AN EVALUATION OF SOFTWARE FAULT TOLERANCE TECHNIQUES IN REAL-TIME SAFETY-CRITICAL APPLICATIONS

Nancy G. Leveson

Information and Computer Science
University of California, Irvine

Shaula Yemini

IBM T.J. Watson Research Lab
Yorktown Heights, New York

Technical Report 192

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

November 1982

AN EVALUATION OF SOFTWARE FAULT TOLERANCE TECHNIQUES IN REAL-TIME SAFETY-CRITICAL APPLICATIONS

Nancy G. Leveson

Information and Computer Science
University of California, Irvine
Irvine, California 92717

Shaula Yemini

IBM T.J. Watson Research Lab
Yorktown Heights, New York 10598

Abstract

The usefulness of three software fault tolerance techniques -- n-version programming, recovery blocks, and exception handling is examined within the context of real-time safety-critical environments. The general requirements of such application systems are presented and the techniques evaluated with regard to how well they satisfy these requirements.

Introduction

In real-time environments requiring continuous service, e.g. air traffic control systems, hospital patient monitoring systems, public transportation systems, nuclear power plants, and space craft, the "cost" of a failure is high and may involve loss of life. In this case, the use of fault-tolerance techniques becomes critical.

Several approaches have been suggested for software fault tolerance. These include recovery blocks [12], n-version (or design diversity) programming [2,5], and exception handling techniques [8,11,12,16,22,23]. This paper presents a framework for evaluating software fault tolerance techniques with respect to real-time, safety-critical applications. The framework is then used to evaluate the usefulness of n-version programming, recovery blocks, and programmed exception handling for this type of application.

Basic Requirements

In order to evaluate any programming technique, it is necessary to define the requirements of the environment in which it is to be used. Only then can the technique be evaluated as to how well it satisfies these requirements.

Relevant requirements of real-time, safety-critical systems include the following:

Time Constraints: Since we are assuming real-time systems, responses must be received within a limited period to be useful. Usually this required response time is relatively short. For example, aircraft which are ten miles apart and flying at 600 mph could collide in just 30 seconds. One design criterion for the present U.S. air traffic control system, called the National Airspace System (NAS), is that computer response time should be less than two seconds.

Safety: The application may have safety aspects, that is, run-time failures or errors can result in loss of life or property. There are three aspects of software fault tolerance. When the software function is life-sustaining, fault-tolerance providing full functionality ("fail-operational") behavior is required. In situations where full fault tolerance is not possible or fails itself, "fail-soft" procedures must be employed. A fail-soft system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed. As opposed to the fail-operational and fail-soft systems, the goal of a fail-passive or fail-safe system is to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

Software fault tolerance techniques have primarily concentrated on full fault tolerance or fail-operational behavior. However, in environments where safety is important, fail-soft and fail-safe facilities may be required. Furthermore, there may be a set of acceptable degraded modes and a priority of failures and failure-handling procedures. For example, it may be necessary to ensure containment of any possible radiation leakage before attempting other recovery procedures.

Early Detection of Faults: In most cases, the earlier faults are detected, the better. It makes no sense to try to detect and recover from faults at run-time which could have been removed prior to operational use of the system. Therefore we will assume that verification and validation techniques have been used. This includes verification of correctness [9] and safety [13]. Although it can be argued that formal verification is costly [4] and not foolproof [7], the same can be said for software fault tolerance techniques.

Even at run-time, it is usually preferable to detect faults as early as possible. The more processing that is done after the fault has occurred, the harder it may be to detect, isolate, and identify the fault, and the more likely that information will be lost which is necessary for recovery. Furthermore, in a heavily loaded real-time environment, resources should not be wasted on incorrect processing. Finally, from a system viewpoint, an unsafe state may need to be detected quickly to allow for recovery to a safe state before a catastrophe occurs, e.g. when two aircraft are on a collision course, or to contain any possible damage or dangerous conditions, e.g. radiation leakage.

