Fault-Tolerant Software

Herbert Hecht, Senior Member IEEE
SoHaR Inc., Los Angeles

Key Words–Software reliability, System reliability, Redundant software

Reader Aids–
Purpose: Widen state-of-the-art
Special math needed: None
Results useful to: Software engineers

Abstract–Limitations in the current capabilities for verifying programs by formal proof or by exhaustive testing have led to the investigation of fault-tolerance techniques for applications where the consequence of failure is particularly severe. Two current approaches, *N*-version programming and the recovery block, are described. A critical feature in the latter is the acceptance test, and a number of useful techniques for constructing these are presented. A system reliabiliy model for the recovery block is introduced, and conclusions derived from this model that affect the design of fault-tolerant software are discussed.

1. INTRODUCTION

The fault-tolerance features for software described here are primarily aimed at overcoming the effects of errors in software design and coding. In fortuitous circumstances they may also circumvent failures due to hardware design deficiencies or malfunctions in input channels. Fault-tolerance for random computer failures is outside the scope of the present discussion. In practice, the user wants computer system fault-tolerance, i.e. toleration of failures due to all causes, and this can be provided by a combination of established hardware fault-tolerance techniques [1] and the fault-tolerant software described here.

It is generally recognized that even very carefully designed and manufactured computer components may fail, and hardware redundancy is therefore provided in applications where interruptions of service can not be tolerated. That software may fail is also widely recognized; yet this seems to have been perceived as a temporary shortcoming: today's software contains design and coding errors but it is expected that these will be eliminated once improved development and test methodologies or efforts at formal verification become fully effective. Deliberate use of redundant software to tolerate software faults is not an established practice today.

It is expected that the many efforts to improve software quality and reliability will indeed reduce failures but will not completely eliminate them. A number of thoughtful articles have pointed out the limitations of current test methodology [2] and of formal verification [3 - 5]. Sometimes it is believed that maturity can provide freedom from software errors but that is not borne out by the experience on extensively used operating systems [6]. For the foreseeable future even the most carefully developed software will contain some faults, probably of a subtle nature such that they were not apparent during software and system testing. These faults will manifest themselves during some unusual data or machine state and will lead to a system failure unless fault-tolerance provisions are incorporated in the software. In the Bell Laboratories' Electronic Switching Systems (which employ hardware redundancy and thoroughly tested software such) software faults accounted for approximately 20% of all failures [7].

Fault-tolerance always involves some redundancy and therefore increases the resource expenditure for a given function. Software fault-tolerance of the type described here is therefore primarily aimed at applications where the consequences of failure are particularly severe, e.g. advanced aircraft flight control systems [8, 9], air traffic control programs, and nuclear reactor safety systems. Sometimes fault-tolerance applied to a small segment of a large software system can provide a hardened kernel that can then be used to organize the recovery from failures in other segments [10]. Such applications are today under study or in early development. Experience with these techniques in an operational environment has not yet been reported.

A survey of current approaches to software fault-tolerance is presented in the next section. This is followed by a discussion of a particularly critical component, the acceptance test that determines when the primary software routine has failed. In the final section a system reliability model for fault-tolerant software is described.

2. CURRENT APPROACHES

Two different techniques for achieving fault-tolerance in software have been discussed in the recent literature: the recovery block and *N*-version programming. In the latter a number ($N \geq 2$) of independently coded programs for a given function are run simultaneously (or nearly so) on loosely coupled computers, the results are compared, and in case of disagreement a preferred result is identified by majority vote (for $N > 2$) or a predetermined strategy. This approach had been suggested in a general way by Elmendorf [11] and has more recently been developed into a practical form by Avizienis & Chen [12, 13] who report results on the use of this technique on a classroom problem.

The success of this technique is obviously governed by the degree of independence that can be achieved in the *N*-versions of the program. Ref. [13] states "Wherever possible, different algorithms and programming languages or translators are used in each effort." One might also want to see different data

0018-9529/79/0800-227 $00.75

structures so that the requirements document rather than the specification (which usually defines data structures quite rigorously) becomes the common starting point for the N versions. In addition, the voting algorithm and the housekeeping for 'results' prior to and after voting have been identified as critical items for this technique.

