DAA/Ames

DYNAMIC ASSERTION TESTING OF FLIGHT CONTROL SOFTWARE

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

HICSS-19

July 1985

(NASA-CR-176715) DYNAMIC ASSERTION TESTING OF FLIGHT CONTROL SOFTWARE (Stanford Univ.) 25 p HC A02/MF A01 CSCL 09B

N86-23321

Unclas

G3/61 15558

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.

Copyright (c) 1985 by the Center for Reliable Computing, Stanford University. All rights reserved, including the right to reproduce this report, or portions thereof, in any form.

APR 1986
REC
NASA STI FACILITY
ACCESS DEPT.

DYNAMIC ASSERTION TESTING OF FLIGHT CONTROL SOFTWARE

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

HICSS-19

JULY 1985

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

ABSTRACT

This report describes an experiment in dynamically testing flight software with assertions. Digital flight control system software was used as a representative test case. The experiment showed that 87% of typical errors introduced into the program would be detected by assertions, thus demonstrating that assertion testing would provide a good basis for a flight software testing methodology. Detailed analysis of the test data showed that the number of assertions needed to detect those errors could be reduced to a minimal set. The study also revealed that the most effective assertions provided greater collateral testing of the program parameters than those assertions detecting fewer errors.

Index Terms: Software Testing, Fault Tolerance, Testing Methodology, Flight Software.

1. INTRODUCTION

The Center for Reliable Computing at Stanford University has completed a two year study on the application of executable assertions to testing flight software. The purpose of this study is to develop a methodology for testing real-time flight software. Prior to this study, the results from other research projects had already demonstrated the practicality and effectiveness of assertion testing [Andrews 78-81]. However, even though assertions had proved to be successful in detecting errors in other types of software, it was important to show they could be used for testing flight software because it has different characteristics. (Flight software is real-time, has many logical variables, and uses fault- tolerant techniques, such as, voters and limiters built into the software.)

In order to demonstrate the effectiveness of assertions in detecting errors in flight software, an experiment was conducted using Digital Flight Control System (DFCS) software as a test case [DFCR-96 80]. Assertions were written and embedded in the code, then errors were inserted (seeded) one at a time and the code was executed. The results from this experiment showed that 87% of the errors introduced into the DFCS program would be detected by assertions.

Following the experiment, analysis of the research results demonstrated the following:

* Assertions are effective in detecting errors in digital flight

control system software.

* Assertion testing can reduce the time and cost of testing flight software in simulators and in actual flight.

* The variables that are most dependent on other variables provide the greatest collateral testing and, therefore, the assertions that test the most dependent variables are the most effective and detect the largest number of errors.

* Placement of assertions is an important factor in determining the effectiveness of an assertion, since those assertions placed at the end of modules detected the most errors.

* Assertions can be used as a basis for implementing fault-tolerant techniques in flight software because they have an excellent error detection rate and can have a low overhead.

* Assertions can be executed independently of the main processor by using a separate "watchdog" processor to reduce the overhead of assertion testing.

The fact that assertion testing proved to be effective for flight software has far reaching implications. The major one is that assertion testing can be used to eliminate errors at an earlier stage in the development cycle than before. Testing flight software has been extremely costly and time consuming because the elimination of errors has been done primarily by using simulators followed by actual flight

testing. If the number of simulations and flights can be reduced because errors are detected sooner, there should be a considerable reduction in time and money spent on testing.

Many potential applications for assertions have not yet been fully explored. Due to their excellent error detection capability, assertions can provide detection for faults that can be corrected by fault tolerance techniques incorporated in the software. The use of assertions, however, is not limited solely to testing or fault-tolerant applications. System specifications and requirements can be expressed in assertions before implementation of the code as an aid in writing assertions, as well as in providing documentation throughout the entire software cycle.

In this paper, background information about assertion testing is presented first, then a description of the experiments, followed by a discussion of the research results and a conclusion.

2. ASSERTION TESTING

Assertion testing is a technique for dynamically testing software by adding additional statements, called assertions, to 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 true, then the condition is true at that particular point in the program; if it is false, then an error has occurred. Notification of the error is most often made in an output message, such as, "Assertion in module <xxxx> at statement # <nn> is false."