Flexible Recovery Responses: Once an error has been detected, the techniques must be able to support any reasonable recovery response. Software fault-tolerance techniques which provide only backward recovery are useful under some circumstances, but will not be adequate in many real-time systems. Thus, backward and forward error-handling facilities are both necessary. For example, because it is impossible to roll back a system state which includes time, backward error recovery may just not be feasible. Also, forward error-handling may be preferred when feasible because of time considerations. Consider a sailboat with a computerized navigation system. If the boat is off its intended course, it is clearly more practical to recompute a course from the present location (perhaps using an alternate algorithm) than to require it to return to its starting point and begin again.

Recovery responses must also allow for dynamically changing flow of control. Some components of a system are less critical than others, and thus may be temporarily eliminated or suspended if necessary. Furthermore, the criticality of components may vary during processing and may depend upon run-time environmental conditions. Thus, run-time reconfiguration facilities may be required in order to reconfigure out an erroneous but noncritical component or to temporarily reconfigure out a noncritical component because system overload is increasing the response time of the system above a critical point. Following reconfiguration, the system must be able to resume its processing. As an example, when during peak load the response time of the NAS is over the two second limit, noncritical functions are delayed for later processing.

A System Model

In evaluating software fault tolerance techniques, it is necessary to determine the types of faults which can be "tolerated" by each technique. Figure 1 shows the simplified model of an application system life cycle which we will use.

When building a computer model of a real world system, the essential properties of the real world system are first abstracted into a specification or model. The specification is then implemented in software and hardware. Following testing and validation, the system can be put into operational use. Figure 1 is a simplification since a computer system is actually made up of a layers of interconnected hardware and software components, but this simplified model is adequate for our purposes.

Using figure 1, there are three times in the system life cycle when faults can be introduced:

1. when the system specification is formulated. These modelling or specification errors are denoted by the mapping marked "a." Studies have shown that more than half the faults in a software system can be traced to specification errors [15].

2. when the model is implemented in hardware and software, denoted by the mapping marked "b." Errors occurring at this stage we will call implementation or consistency errors.

3. when the system is in operational use, denoted by the mapping marked "c." Faults here involve machine failures.

The focus in this paper will be on specification and implementation errors. Although software fault tolerance techniques may also be able to detect hardware faults, it is not clear that the application system level is the right level for their detection and handling.

N-Version Programming

In n-version programming, multiple versions of an algorithm are executed simultaneously and their results compared. If the results differ, voting or other strategies can be used to select one result.

If only one specification is used, then figure 2 models the result. Obviously the only faults which could possibly be detected are those caused by implementation errors since the implementation is the only place where redundancy occurs. It is important to note that because of the possibility of common faults, majority voting may not result in a correct answer. European experience seems to indicate common errors in about half of the redundant software systems developed to date [20]. Furthermore, voting causes an additional mapping to occur at run-time which introduces the possibility of additional faults. For example, the voting procedure must also be duplicated or in some way made

fault-tolerant. The resulting complexity may itself be the cause of failures, e.g. the the synchronization problem caused by back-up redundancy on the first Space Shuttle flight [6].

In order to have any chance of detecting specification errors with n-version programming, multiple specifications must be used (see figure 3). This requires building multiple models of the real world system, implementing them, and then voting on the results, hoping that a majority are correct.

Experimental studies are needed to determine how effective n-version programming is in detecting specification errors at run-time. Voting systems are effective only if the versions are independent, and it is not clear at this time whether an adequate level of independence is achievable or practical in terms of cost.

Two advantages of n-version programming are its conceptual simplicity and timeliness of results. Since all versions are run concurrently, there are no time delays for rollback and recovery.

One major weakness of n-version programming is that it does not make use of the programmability of software for introducing intelligent decision making into the output acceptance process. Majority vote is a weak test for correctness of results. Thus, n-version programming makes better sense if the simple voting algorithm is expanded to include other types of checks on the results of the n-versions such as reasonableness checks or comparison with run-time external information. In order to detect specification errors, either multiple specifications and implementations must be used, or run-time checks must use information which is independent of the specification, e.g. radar information or human monitors.

