A New Method for the Verification of Fault Tolerant Software

Ann Tai, Myron Hecht, and Herbert Hecht SoHaR Incorporated,
8500 Wilshire Blvd. Suite 1027, Beverly Hills, CA

KEY WORDS: Fault Tolerant Software, Software Verification, Software Reliability, Condition Tables

ABSTRACT: This paper presents a new approach for verification of fault tolerant software in aerospace applications. The approach, called the "Enhanced Condition Table", integrates the merits of functional and structural testing in a single framework. The goal is to generate a reasonably sized set of test cases that will reveal operationally significant defects in the software. The method starts with the generation of a conventional condition table based on analysis of the specification. It then proceeds through a definition of possible failure modes by means of fault trees, and finally, the definition of sequences of values for testing of loops.

Introduction

Verification is the attempt to remove faults in a program, usually by executing it in a test or simulated environment. Fault tolerant software poses significantly greater problems for verification than conventional software because the (1) fault detection and recovery provisions must be demonstrated to work in the presence of faults and (2) the fault tolerance provisions must be proved not to interfere with the normal functioning of the system in the absence of a failure condition. This paper describes a new method for verification of fault tolerant software based on condition tables which address these problems.

Testing is usually categorized as either functional or structural. Functional testing develops test cases from the perspective of what a software module is required to do. Functional testing regards internal structure and behavior of a software module as a black box, and is therefore frequently referred to as "black box testing".

Structural testing develops test cases based on how a software module has been constructed. Because the internal structure of the code is relevant to the development of test cases, structural testing is sometimes referred to as "white box testing". Most structural testing methods do not require a knowledge of the functional requirements of a module in order to develop test cases. There are numerous forms of structural testing which have been cited in the literature [MILL81, SHOO83, HOWD78]. Among the most frequently mentioned are branch testing and path testing. Branch testing requires that each branch in a program be tested at least once. It is one of the simplest and best known approaches to systematic testing. Path testing involves the testing of every path through a program at least once. Since most programs will have a very large, if not infinite, number of paths, the method is of theoretical rather than practical importance.

When used independently, both functional and structural testing methods have significant limitations. Functional testing can only demonstrate that the module has correctly executed its function under a specific set of test conditions. If the conditions fail to exercise defective branches of the code,

no faults will be revealed. Thus, functional testing can not be used to verify that a software module provides all the desired functions and does not carry out any unintended functions in the absence of exhaustive testing of the entire input space [LISK86]. Because the number of test cases for exhaustive testing is infinite or impractically large, most functional testing can not provide this assurance. Although a structural testing program can in principle be designed to provide complete or nearly complete coverage of branches or paths, it can not be relied upon to reveal data dependent defects that may be due to incorrect conditions (on a branch with multiple conditions) or other problems.

The limitations of structural or functional testing when used alone has resulted in the observation that they should be used in a complementary rather than exclusive manner [HOWD78, HOWD82]. The Enhanced Condition Table (ECT) described in this paper is based on this observation.

The ECT is an extension of the condition table developed by Goodenough and Gerheart [GOOD75] that considers failure modes and specifies sequences of executions. A condition table is a tabular representation of conditions and combinationsof outcomes at each decision point in a program. The rows of the condition table contain all conditions of a particular program segment (e.g., A > B, x < 1, etc.). The columns specify a particular combination of outcomes of the conditions stated in the rows and are called "rules". Each outcome in a rule can have one of five values: TRUE (e.g., A is greater than B), FALSE (e.g., A is less than or equal to B), NECESSARILY TRUE (A must be greater than B because of the data dependency), NECESSARILY FALSE (A must be less than B), and DON'T CARE (the nature of the code is such that the conditions may be true or false without affecting the output). They are represented in the table as y, n, (y), (n), and - respectively. Test case generation is guided by the condition table, and complete coverage is achieved when all rules are tested.

The original condition table methodology is based only on the structure of the program being verified. The Enhanced Condition Table (ECT) utilizes knowledge of functional requirements, analysis of the program text, knowledge of possible faults, and selection of special values for test data. The goal of the approach is to find a reasonably small set of tests that approaches the coverage obtained through exhaustive testing.

Verification Using an ECT

The process of developing an ECT and its use for verification consists of five steps:

1. Construct a preliminary version of the condition table based on the specification.

2. Refine the preliminary version of the table based on the code. The conditions should be modified according the data structures used by the code. Based on the implementation, it may be necessary to add more conditions. The refining process itself is often effective by

focusing a tester's attention on the inconsistencies between the code and the specification. Any discrepancies signal possible software faults.