A specific constraint on N-version programming is the requirement for N computers that are hardware independent yet able to communicate very efficiently so that rapid comparisons of results can be achieved. These N computers must all be operating at the same time, and a hardware failure in any one of them will at best force the system into a different operating mode and may, in minimal configurations, cause loss of the fault-tolerance provisions. An example of a system that seems well suited to host N-version programming is SIFT [9]. N-version programming is capable of masking intermittent hardware faults, and this can be an advantage in some applications. It also has the ability to aid in detection of permanent hardware faults, although detail fault-tolerance provisions for these may best be handled by dedicated hardware/software reconfiguration provisions.

The recovery block technique [14, 15] can be applied to a more general spectrum of computer configurations, including a single computer (which may also include hardware fault-tolerance). The simplest structure of the recovery block is:

Ensure T

By P

Else by Q

Else Error

where T is the acceptance test condition that is expected to be met by successful execution of either the primary routine P or the alternate routine Q. The internal control structure of the recovery block will transfer to Q when the test conditions are not met by executing P. Techniques have been described for purging data altered during processing by P when Q is called [15].

For real-time applications it is necessary that the execution of a program be both correct and on time. For this reason the acceptance test is augmented by a watchdog timer that monitors that an acceptable result is furnished within a specified period. The timer can be implemented in either hardware or software or a combination. The structure of a recovery block for real time application modules is shown in fig. 1 [16]. In normal operation only the left part of the figure is traversed. When the acceptance test fails, or if the time expires, a transfer to the alternate call is initiated, a flag is set, and process Q is executed. If its result satisfies the acceptance test, the normal return exit in the right part of the figure is taken, and processing continues. If the acceptance test fails again, or if a time-out is encountered in the execution of Q (with the flag now set), an error return results.

A fault-tolerant navigation module using these techniques

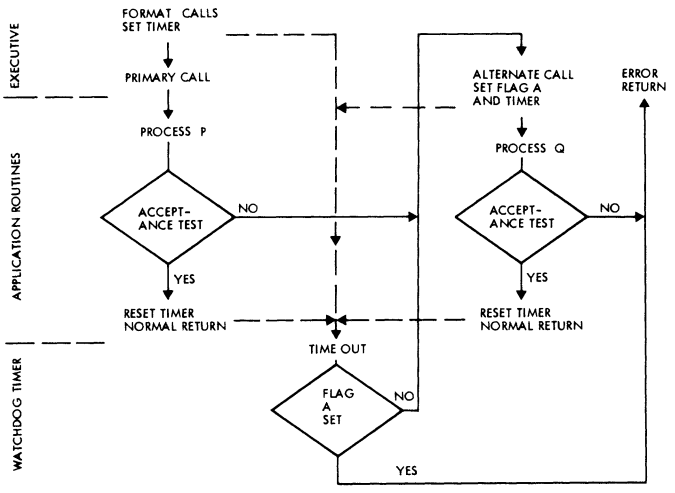

Fig. 1. Recovery Block for Application Modules

is described in [16], and the interaction of fault-tolerant application modules with the executive is also discussed there. Principles of a fault-tolerant scheduler based on the recovery block technique have also been described [17]. The number of alternate routines is not restricted, and where it seems desirable any number of back-ups can be entered successively on failure of the acceptance test. Recovery blocks can also be used in concurrent processes and multi-level structures [18, 19].

As in N-version programming, it is desirable that the redundant routines in a recovery block be as independent of one another as possible. A specific and critical feature of the recovery block is the acceptance test. Alternate routines will be useless if failure of the primary one is not promptly detected. Thus acceptance tests must be thorough. On the other hand, the acceptance test is traversed on every program execution, and the amount of code required for it should therefore be minimized. Few formal guidelines exist for satisfying these partly contradictory requirements. The following section attempts to classify techniques that have been used as a first step towards a systematic study of the design of acceptance tests.

3. ACCEPTANCE TESTS

Acceptance tests can be devised against two criteria: to detect deviations from expected program execution, or to prevent unsafe output. The first is more restrictive, and will result in more frequent transfer to the alternate routine. However, the penalties for unnecessary transfer are usually small, whereas the penalties for failure to switch when necessary can be much greater. For software that has been in use a long time, testing for unsafe output may be preferable because it can be simpler and it avoids unnecessary transfers. However, for programs just emerging from development, testing for expected program execution has 3 benefits:

1) Unexpected behavior of the primary system will be noted even in cases where only a mild degradation is encountered. This aids in program evaluation.