Another deficiency of n-version programming as proposed is that when there is no agreement on results (no majority exists), there is no provision for fail-soft or fail-safe facilities. This remains true even when if voting is replaced or supplemented by reasonableness checks or checks against external information. This is unacceptable for safety-critical systems.

Recovery Blocks

A second method of providing software fault tolerance uses recovery blocks [Randell]. A recovery block consists of a regular programming language block (called the primary block), an acceptance test, and a sequence of alternate blocks.

The acceptance test is a logical expression which is evaluated to determine if the result of a block is acceptable. If a primary or alternate block does not complete (because of an error or expiration of time limit) or fails the acceptance test, the state is restored to just prior to entering the recovery block, and the next alternate (if there is one) is entered. If all alternates fail to pass the acceptance test, recovery is attempted at the level of the next enclosing recovery block. If the acceptance test is passed, control passes to the statement after the recovery block. Prior states to be used for rollback are stored in a "recovery cache."

The acceptance test of recovery blocks can potentially be used to detect faults resulting from both implementation errors and specification errors (by using run-time specification-independent information). This statement must however be carefully examined. The technique can detect only those faults that cause the acceptance test to fail. But acceptance tests can capture only part of the intended functionality if the complexity and the performance of a system are to be kept at an acceptable level. Thus the possibility exists of accepting erroneous results.

An advantage of the recovery block is its simple control structure, although structuring systems of communicating processes adds complications and "domino effects" may be created when restoring the state of a process which communicates with other processes [19].

Another advantage is that fail-soft and fail-safe facilities can be supported by the recovery block structure since the alternates need not be identical to the primary block, though all use an identical acceptance test, and thus variability is somewhat limited. Furthermore, unlike n-version programming, the acceptance test provides that an acceptable result can be found even if a majority of the algorithms give unacceptable results.

The primary disadvantage of the recovery block structure is in real-time applications. Time requirements may be such that backward recovery means that results will arrive too late to be useful. This can be overcome by running the several alternates in parallel and providing a selection scheme to decide on a result, e.g. using the results of the first alternate to pass the acceptance test. More important, in some situations, backup to a prior state may be impossible. But the recovery block structure allows for no forward error-handling.

The recovery block structure also lacks the ability to differentiate between erroneous states and to provide for differential handling depending upon run-time information. A proposal to add safety assertions to recovery blocks in

order to allow this can be found in Leveson and Shimeall [14]. Thus the processing after a failure of an acceptance test cannot be decided dynamically to allow different responses under different conditions. In particular, dynamic reconfiguration as a response to run-time conditions is not supported. Moreover, all the decision making is performed in the context of the failed block, which may not have enough information to make the required recovery decisions.

Exception Handling

Exception handling consists of identifying exceptional conditions which require special processing, detecting those conditions at run time, and providing facilities for the program to respond, i.e. handle exception conditions. Thus, effectively, the detection of an exception condition corresponds to a failure of an acceptance test in the recovery block model.

Several proposals for and implementations of exception handling exist. These include the mechanisms of PL/I [11], Ada [12], CLU [16] and Mesa [22], and the proposal of Goodenough [8]. The mechanisms differ basically in the handling responses they support.

The replacement model of exception handling [23] supports all the responses of: termination (doing something and then terminating the operation detecting the exception or terminating a closed program construct containing the operation's invocation); resumption (doing something and then resuming the operation that detected the exception); retry (doing something and then retrying the operation); and propagation (doing something and then propagating the exception to higher layers of the system, uniformly and in a modular fashion). Since it supports more flexible responses and compile time checking than other mechanisms, the replacement model will be used in this paper.

Like recovery blocks, exception handling can detect faults arising from implementation and specification errors. In fact, it is possible to define the exception condition as the negation of the recovery block acceptance test and to test it at the end of the module exactly as in recovery blocks. Thus there is no distinction between the types of errors which can be detected and handled by recovery blocks and exception handling. However, unlike recovery blocks, an exception can be detected at any point in a program. This supports both early detection of input errors, as well as checking the correctness of the obtained results at the end of a program block. Of course, bad input data could be caught by waiting until (hopefully) unreasonable answers occur. This makes little sense in a time-constrained environment and also increases the likelihood of bad data

polluting the entire system. Thus exception handling's ability to check input assertions is very useful in the context of software fault tolerance.