3. Develop a fault tree analysis to determine significant functional failures and consider how these functional failures impact conditions (i.e., rows) and DON'T CARE entries to force selection of test data that check these error possibilities in the table. This process may result in additional conditions, decomposition of original conditions (e.g., a >= b may be decomposed into two conditions, a > b and a = b in order to generate more specific test cases).

4. Develop rule classifications and determine sequences of execution of these classifications for sections of the code containing loops.

5. Generate test cases so that all rules on the ECT are exercised.

The remainder of this section illustrates ECT-based verification using a portion of the operating system of the Software Implemented Fault Tolerant (SIFT) Computer operating system [PALU85, WENS73] as an example. The SIFT computer is an experimental flight control computer consisting of 8 loosely coupled processing modules (CPU, memory, and I/O hardware) which communicate with each other via serial lines. No central memory, clock, or control hardware is used. For each application task, several processors (the specific number depends on the criticality of the task) perform the same calculation in parallel, and all processors vote on the output. A large degree of hardware and some software fault tolerance is achieved by this voting scheme. The SIFT operating system is replicated in all processors, and performs task scheduling, dispatching, interprocessor synchronization, interprocessor communication, and fault detection and isolation.

The specific segment used as an example is the SIFT Fault Isolation Task. At periodic intervals referred to as "frames", the task determines which processors are faulty and should be retired (i.e., isolated from the system). This fault detection and recovery function is representative of redundancy and system management functions needed in future spaceborne fault tolerant computers.

The inputs to the Fault Isolation Task are "error reports" produced by each of eight processors based on the results of their own local vote and which are sent to all other processors through the interprocessor communication hardware. The error reports are a single byte in which each of the 8 bits corresponds to a processor; in the terminology of SIFT, this type of data structure is referred to as a "bitmap". If the local vote indicates that a specific processor has disagreed with the majority, the corresponding bit is set to 1 in the error report. Figure 1 (a) shows an example error report in which Processor No. 5 is marked as faulty. The Fault Isolation Task uses the error reports of all working processors (i.e., processors which have not been retired) to determine which processors should be configured out of the system. Figure 1 (b) shows a second data structure, WORKING,

an 8-element logical array which indicates which processors in the SIFT computer are currently configured in the system. Each processor stores the error reports transmitted from all other processors in an array called "errpt" which is depicted in Figure 1 (c). The output of the fault isolation task is a bitmap showing which processors are to be retired and is called "reconf" as shown in Figure 1 (d). If the error reports from at least two processors indicate that a third is faulty for two consecutive frames, then the processor is marked for retirement. The following is an informal specification of this process taken from the description of the operating system [PALU85]:

For a processor to be considered faulty, at least two other processors must report the subject processor failed. All processors, both working and not working, are tested. The error report is a bitmap of processors where a marked bit indicates the processor as failed. The Fault Isolation Task uses WORKING (an 8-element boolean array indicating which processors are working in the current configuration) and ERRPT (a bitmap indicating which processors have disagreed excessively with the majority vote) to construct the new configuration, RECONF. RECONF is a bitmap of processors where a marked bit also indicates a failed processor. For all processors then, all working processor error reports are tested, excluding the subject processor.

This specification was implemented using a fault tolerant software structure known as the recovery block [RAND75]. A recovery block consists of a primary routine, an acceptance test which is a real-time check on the output of the primary routine, and an alternate routine which is executed if the acceptance test detects a primary routine failure. The following two subsections document development of the ECT for the primary routine and the acceptance test.

Primary Routine Verification

Table 1 is the condition table generated from the above specification and represents the first step in the development of an ECT. The possible rules resulting from this table include two fully specified rules (1: processor is working, not examined by itself, and accused by more than one other processor; and processor is working, processor is not examined by itself, and 2: processor is not accused by more than one other processor), and two rules which have DON'T CARE conditions in them (3: if the processor is working and is examining itself, the error report will be disregarded; and 4: if the processor is not working, the error report will be disregarded).