Assertions are written in the same language as the software, but they usually have a slightly different format (typically beginning with the word ASSERT) so they can be distinguished from the rest of the software. Before the program can be executed, the assertions must be translated into code that is acceptable by the compiler. This translation is done by a preprocessor, program analyzer, or a pre-compiler. Assertions are frequently made conditionally compilable, so they can be turned into comment statements and easily stripped out of the code after testing is complete.

Assertions may be placed appropriately throughout the software, although sometimes they need only be added to certain strategic modules and still retain their effectiveness. An assertion can test the relationship between one or more variables, the range or limit of a variable, or check the results of a numerical computation. Some

examples of assertions are:

```
ASSERT (ABS (LAT_INN_CMD) > MAX_CPL)

ASSERT (ABS (K2 - 0.95133) > 0.0005)

ASSERT (ABS ((LAT_INN_CMD) - 0.5 * (RL5 + 0.753 * ROLL)) > 0.0001)
```

2.1 PROCEDURE FOR ASSERTION TESTING

Assertion testing differs from other forms of dynamic testing of software because assertions must be added to the code before it is executed. Assertion testing has two distinct advantages over other testing methods: First, determining the correctness of the output is remarkably simplified because of the automatic notification of an error when an assertion is violated. Second, because of this reduction of time required for assessment of test results, the generation of a larger set of input data becomes possible and automation of the process of adaptively generating test data becomes easier to implement [Andrews 81,85], [Cooper 76]. However, the generation of input test data can be the same as is used in any other testing procedure [Adrion 82], [Duran 84], [Gannon 79], [Howden 80], [Ntafos 85]. The procedure in testing software with assertions is as follows:

* Add assertions to the code - preferably this should be done during code implementation.

* Execute the code to test the correctness of the assertions.

* Generate test case data automatically or by the usual testing

methods.

* Input test data and execute the software.

* When testing is complete, assertions may be removed or left in the code during deployment.

2.2 FAULT-TOLERANT APPLICATIONS

Another important use of assertions is in building fault-tolerant systems [Randall 75], [Andrews 79]. A designer of a fault-tolerant system assumes that faults will occur and tries to prevent failures by incorporating methods for error detection and correction during system operation. Assertions embedded in the software provide a convenient and effective way to implement on-line fault tolerance for hardware faults, as well as software errors. Assertions are used to detect the errors, and additional code (traditionally referred to as a recovery block) provides a way to handle the error. When an assertion is evaluated as false, control is transferred to the recovery block statements that are then executed. This technique, although simplistic in concept, allows implementation of a variety of responses to potentially critical problems.

Due to the increasing criticality of computer applications, it has become necessary to provide recovery from software, as well as hardware errors. Even with state-of-the-art program validation and verification, there can be no guarantee that the software is correct and free of errors. The complexity of a software system is at least an order of

magnitutude greater than that of the hardware because of the enormous number of different states in a program. This makes it possible for an error that will only surface under a rare combination of input values to remain undetected. Therefore, it is not surprising that residual errors in software have been a major source of system failures [Losq 77].

Although implementation of tolerance for hardware faults has been commonly used in the past, there are many reasons why it is becoming even more important. For one, the problems of adequately testing hardware are increased drastically by development of microelectronic systems, including submicron devices, Very Large Scale Integrated (VLSI) circuits, and large-scale electronic systems. Second, not all hardware faults are detectible and, when more than one fault is present, detection becomes more difficult. Because assertions can check that data is within an acceptable range of values, they can easily detect some types of hardware faults. For example, a typical problem that can have catastrophic consequences is a transient hardware fault. If the problem is due to a faulty sensor, the bad data can be discarded and a second data point could be read or another could be calculated based on the previous reading and the rate of change. Such simple procedures can prevent undesirable consequences that can result from intermittent or transient hardware faults [Andrews 79].

3. RESEARCH EXPERIMENT

This section describes the flight control software used as a test case and the procedure followed in developing the assertions and generating the errors.

3.1 TEST CASE SOFTWARE

The software used as a test case was the autopilot code for a large, wide-bodied commercial airplane. It is a good example of Digital Flight Control System software and is written in AED (Automated Engineer Design) [DFCR-96]. The software was written incrementally over the past decade and most of the "bugs" have been corrected. The code is almost identical to that in use at the present time. (It is the next to the last version before deployment.)