2) Switching to the alternate program is exercised more often under realistic (unplanned) conditions. Providing realistic testing of the fault-tolerance mechanism is a difficult undertaking.

3) As a program matures it is usually easier to relax acceptance conditions than to make them more restrictive.

The four types of acceptance test described below can usually be designed to test either for expected execution or for unsafe output.

1. *Satisfaction of requirements*

In many cases the problem statement imposes conditions which must be met at the completion of program execution. These conditions can be used to construct the acceptance test.

In the 'Eight Queens' problem it is required that eight queens be located on a chessboard such that no two queens threaten each other. A suitable acceptance test for a computer program solving this problem is that the horizontal, vertical, and the two diagonals identified with the location of a given queen do not contain the location of any other queen. If testing for these conditions is already included in one or more of the routines in a recovery block, then the acceptance test should use a different sequence and a different program structure.

The acceptance test for a sort problem described by Randell is also a test for satisfaction of requirements [15]. The test involves checking at the completion of the execution that the elements are in uniformly descending order, and that the number of elements in the sorted set is equal to the number of elements of the original set. This test is not exhaustive: changes in an element during execution would not be detected. A stronger test, to determine that the elements of the sorted set are a permutation of the original set, was rejected because of excessive programming complexity and execution time.

An important subset in this class is the inversion of mathematical operations, particularly those for which the inverse is simpler than the forward operation. A typical example is the square root which is frequently handled as a subroutine call whereas squaring a number is a one-line statement. The effectiveness of inversion for the construction of acceptance tests is limited by the fact that some logical and algebraic operations do not yield a unique inverse, e.g. OR, AND, absolute value and trigonometric operations.

Testing for satisfaction of requirements is usually most effective when carried out on small segments of a computer program because at this level requirements can be stated simply. On the other hand, for efficiency of the overall fault-tolerant software system, it is desirable to construct acceptance tests that cover large program segments. The classes of acceptance tests described in the following two sections have better capabilities in this regard but are more limited in the types of programs which they can handle. For text editing systems, compilers, and similar programs, tests for satisfaction of requirements constitute at present the most promising approach.

2. *Accounting Checks*

Commercial accounting had to struggle with the problem of maintaining accuracy in systems with many records and arithmetic operations long before the advent of the digital computer. Most of the procedures that had evolved for checking manual operations were taken over when bookkeeping evolved into data processing. These accounting checks can be very useful for acceptance tests in software that serves transaction oriented applications. Airline reservation systems, library records, and the dispensing of dangerous drugs all can be checked by these procedures.

The most rudimentary accounting check is the tally which in computer usage has become the checksum. Whenever a volume of financial records (checks, invoices, etc.) is transmitted among processing stations, it is customary to append a tally slip representing the total amount in the records. On receipt, a new total is computed and compared with the tally. The corresponding use of checksums is widespread in data processing. The digital presentation of information makes it possible to apply the checksum to non-numerical information as well.

When a large volume of records representing individual transactions is aggregated, it is almost impossible to avoid errors due to incorrect transcriptions or due to lost or misrouted documents. In the commercial environment such errors are not always of an innocent nature since an employee may be able to pocket the amount corresponding to an improper entry. The double-entry bookkeeping system evolved as an effective means of detecting such errors. In this procedure the total credits for all accounts must equal the total debits for all accounts for any arbitrary time period, provided only that corresponding transactions have been entered into all accounts. This equality can also be used as a criterion in acceptance tests.

Another worthwhile accounting check is the reconciliation of authorized transactions with changes in physical inventory over a period of time. For example, in storage of nuclear material it is possible to determine the quantity in inventory by means of radiation counters which can feed data directly into a computer. At specified intervals the change in radiation level can be compared to that independently calculated for normal decay of the material and authorized inventory transactions. This furnishes an overall check, including most computer and software errors.