The major advantage exception handling has over recovery blocks is that the flexible handler responses allow a handler to perform forward recovery actions in addition to backward recovery. This provides for better real time performance as discussed in the section on requirements. It also allows higher layers of the system to participate in a recovery action, after which the operation can resume where it left off, provided recovery was successful.

Since the replacement model supports exception handling in expressions as well as statements, it is possible to have handlers compute substitute values when the operation is value returning (such as an arithmetic function). The ability to substitute values for operations supports returning a safe but incorrect value as the result of a noncritical operation, e.g. a zero in the case of averaging a large number of values.

Resumption handling allows the handler to take corrective action, after which the operation resumes. The ability to take corrective action without losing current state information supports fast recovery. In the termination case, the operation must be completely restarted. Thus, any correct processing which has completed must be redone. As an example, in the case of a temporary system overload, if the response time for critical outputs has been degraded because of unusually heavy demands on the system, noncritical tasks can be temporarily reconfigured out of the system by the exception handler and later resumed. Thus the resumption response is particularly useful in the case where the system is in an unsafe but correct state (in the sense that there is not an algorithmic error).

Exception handling can support flexible fail-soft and fail-safe facilities. This can be done because of two features of exception handling. First, errors can be differentiated and can result in different exceptions being raised. Second, the invoking layer may select different handlers for the same exception raised in different invocations. In [17] it is argued that exception handling forces enumerating the possible failure modes of an operation and that this is a disadvantage. This is a misconception since it is always possible to distinguish only one exception, i.e. failure to meet some acceptance test. However, differentiating between different exception conditions is in many cases essential for selecting the appropriate recovery action.

Another important difference between exception handling and recovery blocks involves the context in which recovery

decisions are made. In the recovery block model, all recovery decisions are made within the block in which the error is detected. In the exception handling model, the basic procedure involves passing information about the erroneous state, via the exception name and parameters, to the context invoking the operation, allowing the handling decision to be made within this context.

If the handling of the error is fully decided within the operation, there is no way in which the invoker can influence the handling, and therefore each different response will require programming a separate slightly different operation. Exception handling allows more flexibility. The normal case can be encapsulated and the application allowed to use different handlings in special cases. For example, if the operation is noncritical in this context, it can be skipped. There may also be a set of acceptable degraded modes which can be selected depending on the state of the system, etc.

Using the recovery block there is only one predefined response for every exception. Using exception handling techniques it is still possible for the designer of the module to supply (within the module) one or more handlings for any of the exceptions raised by the operations of the module, e.g. alternate algorithms may be provided within the module implementing the operation. These are exported by the module to the invoker. The invoker can then select any of these handlers for a specific detection of an error. Thus exception handling procedures can deal with cases where some typical response is anticipated (e.g. trying an alternate algorithm) or cases where a response requires knowledge of information not available within the operation.

Another advantage of exception handling is that it provides one unified mechanism for both forward and backward recovery.

A disadvantage of exception handling is that the flexibility of response introduces more complexity into the language than the simple recovery block structure. This is partially offset by the ability to verify the exception handling code together with the rest of the program [24].

Summary

The usefulness of n-version programming, recovery blocks, and exception handling have been examined within the context of real-time safety-critical environments. Although exception handling provides the most flexible and complete facilities, no software fault tolerance technique is always effective. Therefore, program verification and other fault avoidance techniques should be used and facilities provided in the system to handle residual, "untolerated" failures.

Note to Reviewers

The final version of this paper will include program examples illustrating the techniques and the differences between them.

References

[1] Anderson, T. and P.A. Lee. Fault Tolerance Principles and Practice," Prentice Hall, 1981.

[2] Chen, L. and A. Avizienis. "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Eighth Int. Conf. on Fault-Tolerant Computing, Toulouse, France (June 1978).

[3] Cristian, F. "Exception Handling and Software Fault Tolerance," Proc. Symposium on Fault Tolerant Computing Systems, (June 1980), pp. 97-103.