The software is an integrated system that provides autopilot and flight director modes of operation for automatic and manual control of the plane during all phases of flight. The software is partitioned into five major categories: the first, of course, is control and navigation of the plane. In addition to this, are various supporting functions, namely, testing and voting, logic (engage and mode calculations), input/output (data handling, transmission, display, etc.), and the executive. The subset of modules chosen for testing calculates the commands to the ailerons. They use the selected heading and data from sensors as input. Figure 3.1 shows the relevant procedures and the flow of data.

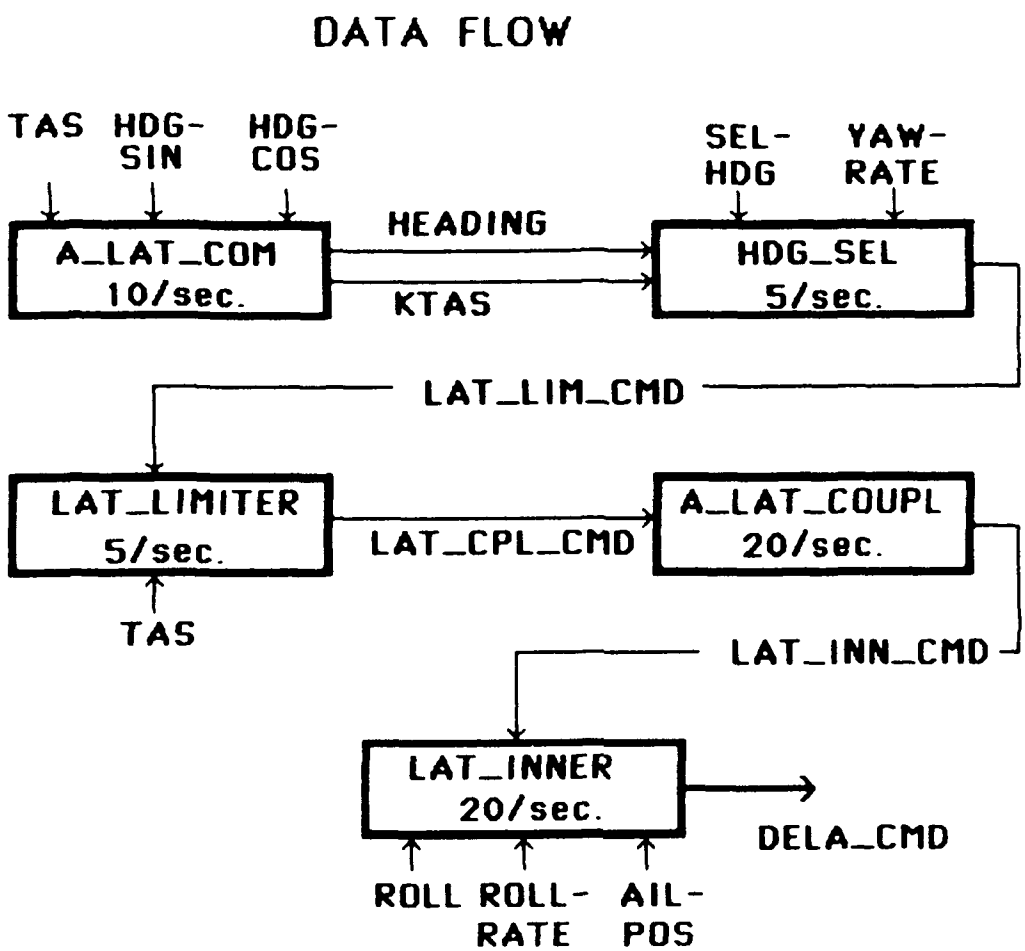

Fig 3.1 Flow of Data in Test Case Software

3.2 PROCEDURE

The original plan for the experiment was to add assertions, put in errors, and execute the autopilot code on the flight control computers installed at the Digital Flight Control Systems Verification Laboratory at NASA-AMES [de Feo 82]. However, developing assertions involves a certain amount of experimentation in order to refine them and measure

the desired condition. In addition, the errors were to be seeded one at a time so it would be possible to determine whether or not a particular error had been detected. Each change in the code, for refinement of assertions or inserting an error, requires recompilation of the entire program by an AED compiler which is on a Univac computer at a different location. Then the excutable code must be downloaded into the flight computers on the pallet. It soon became apparent that the process of making changes to the code was so time consuming that very fews runs could be made in one day. For this reason, the code was rewritten in Pascal so it could be executed more efficiently on the DEC-20 at the Stanford University campus.

