahmood 85] or a watchdog task [Ersoz 85] to reduce the overhead of executing the assertions during deployment.

5. SUMMARY

A methodology for testing fault-tolerant software has been presented. Factors contributing to the development of this methodology have been discussed, including the desirable characteristics of such a methodology, various aspects of software testing, and problems inherent in testing fault-tolerant software. The testing methodology for each stage indevelopment and testing a real-time, fault-tolerant software system is outlined. This covers the specification/requirements/design phase; implementation of the code; all cycles of the testing cycle (individual module tests, module integration tests, and system integration testing); deployment; and maintenance. Variations in the testing scenarios are also described as a help in making choices about how much testing can be done.

ACKNOWLEDGEMENT

This work was supported in part by NASA-AMES Research Center under Grant No. NAG 2-246. The authors would like to thank Roger Tapper of Rockwell International, and Dr. Dallas Denery, Jim Saito, and Doug Doane from NASA-AMES Research Center for their cooperation and interest. Appreciation is also due to the members of the Center for Reliable Computing at Stanford University for their suggestions.

REFERENCES

[Adrion 82] Adrion, W.R., M.A. Branstad and J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," ACM COMPUTING SURVEYS, Vol. 14, No. 2, pp. 159-192, June 1982.

[Andrews 83] Andrews, C.L., "RXVP80/COBOL: The Verification and Validation System for COBOL," Proc. IEEE SOFTFAIR, Wasshington, D.C., July 1983.

[Andrews 78] Andrews, D.M., "Software Fault Tolerance Through Executable Assertions," Proc., 12TH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS AND COMPUTERS, Asilomar, California, Nov. 1978.

[Andrews 79] Andrews, D.M., "Using Executable Assertions for Testing and Fault Tolerance," Proc., 1979 INT'L CONFERENCE ON FAULT-TOLERANT COMPUTING (FTCS-9), Madison, Wisconsin, June 20-22, 1979.

[Andrews 81] Andrews, D.M., and J. Benson, "An Automated Program Testing Methodology and Its Implementation," Proc., 5TH ANNUAL CONFERENCE ON SOFTWARE ENGINEERING, San Diego, California, March 9-12, 1981; Reprinted in Tutorial: SOFTWARE TESTING & VALIDATION TECHNIQUES, 2nd Edition, IEEE Computer Society Press, 1981.

[Andrews 85] Andrews, D. M., "Automation of Assertion Testing: Grid and Adaptive Techniques," Proc. HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS - 18), Honolulu, Hawaii, January 2-4, 1985.

[Andrews 86] Andrews, D. M., A. Mahmood, and E.J. McCluskey, "Dynamic Assertion Testing of Flight Control Software," Proc. HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS - 19), Honolulu, Hawaii, January 8-10, 1986.

[Budd 80] Budd, T.A., R.A. DeMillo, R.A. Lipton, and F.G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," Proc. 7TH. ACM SYMP. PRINICIPLES OF PROGRAMMING LANGUAGES, Jan. 1980.

[Cooper 76] Cooper, D.W., "Adaptive Testing," Proc. SECOND INT'L CONFERENCE ON SOFTWARE ENGINEERING, San Francisco, California, Oct. 13-15, 1976.

[De Millo 78] De Millo, R.A., R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," COMPUTER, Vol. 11, pp. 34-43, Apr. 1978.

[DFCR-96 80] L-1011 DAFCS Software description, DFCR-96R1, L-1011 Digital Flight Control System Report, Collins Avionics Division, Rockwell International, 1980.

[Duran 84] Duran, J.W., and S.C. Ntafos, "An Evaluation of Random Testing," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-10, No. 4, July 1984.

[Ersoz 85] Ersoz, A., "The Watchdog Task: Concurrent Error Detection using Assertions in ADA," CRC Technical Report No. 85-8, CSL TR No. 85-267.

[Gannon 79] Gannon, C., "Error Detection Using Path Testing and Static Analysis," COMPUTER, Vol. 12, Aug. 1979.

[Howden 80] Howden, W.E., "Functional Program Testing," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-6, No. 2, Mar. 1980.

[Mahmood 84a] Mahmood, A., D.M. Andrews and E.J. McCluskey, "Executable Assertions and Flight Software," Proc., 1984 IEEE/AIAA 6TH DIGITAL AVIONICS SYSTEMS CONFERENCE, Baltimore, Maryland, Dec. 3-6, 1984.

[Mahmood 84b] Mahmood, A., D.M. Andrews and E.J. McCluskey, "Writing Executable Assertions to Test Flight Software," Proc., 18TH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS AND COMPUTERS, Pacific Grove, California, Nov. 4-7, 1984.

[Mahmood 84c] Mahmood, A., D. M. Andrews and E. J. McCluskey, "Executable Assertions and Flight Software," CRC Technical Report No. 84-16, CSL TR No. 84-258.

[Mahmood 85] Mahmood, A., and E. J. McCluskey, "Watchdog Processors: Error Coverage and Overhead," Proc., 15TH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING (FTCS-15), Ann Arbor, Mich., June 19-21, 1985.

[Miller 75] Miller, E.F., and R.A. Melton, "Automated Generation of Test Case Data Sets," Proc. 1975 INT. CONF. RELIABLE SOFTWARE, 1975.

[Ntafos 85] Ntafos, S.C., "An Investigation of Stopping Rules for Random Testing," Proc. HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS - 18), Honolulu, Hawaii, Jan. 2-4, 1985.

22