Accounting checks are suitable only for transaction-oriented applications and they cover only elementary mathematical operations. Within this sphere, accounting checks provide a time-tested and demonstrably complete means for assessing the correctness of computer operations. They can test recovery blocks containing large software segments, and portions of the input operations and physical inventory can be checked in the same process.

3. *Reasonableness Tests*

This heading includes acceptance tests based on precomputed ranges of variables, on expected sequences of program states, or on other relationships that are expected to prevail for the con-

trolled system. The dividing line between reasonableness tests and testing for satisfaction of requirements can become blurred, but in general reasonableness tests are based on physical constraints whereas testing for requirements uses primarily logical or mathematical relationships.

Reasonableness tests for numerical variables can examine the individual values (e.g. to be within range), increments in individual values of the same variable (increments between successive values or deviations from a moving average), or the correlation between values of different variables or of their increments. Examples of these different types are examined for an airspeed calculation. The indicated airspeed (typically an input quantity), and the true airspeed (a computed quantity) must each be within a range that is dictated by the aerodynamic and structural capabilities of the airframe, e.g. 140 to 1100 km/hr. Obviously only gross malfunctions of either the sensor or of the computing process can be diagnosed by such a test. The airspeed range is a function of aircraft configuration (flap position, etc.) and an acceptance test with narrower limits can be constructed if adjustments for configuration are included.

A much more sensitive test for sudden malfunctions (and these are usually the most critical ones) can be devised by examining the increments in each quantity. Changes in speed are equivalent to acceleration, and in the normal flight mode changes in forward speed are well below the 1 g level. Even if the acceptance test is based on the maximum allowable acceleration for structural integrity (which may be 6 g), the corresponding change in speed is limited to 213 km/hr/sec. Individual speed calculations can be carried out ten times per second, yielding an allowable speed increment between successive values of 21.3 km/hr. To suppress noise in the sensor output, the acceptance test may operate on averaged readings so that the effective sampling interval is increased, but even under these circumstances the acceptance test based on increments will for sudden malfunctions be much more sensitive than one based on range.

The correlation between increments of indicated and true airspeed can be used for further refinement, but an even more useful correlation can be obtained by comparing increments in true airspeed with the acceleration measured by an appropriately oriented accelerometer. In this case, limits of the acceptance test will depend on the noise characteristics of the instruments used, elastic deformations of the aircraft, and other secondary characteristics. For filtered observations the acceptance region can probably be reduced by an order of magnitude over that obtained in the test based on increments alone.

The use of correlated measurements for acceptance tests always raises the problem that errors might be introduced by the variable added to provide the refinement (in this case the accelerometer output). The consequences of a spurious failure of the acceptance test must be evaluated to determine whether the refinement is indeed warranted. To be particularly avoided is the use of a variable in the acceptance test that is also used in the back-up routine because this could cause transfer to the back-up at exactly the time when its data source is unreliable. The importance of keeping the back-up program independent of software structures and data used in the acceptance test is discussed further in connection with reliability modeling.

An example of a reasonableness test based on state transition is found in an electronic telephone switching system. Once a call has proceeded to the 'connected' state it is inadmissible for it to subsequently go to 'ringing' or 'busy'. Inappropriate transitions of this type can therefore be used to signal the need for switching to an alternate routine.

Tests for reasonableness of numerical or state variables are a very flexible and effective way of constructing acceptance tests for fault-tolerant software. They permit acceptance criteria to be modified as a program matures. Reasonableness tests can be devised for most real-time programs that control physical variables, and they may monitor overall performance of a computing system, e.g. by reasonableness tests on output variables.

4. *Computer Run Time Checks*

Most current computers provide continuous hardware-implemented testing for anomalous states such as divide by zero, overflow and underflow, attempts to execute undefined operation codes, or writing into write-protected memory areas. If such a condition is detected, a bit in a status register is set, and subsequent action can then be defined by the user. When fault-tolerant software is being executed, encountering one of these conditions can be equated with failure of the acceptance test and transfer to an alternate software routine is then effected. The previously mentioned watchdog timer can be tied into this status reporting scheme.