There were two other even more important reasons for moving the code to another computer. One was that introducing errors into the code often caused the flight computers installed on the pallet to "crash" (or not fly at all) because the effect of the error was so drastic. Consequently, the section of code containing assertions was never executed. The other reason was intrinsic to the nature of the flight computers which have a dual-dual redundancy architecture. Aberrations are corrected by voters and limiters built into the software [de Feo 82], so errors introduced in the software running on one channel would be "corrected" by the voters or limiters before detection by an assertion.

In this experiment, the assertions were written by one person and the errors by another person. The reason for doing this was to maintain

complete independence. Since existing documentation did not contain enough information to write assertions, the flight computers were run on the simulator in conjunction with a strip chart recording device to determine the normal values of the program variables. From this information, it was possible to write assertions for the set of modules to be tested. More detailed information may be found in the following: a description of the experiment to test flight software with assertions [Mahmood 84a]; suggestions for writing assertions in flight software gained from this experience [Mahmood 84b]. A combination of these two papers along with additional information was published as a technical report of the Center for Reliabile Computing at Stanford University [Mahmood 84c].

The selection of errors was taken from two studies of errors made during development of flight control software [Hecht 82]; one was remarkably similar to the software we were using as a test case. Errors, chosen from four different classifications, were seeded one at a time in the software to determine the effectiveness of assertions in finding errors of different types. Effort was made to duplicate exactly the original errors whenever enough information was available.

4. ANALYSIS OF RESEARCH RESULTS

This research study can be divided into three phases: the first was the original software testing on the flight simulators installed at the NASA-AMES Research Center; the second was conducting the tests on the DEC-20 at Stanford University; and the third was exploration of the factors affecting the effectiveness of the assertions themselves. The results from each of these phases contributed to understanding the problems of testing flight software. This section describes the results from each phase.

4.1 FLIGHT SIMULATOR TESTING

The results of the first phase, although not productive in quantitative results (because of the length of time required to run each test), contributed greatly to understanding the problems involved in testing real-time flight software.

The first results clearly showed that testing a software system with built-in redundancy (that is, a fault-tolerant system) is not possible using normal testing techniques. These results also indicated that the same problems encountered in testing fault-tolerant hardware systems (fault masking, etc) exist for testing fault-tolerant software systems and that "design for testability" features should be incorporated into fault-tolerant software design specifications.

When the software was executed on flight computers in a simulated real-time flight environment, the following major differences between

real-time fault-tolerant software and non-real-time software without redundancy were identified:

* 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.

* 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, most errors are masked and become difficult to detect, since propagation of errors is halted.

* 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.

From these results, it was clear that it would be necessary to incorporate into a flight software test methodology some of the same concepts that have been proposed for simplifying the testing of fault-

tolerant hardware; such as, building observation points into the software to break the system into manageable partitions, removing redundancy during testing, and removing internal automatic channel synchronization. Therefore, in the subsequent testing, the program was tested as a single entity (without redundancy and synchronization).

4.2 SIMULATION ON DEC-20 COMPUTER

The initial test runs on the flight computors revealed major differences between real-time flight software and non-real-time software. More comprehensive testing done in this phase indicated that, regardless of these differences, assertion testing of Digital Flight Control System software is an effective method for detecting errors.

Eighty one errors were seeded in the program one at a time to determine the effectiveness of assertions in finding errors of different types. The errors were from four different error classifications - data handling, logic, database, and computational. As Fig. 4.1 shows, nearly 70% of the errors were detected and, if all paths had been asserted, nearly 90% of all errors would have been detected. Assertions were not written for the parts of the code that were not supported by the flight simulator. Some errors (especially logic errors) caused execution of the code without assertions and, consequently, were not detected. The reason the remaining errors were not detected was due most frequently to the fact that they had no effect on the computations. For example, Boolean variables (having values of either 0 or 1) are typically assigned a value in flight software in statements such as,

MODE = A or B or C and not D. Suppose A equals 1, then an error resulting in a change in value of B or C will have no effect on the outcome of this assignment statement and therefore would not be detected by an assertion. In another example, some errors changed the name of a Boolean variable into another. When the value of the variables was identical, the error could not be detected.