[4] De Millo, R.A., Lipton, R.J., and A.J. Perlis. "Social Processes and Proofs of Theorems and Programs," CACM, vol. 22, no. 5 (May 1979), pp. 271-280.

[5] Elmendorf, W.R. ""Fault-Tolerant Programming," Digest of 1972 Int. Symp. on Fault-Tolerant Computing, 1972, pp. 79-83.

[6] Garman, J.R. "The Bug Heard 'Round the World," Software Engineering Notes, vol. 6, no. 5 (October 1981).

[7] Gerhart, S.L. and L. Yelowitz. "Observations on the Fallibility in Applications of Modern Programming Methodologies," IEEE Trans. on Software Engineering, vol. SE-2, (March 1976), pp. 195-207.

[8] Goodenough, J.B. "Exception Handling: Issues and a Proposed Notation," CACM, 18, 2 (December 1975).

[9] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," CACM, 12, 10 (October 1969)

[10] Horning, J.J. "Programming Languages," in T. Anderson and B. Randell (eds.) Computing Systems Reliability, Cambridge University Press, 1979.

[11] IBM OS PL/I Checkout and Optimizing Compilers: Language Reference Manual.

[12] Ichbiah, J. et. al. "Rationale for the Design of the Ada Programming Language," SIGPLAN Notices, 14, 6 (June 1979).

[13] Leveson, N.G. and Harvey, P. "Software Fault Tree Analysis," Journal of Systems and Software, vol. 3, in press.

[14] Leveson, N.G. and Shimeall, T. "Safe Recovery Blocks," Technical Report, University of California Irvine, 1982.

[15] Lipow, M. "Prediction of Software Errors," Journal of Systems and Software, vol.1, 1979, pp. 71-75.

[16] Liskov, B.H. and A. Snyder. "Exception Handling in CLU", IEEE Transactions on Software Engineering, SE-5, 6 (November 1979).

[17] Melliar-Smith, P.M. and B. Randell. "Software Reliability: The Role of Programmed Exception Handling," Proc. Int. Conf. on Reliable Software, SIGPLAN Notices, vol. 10, no. 6 (June 1975).

[18] Ramamoorthy, C.V., Bastiani, F.B., Favaro, J.M., Mok, Y.R., Nam, C.W., and K. Suzuki. "A Systematic Approach to the Development and Validation of Critical Software for Nuclear Power Plants," Proc. Fourth Int. Conf. on Software Engineering, 1979.

[19] Randell, B. "System Structure for Software Fault Tolerance," Proc. Int. Conf. on Reliable Software, SIGPLAN Notices, vol. 10, no. 6 (June 1975), pp. 437-449.

[20] Taylor, B. "Letter from the editor," Software Engineering Notes, vol. 6, no. 1 (January 1981).

[21] Wensley, J.H. et. al., "SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control," Proc. IEEE, vol. 66, no. 10 (October 1978), pp. 1240-1255.

[22] XEROX PARC Mesa Language Manual, March 1979.

[23] Yemini, S. "The Replacement Model for Modular Verifiable Exception Handling," Ph.D. Dissertation, UCLA, 1980.

[24] Yemini, S. "An Axiomatic Treatment of Exception Handling," Proc. 9th Symp. on Principles of Programming Languages, January 1982.

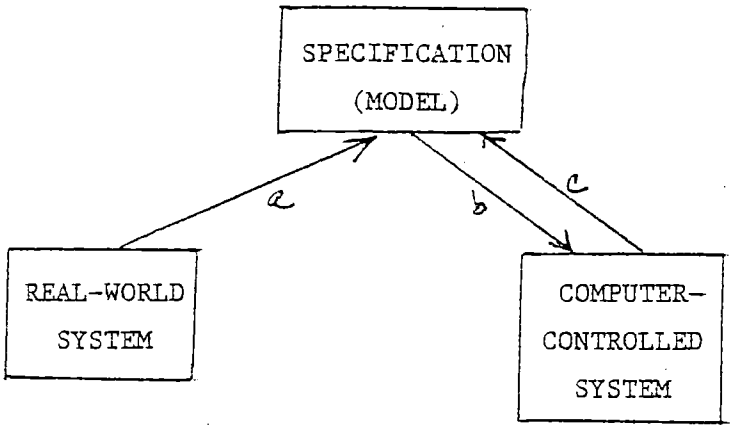

Figure 1.