Run-time checks can also incorporate data structure and procedure oriented tests that are embedded in special support software or in the operating system. Checking that array subscripts are within range is already implemented in many current computer systems. Array value checking (for being within a given range, being in ascending or descending order, etc.) has also been proposed [20]. Under the title "Self-checking Software" and "Error-Resistant Software" a number of interesting run-time monitoring techniques have been described, many of which are akin to acceptance tests mentioned earlier in this section [21, 22]. A particularly appropriate concept for a run-time acceptance test is an Interaction Supervisor [22]. In its simplest form this requires declaration for each module of authorized callers and authorized calls. The Interaction Supervisor will cause failure of the acceptance test if access to, or exit from, a module involves unauthorized locations.

The value of run-time checks is not restricted to prevention of failures due to errors arising directly from the attribute that is being monitored. They cover a much wider area, e.g. attempts to write into write-protected memory may have as their original cause an improper indexing algorithm, a failure to clear a register, or similar more subtle software discrepancies. It is therefore appropriate to use all of these facilities that modern computers, operating systems, and programming languages can contribute to implementing acceptance tests even if the occurrence of the monitored conditions per se could be prevented by other means. Run-time checks are not exhaustive but they require very little development time or other resources. They supplement the previously mentioned types of acceptance tests for critical segments, and they can be used by themselves as

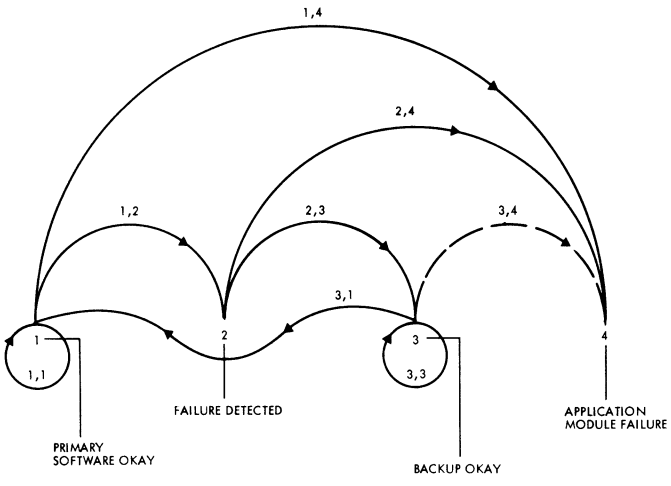

Fig 2. Transition Model

the acceptance test for non-critical segments operating as part of an overall fault-tolerant software system.

Ultimately one would like to see a classification of acceptance tests that characterize them by error detection capability, run-time penalty, and storage requirements, thus permitting a rational selection for each application. This stage, unfortunately, has not yet been reached. But that need not deter advancing with practical applications; there is little methodology for the routine testing of software which is being carried out all the time, and occasionally even with satisfactory results.

4. SYSTEM RELIABILITY MODELING

A potential advantage of the fault-tolerant software approach over other techniques for software reliability improvement is that it permits using (with some modifications) system reliability modeling techniques that have been developed in the hardware field. The qualifying 'potential' in the previous sentence reflects the fact that adequate parameters for the models do not yet exist although a methodology for obtaining them has been defined [23, 24]. Interesting insights can be achieved by modeling even at the present stage.

This will be demonstrated with a transition model shown in fig. 2 for a recovery block consisting of a primary and an alternate routine [16]. Reliability models can also be developed for *N*-version programming and for recovery blocks having multiple alternates.

Along the bottom of the figure are four possible states for the recovery block (here identified as an application module). Starting at state 1, primary routine operating, the immediate transitions are:

Loop 1, 1 – Primary routine continues to operate
Arc 1, 2 – Failure in primary routine detected
Arc 1, 4 – Undetected failure in the primary routine

Out of state 2, a transient state after detection of failure, two transitions are possible:

Arc 2, 3 – Transition to a satisfactory alternate
Arc 2, 4 – Failure at transition

State 3 is defined as satisfactory operation of the back-up routine through at least one complete program pass, and once this is achieved, the further possible transitions are:

Loop 3, 3 – Back-up continues to operate
Arc 3, 1 – Reversion to primary routine
Arc 3, 4 – Uncorrelated failure of the back-up

The word 'uncorrelated' is inserted because failures in the back-up routine occasioned by the transition from state 2 are represented by the arc 2, 4 as will be further discussed. The uncorrelated failures (arc 3, 4) are extremely unlikely, given thoroughly tested software and the usually limited time period for operation in the back-up mode. This transition is therefore shown in dashed symbols.