Figure 2 is the Pascal code for this fault isolation task which meets the specification. The RECONF word is initialized to 0 and a temporary variable, BITEST is set to 1. The loop in lines 3 through 13 checks the error reports from all processors. Lines 7 and 8 implement the requirement that a processor (1) can not accuse itself and (2) that the error report from a non-working processor will be disregarded. Line 9 is the actual point at which the presence of a 1 or 0 in the error report from an individual processor is checked. The BITEST variable takes the values 1, 2, 4, ... 128 (i.e., consists of all 0's except for a 1 at a particular location -- see line 12); the AND operation is used so each bit of each error report is checked individually. Line 10 counts the number of times that a processor is marked as faulty (by processors meeting the conditions of lines 7 and 8), and line 11 sets the RECONF word if two or more processors have marked a third as being faulty.

The comments on lines 7, 8, 9, and 11 correspond to the conditions shown in Table 2 (the second step in the ECT development). A comparison with Table 1 shows that an additional condition has been added (c3) because a decision point had been introduced in line 9 of the code, and two additional rules (4 and 5) have been added as a result. These additions demonstrates the importance of developing the table based on the code. However, it is important to develop the first condition table based on the specification in order to aid in verifying that the code is correctly implemented.

The first two steps in the development of the ECT considered only structural aspects of the code. However, in the third step, the functional perspective is integrated into the condition table by means of fault trees. For the purposes of software verification, the top event of the fault tree is any functional failure of the module being analyzed. Failures are further decomposed until specific functional failures are identified. Figure 3 shows the fault tree for the verification of the fault isolation task. The top event is failure of the fault isolation task to correctly identify disagreeing processors. At the second level are the two subdivisions of this failure: failure to detect a failed processor and false accusation of a working processor. The development of the left branch is not of interest in this example (such a determination can be made only in retrospect) and is therefore not shown. The right branch can be decomposed into two primal events: spurious retirement of a working processor because (1) an error report from a non-working processor is counted or (2) because a self accusation is counted. The right branches are of interest because they result in an enhancement of the condition table as shown in Table 3. For triggering the primal event entitled "Self-accusation counted", c2 of rule 5 has been set to "n" from DON'T CARE. In order to stimulate the primal event entitled "Error Report from Disabled Processor Counted", c2 and c3 of rule 6 have been set to "y" from DON'T CARE's to represent the case that the previously failed processor accuses another processor of having failed. An additional condition, c5 is added because of the concern in the fault tree that a failed processor together with a working processor can mark a third processor as having failed. This concern is represented as 'Count = 1' because of the structure of the code.

The need for this enhancement to the standard condition table can be demonstrated by comparing test case data generated from a conventional condition table (Table 2) and the enhanced condition table (Table 3) against a seeded error shown in Figure 4, line 10.2. Because this seeded error does not affect the structure of the code, the condition tables of the defective code would be identical to Tables 2 o 3. In a conventional condition table, a DON'T CARE condition such as rule 5, condition c3 (the condition which sets errpt[pj] and bitest > 0) in Table 3 could be arbitrarily set to "n". The test data generated from this table would not detect the error self-accusation. However, the fault will be revealed through the test case generated by the enhanced table because of the specification of the self-accusation case shown in Table 3.

The fourth and final step of the ECT development process is consideration of loops. The need for inclusion of rule sequence testing can be demonstrated by consideration of an initialization fault such as that shown in Figure 5. Test data generated from Table 3 alone will not necessarily reveal the fault because the appropriate input data may not be used.

The conventional condition table does not provide a convenient format for specifying test cases for loops, and they have posed a significant problem for path testing because of the unmanageably large number of paths. The approach taken in the ECT methodology is to establish classes of rules based on significant common characteristics, define sequences of execution of these classes, and then devise test cases that exercise the rules in the defined sequences through the loops.

For the case of the SIFT Fault Isolation Task, the six rules in 3 could be divided into two categories:

A. A processor is accused by two or more other processors of being faulty and is therefore to be retired (rules 1 and 3)

B. A processor is not to be retired (rules 2, 4, 5, and 6)

Table 4 is a rule sequence table that supplements the ECT shown in 3. The rationale in establishing these sequences was that within a loop, each category should be executed alone, together with the other class, and then alternating with the second class.

Once the ECT has been completed, test cases are defined so that all rules and rule sequences will be exercised. Figure 6 (a) shows the input values for the working array and 8 input error reports that will exercise all rules in 3. While it is not always possible to generate a test case which is so efficient, it is frequently possible for a single test case to exercise multiple rules in the condition table. Figure 6 (b) defines additional test cases for rule sequence testing (under some circumstances, test cases generated from the upper part of the table will meet the rule sequence requirements). The guideline that was used in developing test cases for these special values was that at least one rule must be exercised among the rules having the same precedence level for each row in the rule sequence table. However, it is desirable to test as many different rules as possible for each of these sequences.