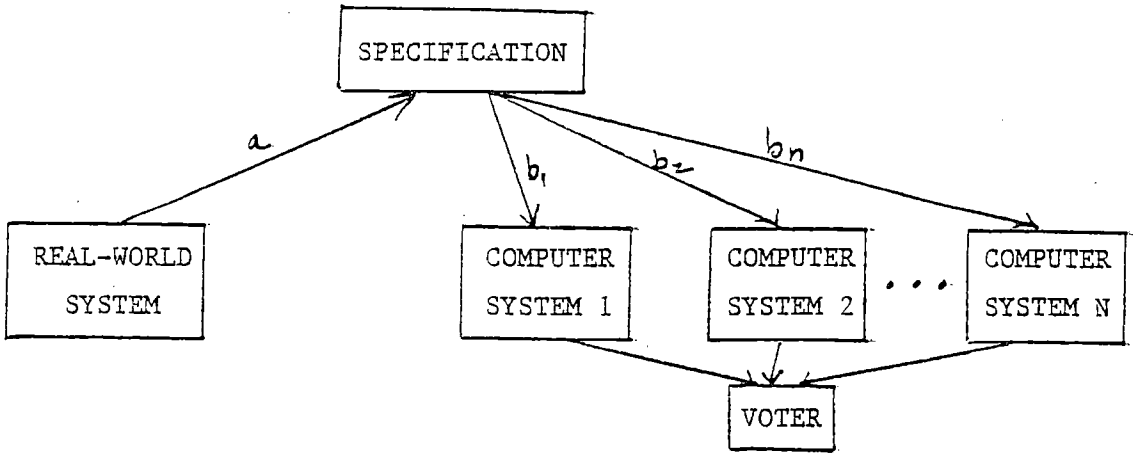

Figure 2.

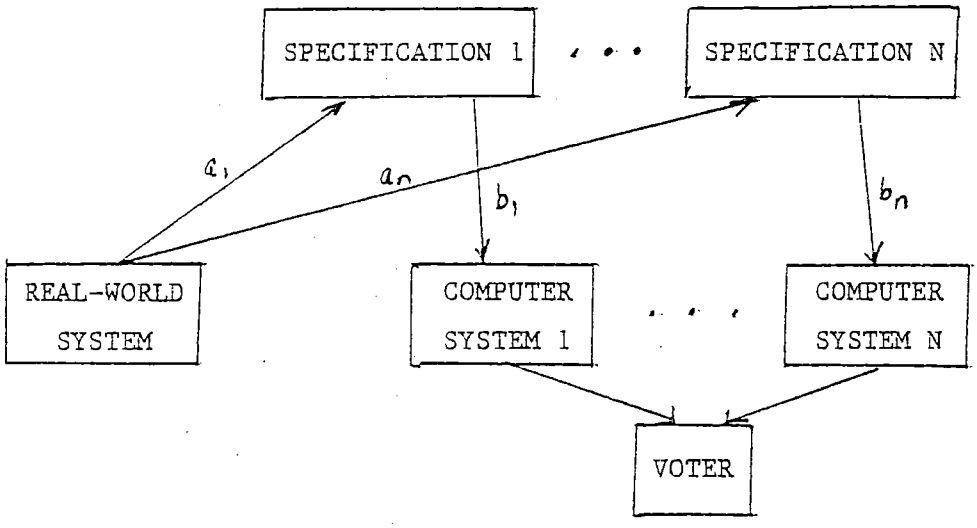
SPECIFICATION 1
SPECIFICATION N
a_1
a_n
b_1
b_n
REAL-WORLD SYSTEM
COMPUTER SYSTEM 1
COMPUTER SYSTEM N
VOTER

Figure 3.