The major sources for failure of the recovery block are expected to be undetected failures of the primary routine and correlated failures in the back-up. The former are caused by deficiencies in the acceptance test. The previous discussion of this subject has shown that design of acceptance tests is far from an established discipline, and the possibility of undetected failures must be accounted for. Correlated failure of the back-up routine (arc 2, 4) can be caused by two circumstances: correlated deficiencies in the primary and back-up routine software, and correlation of faults in the acceptance test with those in the back-up routine. Insistence on independent design and coding of the two software routines, and, wherever possible, use of independent data sources, will minimize the first of these. The second cause of these correlated failures is particularly insidious, since the failure occurs even though the primary software may execute correctly! Possible sources of such correlated failures of the acceptance test and the back-up routine must therefore be thoroughly investigated in the design of a recovery block. The use of common data, common algorithms, and common subroutines should be avoided. Where some commonality is unavoidable (cf. the Eight Queens problem in section 3) at least the detailed software design should be varied.

Use of this transition model, even with the very inadequate data available, has led to some interesting insights into the measures necessary to obtain software reliability consistent with critical flight control applications [17]. A substantial advantage of the recovery block approach illustrated by this model is that the requirement for demonstrated reliability of the primary and alternate routines can be held several orders of magnitude (in terms of failure probability) below that required for the recovery block as a whole.

5. DIRECTIONS FOR THE FUTURE

Complete fault-tolerant software systems can for the foreseeable future be considered only for the most demanding and safety-critical applications, e.g. fly-by-wire passenger aircraft or safety systems for nuclear reactors. But fault-tolerant segments in otherwise conventional software may find much wider use. Such segments may be needed only temporarily, e.g. when a new operating system is being introduced, or they may be permanently installed, e.g. for back-up file manage-

ment in a reservation or inventory control system.

Even more widespread may be the use of fault-tolerant techniques (short of a formal recovery block or *N*-version segment). In many applications it may be sufficient to halt operations (or to flag output) when errors occur. Acceptance tests or the previously referenced techniques for self-checking software will be used here. In other tasks, e.g. in the accounting field, the availability of back-up routines and data caches may be important, while the acceptance test might be relegated to a human observer or to a separately running audit routine.

Research will more and more apply the basic techniques outlined here to multi-tasking and multi-processing environments, and to the multi-layered operating systems of time-shared computers.

All of the present and most of the foreseeable applications are dictated by a need for greater reliability in the computing function without specific economic trade-offs of one technique versus another (the choices are very limited). It is questionable whether at some future time good criteria can be developed on where to apply software fault-tolerance and where to apply intensive validation methods. Where highly reliable illumination is desired we use long-life bulbs, redundant bulbs, and separate emergency lighting systems. Sometime all of these are used together and sometimes they are used separately. Any one of them is better than reliance on a single standard light bulb. Future generations may look in the same way at our efforts to produce more reliable software.

ACKNOWLEDGEMENT

Portions of the work reported here were carried out under subcontracts to C.S. Draper Laboratory and SRI International in connection with efforts by these organizations for the NASA Langley Research Center under contracts NAS1-15336 and NAS1-15428, respectively.

REFERENCES

[1] *Proc. IEEE,* Special Issue on Fault-Tolerant Digital Systems, vol 66, 1978 Oct.

[2] W.E. Howden, "Theoretical and empirical studies of software testing", *IEEE Trans. Software Engineering,* vol SE-4, 1978 Jul, pp 293-297.

[3] S.L. Gerhart, L. Yelowitz, "Observations on the fallibility in applications of modern programming methodologies", *IEEE Trans. Software Engineering,* vol SE-2, 1976 Sep, pp 195-207.

[4] C. Reynolds, R.T. Yeh, "Induction as the basis for program verification", *IEEE Trans. Software Engineering,* vol SE-2, 1976 Dec, pp 244-252.

[5] S.L. Gerhart, "Program verification in the 1980s: Problems, perspectives, and opportunities", ISI/RR-78-71, Information Sciences Institute, Marina del Rey, Calif, 1978 August.