Acceptance Test Verification

Figure 7 is the acceptance test for the SIFT Fault Isolation Task. Lines 9 to 14 and lines 16 to 19 form two structurally independent portions. Although the development of the ECT for the Fault Isolation Task acceptance test follows a similar process to the ECT for the primary routine, the presence of two structurally independent portions allowed partitioning with a significant reduction in the size of the test case set as described below.

The two resulting ECTs are shown in tables 5 and 6 respectively. The additional conditions c4 and c5 in table 5 were added because of the concerns in the Fault Isolation Task fault tree (Figure 3). The fault tree indicates that a failed processor may not be identified, and that an invalid accusation together with a valid accusation can mark a processor as having failed.

The benefit of the partitioning of this fault tolerance provisions is the small number of test cases that are present in a complete test set case for the two ECTs (an ECT test case set can be considered complete if all the rules and the all the rule execution sequences are exercised) shown in Table 7. The separation of fault tolerance provisions testing from the underlying code avoids a combinatorial explosion. Rules are selectively expanded to keep the input space to a reasonable size.

Discussion

The general goal of software testing is to affirm the quality of a program through systematic exercising of the code in a carefully controlled environment. The key activity in test-based verification is the generation of sets of test cases. The figures of merit for such test case sets are effectiveness, the ability of the test cases to reveal coding faults, and efficiency, the extent to which the number of test cases has been minimized to

provide a given level of confidence that operationally significant software defects have been revealed.

Test case sets generated using the ECT more effective and efficient than other testing methodologies because:

1. Explicit Consideration of Multiple Conditions on Branches: Branch or path coverage measures are based on a single pass through the branch or the combination of branches (i.e., path). These measures do not consider between different conditions that cause the same branch or path to be traversed, and may result in the selection of input data over which a path computes fortuitously correct output [HOWD78b]. The ECT method lists conditions individually and enumerates all possible and necessary combinations. Because the ECT generates the test cases for all possible way to terminate a loop or to enter a branch, a defect of a particular condition on a multiple condition branch will be detected.

2. Resolution of DON'T CARE conditions: As noted by Goodenough and Gerhart, greater test case effectiveness is achieved if all the DON'T CAREed outcomes of conditions are expanded out [GOOD75]. Expanding out all these rules for any realistically sized condition table may result in an unmanageably large number of test cases, and hence, they are usually ignored. Use of fault trees allows the replacement of DON'T CARE entries in the condition table with meaningful values. As decision multiply, the size of a condition table increases geometrically if all the DON'T CAREs are expanded out. Thus, this saving will become significant. With the guidance from fault tree analysis, a test data set derived from ECT can be of a reasonable size and still effective.

3. Treatment of Loops: In general, a program with loops will have an enormous or infinite number of paths -- even if the number of iterations of loops is small number. The ECT method does not require all permutations of rule traversal through a loop on coverage measure in order to avoid the combinatorial-explosion problem. The ECT method considers only special functionally important, and/or error-prone rule traversal patterns. A carefully chosen subset of the large number of rule traversal patterns forces the manifestation of associated error, and greatly increase our confidence that a program works as specified.

4. Representation of Program Logic: A typical attribute of fault tolerant software that makes testing more difficult is the inherent logical complexity. An important representative of the complexity is the large number of decision statements [DEMI87]. Most software errors and testing deficiencies result from failing to see or deal correctly with all predicates in decision statements. The ECT focuses on uncovering these predicates and their combinations and is therefore well suited to fault tolerant software verification.

Although there are no inherent properties of the ECT that preclude its use in conventional software verification, it is particularly amenable to fault tolerant software because

1. The ECT explicitly considers failure modes and data dependent faults in contrast to most other structural and functional testing methods.

2. In general, the fault detection and recovery provisions in fault tolerant software are functionally simpler and smaller in size than the operational

code. Hence, even when applications cannot be tested exhaustively, the ECT can still be used to provide high confidence verification for the fault tolerance provisions. In addition, most of these critical components are partially reusable (e.g., routine switching procedure of the recovery block or the supervisory system in an N-version program).