EXPERIMENTAL RESULTS

ERROR TYPE	No. INSERTED	% ERRORS DETECTED	
		PARTIALLY ASSERTED	FULLY ASSERTED
DATA HANDLING	22	63.6	90.9
LOGIC	19	47.3	84.2
DATABASE	19	78.9	94.7
COMPUTATIONAL	21	76.1	80.9
TOTAL	81	66.6	87.6

Fig 4.1 Types of Errors Detected by Assertions

4.3 OPTIMIZATION OF ASSERTION TESTING

Once the effectiveness of assertion testing of flight software was established, it became necessary to explore various aspects of optimizing the use of assertions in order to develop an efficient testing methodology. Efforts were directed toward answering questions about both the qualitative and quantitative aspects of assertion testing. For example, how should assertions be written, what type of assertions are the most effective, where is the best placement for assertions, how many are needed, etc.

4.3.1 Writing the Assertions

From the difficulty realized in this experiment in trying to write assertions with little knowledge of the program behavior and inadequate software specifications, it is obvious that assertions must be written in cooperation between a flight system analyst and a software person who is designing or implementing the code. Some of the conditions that should be tested by assertions would best be known to flight specialists; and for that reason, it is imperative to have their help and guidance. The best time to add assertions is during the original coding, so the assertions will detect errors during module, as well as system integration testing.

4.3.2 Number of Assertions

The number of assertions depends on the phase of testing. When used for debugging, assertions should be embedded frequently throughout

flight software code so they can best pinpoint the location of the errors. However, once the software is ready for testing in a flight simulation environment, then a smaller number of assertions is desired in order to minimize memory space in the computer and execution time overhead.

This is also true when assertions are used for error detection in implementation of fault tolerance techniques. In that case, the suggested procedure is to seed the program with errors (as was done in this experiment) and then retain a covering set of assertions, that is, the set detecting all seeded errors. The assumption would be that those assertions would be most able to detect intermittent and transient hardware faults, as well as any new software errors that might be introduced during maintainance.

Analysis of the test results showed that four errors were detected by only one assertion. One assertion detected one of those errors and another detected the remaining three errors. These two assertions constitute the set of "critical" assertions, that is, those assertions that were necessary if all errors were to be detected. Of the remaining assertions, either one of two assertions would detect all of the errors not detected by the two "critical" assertions. This means that out of all the assertions written, three assertions could be used to detect all the detectible 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. This

result makes assertion testing even more attractive. Nevertheless, suggestions for alternate methods of executing assertions in parallel may be found in [Saib 77], [Mahmood 85d], [Ersoz 85].

4.3.3 Placement of Assertions

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 critical assertions were in the last two procedures. This is not surprising since assertions placed earlier in the code would not catch errors introduced later on. Although at first it appeared that most of the errors would be corrected by the limiters built into the software, this result demonstrated that many errors do escape those built-in protections and that assertions can detect those errors when the software system is tested as a single entity (with the redundancy disabled). In the testing phases where execution time computer space are an important factor, then assertions should be placed in the procedures that calculate commands to the mechanical parts of a flight system.

4.3.4 Most Effective Assertions

Assertions can be different types. They can measure the relationship between variables, check for maximum or minimum allowable values of a variable, or perform a numerical computation with a

different algorithm to determine correctness of the calculation. Examples of each of these types of assertions were given in Section 2. Assertions also can have tests for more than one variable, and a factor for time may also be included in the assertion. All types of assertions were written for this experiment and, interestly enough, none of these factors - type of assertion, inclusion of a factor for time, or checking multiple variables - seemed to affect the ability of the assertion to detect errors.

Further in depth analysis did reveal a factor that appears to influence the effectiveness and criticality of an assertion. It seemed possible that testing certain variables might be more effective than testing others - depending on which variables provide greater collateral testing. One measure of collateral testing is the number of variables that are utilized in assigning a value to a variable. This number is refered to as the "data dependency" of that variable. The variables with the highest "data dependencies," therefore, would be expected to provide the greatest collateral testing of other variables.