[6] L.A. Belady, M.M. Lehman, "A model of large program development", *IBM Systems Journal,* vol 15, no. 3, 1976, pp 225-252.

[7] W.N. Toy, "Fault tolerant design of local ESS processors", in [1], pp 1126-1145.

[8] A.L. Hopkins, et al., "FTMP – A highly reliable fault-tolerant multiprocessor for aircraft", in [1], pp 1221-1239.

[9] J.H. Wensley, et al., "SIFT: The design and analysis of a fault-tolerant computer for aircraft control", in [1], pp 1240-1254.

[10] K.H. Kim, et al., "Strategies for structured and fault-tolerant design of recovery programs", *Proc. COMPSAC '78,* 1978 Nov, pp 651-656.

[11] W.R. Elmendorf, "Fault-tolerant programming", *Digest of the 1972 International Symposium on Fault-Tolerant Computing,* pp 79-83.

[12] A. Avizienis, L. Chen, "On the implementation of *N*-version programming for software fault-tolerance during execution", *Proc. COMPSAC '77,* 1977 Nov, pp 149-155.

[13] L. Chen, A. Avizienis, "*N*-Version programming: A fault-tolerance approach to reliability of software operation", *Digest of Papers, FTC – 8,* 1978 Jun, pp 3-9.

[14] J.J. Horning, et al., "A program structure for error detection and recovery", *Proc. Conf. Operating Systems: Theoretical and Practical Aspects, IRIA,* 1974 Apr, pp 174-193.

[15] B. Randell, "System structure for software fault-tolerance", *IEEE Trans. Software Engineering,* vol SE-1, 1975 Jun, pp 220-232.

[16] H. Hecht, "Fault-tolerant software for real-time applications", *ACM Computing Surveys,* vol 8, 1976 Dec, pp 391-407.

[17] Advanced Programs Division, The Aerospace Corporation, "Fault-tolerant software study", NASA CR 145298, 1978 Feb.

[18] J.S.M. Verhofstad, "The construction of recoverable multi-level systems", PhD Dissertation, University of Newcastle upon Tyne, 1977 Aug.

[19] K.H. Kim, C.V. Ramamoorthy, "Failure-tolerant parallel programming and its supporting system architectures", *AFIPS – Conf. Proc.,* vol 45 (NCC 1976) pp 413-423.

[20] L.G. Stucki, G.L. Foshee, "New assertion concepts for self-metric software validation", *Proc. 1975 International Conf. Reliable Software,* IEEE Cat. 75CH0940-7CSR, 1975 Apr, pp 59-71.

[21] S.S. Yau, R.C. Cheung, "Design of self-checking software", same source as [20], pp 450-457.

[22] S.S. Yau, R.C. Cheung, D.C. Cochrane, "An approach to error-resistant software design", *Proc. Second International Conf. Software Engineering,* IEEE Cat. 76CH1125-4C, 1976 Oct, pp 429-436.

[23] J.D. Musa, "Measuring software reliability", *ORSA/TIMS Journal,* 1977 May, pp 1-25.

[24] H. Hecht, et al., "Reliability measurement during software development", NASA CR-145205, 1977 Sep.

AUTHOR

Herbert Hecht; SoHaR, Inc.; 1040 S. LaJolla Ave.; Los Angeles, CA 90035 USA.

Dr. Hecht (M'47,SM'54) is president of SoHaR Incorporated, an organization engaged in studies and consulting in computer software and hardware reliability problems. In prior employment he was Director of Computer Technology in the Advanced Programs Division of The Aerospace Corporation and Department Head for Helicopter and Light Aircraft Flight Controls at the Flight Systems Division of the Sperry Rand Corporation. Hecht received a Bachelor of Electrical Engineering degree from City College, New York and a Master's degree in the same subject from Brooklyn Polytechnic Institute. He obtained a PhD in Engineering from UCLA. He has published many papers on computer and control system reliability subjects and has conducted a short course in Software Reliability at UCLA. He is registered as a Professional Engineer (Control Systems) in California. He is Vice Chair'n of the IEEE Computer Society Technical Committee on Software Engineering and has initiated efforts to standardize software terminology and practices.

Manuscript SI79-08 received 1978 December 1; revised 1979 January 18.

☆☆☆