Because generation of ECTs is a skill and labor intensive process, it must be judiciously applied. Each fault tolerant software system must have a hard core that continues to function in the presence of other faults. Examples are (1) the acceptance test and the routine switching procedure in the recovery block, and (2) the supervisory system including the decision function, (the function that establishes the majority value), the synchronization procedure, and the multi-version driver routine in n-version programming. Failures of these elements not only reduce the capability of fault tolerance of the system but also possibly reject some non-faulty components and accept faulty results. For example, an incorrect acceptance test might reject a non-faulty routine and accept a faulty routine in the recovery block; an erroneous voter in a N-version system might favor a wrong result from a group of erroneous versions which is not a majority. Therefore, an intensive testing program based on the ECT should be applied to the verification of these critical components in fault tolerant software systems even if they can not be applied to the entire system.

Acknowledgements

Portions of this work were performed under USAF Rome Air Development Center contract F30602-85-C-0229, "Fault Tolerant Software Technology". The SIFT code and documentation were obtained from the NASA Langley Research Center. The authors wish to acknowledge the interest and support of Mr. R. Slavinski (Rome Air Development Center) and of Messrs. D. Palumbo, and M. Holt (NASA Langley Research Center).

References

[DEMI87] R. DeMillo, et al, "Software Testing and Evaluation," The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[GOOD75] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp.156-173.

[HOWD78] W.E. Howden, "Functional Programming Testing," Technical Report, Dept. of Mathematics, University of Victoria, Victoria, B.C., Canada, DM-146-IR, August, 1978.

[HOWD82] W.E. Howden, "Validation of Scientific Programs," ACM Computing Surveys, Vol. 14, No. 2, June 1982, pp.193-227.

[LISK86] B. Liskov and J. Guttag, "Abstraction and Specification in Program Development," The MIT Electrical Engineering and Computer Science Series, McGraw-Hill Book Co., New York, 1986.

[MILL81] E.F. Miller, Jr., et. al., "Application of Structural Quality Standards to Software", Software Engineering Standards Applications Workshop, IEEE Catalog No. 81CH1663-7, pp. 51-57, July, 1981

[PALU85] D. Palumbo, The SIFT Hardware/Software Systems -- Volume I, A Detailed Description, NASA Technical Memorandum 87574, NASA Langley Research Center, September, 1985

[RAND75] B. Randell, "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, no. 2, pp. 220-232, June, 1975

[SHOO83] M.L. Shooman, "Program Testing," Software Engineering, McGraw-Hill, Inc., 1983, Singapore, pp.223-295.

[WENS73] J. Wensley, et. al., "Design of a Fault-Tolerant Airborne Digital Computer. Volume I - Architecture", NASA CR-132252, NASA Langley Research Center, 1973

Table 4. Rule Sequence Table for Table 3

Sequence Number Output	Description	RECONF
1	Exercise only rule category A (rules 1 and 3)	11111111
2	Exercise only rule category B (rules 2, 4, 5, and 6)	00000000
3	Exercise rule category A, then rule category B	At least one "1" followed by at least one "0"
4	Exercise rule category B, then rule category A	At least one "0" followed by at least one "1"
5	Alternate between rule category A, then rule category B, then back to category A	At least one "1" followed by at least one "0" followed by at least one "1"
6	Alternate between rule category B, then rule category A, then back to category B	At least one "0" followed by at least one "1" followed by at least one "0"

(b) Rule Sequence Table

Sequence Number	Description	RECONF Output
1	Exercise only rule category A (rules 1 and 3)	11111111
2	Exercise only rule category B (rules 2, 4, 5, and 6)	00000000
3	Exercise rule category A, then rule category B	At least one "1" followed by at least one "0"
4	Exercise rule category B, then rule category A	At least one "0" followed by at least one "1"
5	Alternate between rule category A, then rule category B, then back to category A	At least one "1" followed by at least one "0" followed by at least one "1"
6	Alternate between rule category B, then rule category A, then back to category B	At least one "0" followed by at least one "1" followed by at least one "0"

(b) Rule Sequence Table

Sequence Number	Description	RECONF Output
1	Exercise rule category A, then rule category B	At least one "1" followed by at least one "0"
2	Exercise rule category B, then rule category A	At least one "0" followed by at least one "1"
3	Alternate between rule category A, then rule category B, then back to category A	At least one "1" followed by at least one "0" followed by at least one "1"
4	Alternate between rule category B, then rule category A, then back to category B	At least one "0" followed by at least one "1" followed by at least one "0"