This theory was tested against the results from this experiment. A high correlation was found between the data dependency of the variables tested in an assertion and the effectiveness of the assertion. Those assertions with the highest accumulated data dependency factors (for the variables included in the assertion) proved to be the most effective in detecting errors. The difference in detection effectiveness was significant, since they detected ten times as many errors as the

assertions with the lowest dependency factors. Not only did the most effective assertions have the highest data dependency factor, but the two critical assertions also had very high dependency factors.

Therefore, to ensure that testing covers as many of the variables as possible, the dependency factor for each variable should be calculated and the variables with the highest data dependency number should be included in the assertions. Discovery of this relationship between assertion effectiveness or criticality and the data dependency factor of the variables being tested should be of considerable help in writing good assertions for flight software.

5. CONCLUSION

The broad and important conclusion to be reached from this research investigation is that assertion testing is an effective and efficient method of detecting errors in flight software. A major implication of this result is that assertion testing can be used effectively to eliminate most errors at an earlier stage in the development cycle than before. Testing flight software has been extremely costly and time consuming, because the elimination of errors primarily has been done using flight simulators as well as actual flights. If the number of simulations and flights can be reduced because errors are discovered sooner, there should be a considerable reduction in time and money for testing.

Therefore it is proposed that assertions be added to the software during implementation and that assertion testing be utilized from the beginning to shorten the testing cycle. Furthermore, in fault tolerant-computing applications, the suggested procedure is to retain the assertions during deployment and include additional code to provide error recovery. One of the conclusions reached as a result of this experiment is that the number of assertions required to detect all possible detectable errors may be a small, minimal set - therefore making assertions a useful medium for providing fault tolerance in flight software.

Assertions should be written as a cooperative project by a flight systems specialist and a software person. According to these test results, effectiveness of an assertion was not affected by factors such as checking multiple variables, inclusion of a factor for time, etc. However, testing program variables that provided the greatest collateral testing of other variables seemed to improve the effectiveness or criticality of an assertion.

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 with this experiment. Appreciation is also due to the members of the Center for Reliable Computing at Stanford University for their suggestions.

6. 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 85] Andrews, D. M., "Automation of Assertion Testing: Grid and Adaptive Techniques," Proceedings of the Hawaii International Conference on System Sciences (HICSS - 18), Honolulu, Hawaii, January 2-4, 1985.

[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 9] 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 78] Andrews, D.M., "Software Fault Tolerance Through Executable Assertions," Proc., 12TH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS AND COMPUTERS, Asilomar, California, Nov. 1978.

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

[de Feo 82] de Feo, P., D. Doane, and J. Saito, "An Integrated Usser-Oriented Laboratory for Verificationof Digital Flight Control Systems - Features and Capabilities," NASA Technical Memorandum 84276, Ames Research Center, Moffett Field, CA., 94035, Aug. 1982.

[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., "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-xx, CSL TR No. 85-xxx.

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

[Hecht 83] Hecht, H. and M. Hecht, "Trends in Software Reliability of Digital Flight Control Systems," NASA Technical Report No. 166456, Ames Research Center, Moffet Field, CA., 94035, Apr. 1983.

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

[Losq 77] Losq, J., "Effects of Failures of Gracefully Degradable Systems," Proc., SEVENTH ANNUAL CONFERENCE OF FAULT-TOLERANT COMPUTING, Los Angeles, CA, June 1977.

[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.

[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.

[Ntafos 85] Ntafos, Simeon C., " An Investigation of Stopping Rules for Random Testing," Proceedings of the Hawaii International Conference on System Sciences (HICSS - 18), Honolulu, Hawaii, Jan. 2-4, 1985.

[Randall 75] Randall, B., "System Structure for Software Fault Tolerance," PROC. INT'L CONFERENCE ON RELIABLE SOFTWARE, Los Angeles, California, Apr. 1975.

[Saib 82] Saib, S.H. and S.W. Smoliar, "Software Quality Assurance for Distributed Processing," 15 HAWAII INT'L CONFERENCE ON SYSTEM SCIENCES, Honolulu, Hawaii, Jan. 6-8, 1982.

[Saib 79] Saib, S.H., "Verification and Validation of Avionics Simulations," Avionics Panel on "Modeling and Simulation of Avionics Systems and Command, Control, and Communications Systems," Paris, France, Oct. 15-19, 1979.