Copyright © IFAC SAFECOMP '86
Sarlat, France, 1986

MULTI-VERSION SOFTWARE DEVELOPMENT

J. P. J. Kelly, A. Avižienis, B. T. Ulery, B. J. Swain, R.-T. Lyu, A. Tai and K.-S. Tso

UCLA Computer Science Department, University of California, Los Angeles, CA 90024, USA

Abstract. Multi-version software systems achieve fault tolerance through software redundancy. Diverse software versions are executed concurrently by a supervisory system that reports consensus results, allowing the results from erroneous versions to be masked by the majority. The Second Generation Experiment is a large scale empirical study of multi-version software systems engaging researchers at six sites. This paper presents UCLA's perspective of this experiment, its role in the preliminary analysis, and related research at the Dependable Computing and Fault Tolerant Systems Laboratory.

Keywords. Multi-version software, fault tolerance, N-version programming, reliability, dependability, software development, design diversity.

INTRODUCTION

Multi-version software (MVS) systems are gaining acceptance in the aerospace industry [Hills, 1985; Martin, 1982], nuclear power industry [Bishop, 1985; Ramamoorthy, 1981; Voges, 1985], and ground transportation industry [Taylor, 1981]. The multi-version approach to fault tolerant software systems involves the development of functionally redundant, yet independently developed software components. These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine final output values [Avižienis, 1985].

The NASA Langley Research Center is sponsoring the Second Generation Experiment in fault tolerant software which has been underway at several universities since 1984. During the summer of 1985, the Second Generation Experiment employed 40 graduate students to design, code and document 20 redundant software versions. The analysis of this software and its role in fault tolerant systems currently engages researchers at six sites.

The experiment was partitioned into several phases including the choice of a suitable application, specification of the problem, definition of the experimental protocol, selection and training of participants, generation of redundant software versions (detailed design, coding, unit testing), preliminary testing and analysis of the versions and prototype multi-version configurations, validation of the specifications, verification of the versions, formal certification of the versions, final analysis of the versions and the experiment, and documentation of the experiment. The experiment is now approaching the certification phase.

This preliminary report presents the Second Generation Experiment from UCLA's perspective. The experiment's progress and related research in fault tolerant software are also summarized. Empirical results from this experiment will be published jointly by NASA and all participating institutions after the final phases have been completed.

PREVIOUS EXPERIMENTS

Research on multi-version software was initiated at UCLA in 1975 [Avižienis, 1975]. From its beginning, the fundamental conjecture of the multi-version software system approach at UCLA has been that errors due to residual software faults will be masked by the correct results produced by the other versions in the system. This conjecture does not assume independence of errors, but rather a low probability of their concurrence. For instance, a multi-version system comprised of three versions is likely to tolerate a given fault if the probability of that fault occurring is sufficiently low -- not because the errors occur independently (they do not), but because they are unlikely to influence the majority decision of the system. It might also be noticed that these errors can even be tolerated if they occur frequently (on many input cases), provided simply that the other versions do not concur.

The decision algorithm may need to determine the decision result from a set of similar, but not necessarily identical, results. *Similar results* are defined to be two or more results (good or erroneous) that are within the range of variation that is allowed by the application. When two or more similar results are erroneous, they are called *similar errors* [Avižienis, 1984]. Analytic models using queuing and Markov modeling techniques have been developed for the prediction of the potential MVS reliability improvement, allowing for the existence of similar errors in MVS [Grnarov, 1980].

MVS systems achieve reliability improvements through the use of redundancy and diversity. A "dimension of diversity" is one of the independent variables in the development process of an MVS system. Diversity may be achieved along various dimensions, e.g., specification languages, specification writ-

ers, programming languages, programmers, algorithms, data structures, development environments, and testing methods.

DEDIX, the DEsign DIversity eXperimental supervisory system, is a powerful tool for the concurrent execution and analysis of MVS, that was developed at UCLA. With the aid of DEDIX and the collection of 20 redundant software versions, the relative significance of several of these dimensions of diversity is being investigated.

Zero Generation Experiment

UCLA has been actively engaged in the research of fault tolerant software since 1975. Since that time three generations of experiments have been conducted. The first investigation into the role of design diversity in multi-version software systems, the "Zero Generation" Experiment, was conducted by Chen and Avizienis [Avižienis and Chen, 1977; Chen and Avižienis, 1978]. This experiment was designed to study the effects of multiple algorithms on diversity. A small prototype application was implemented using three different algorithms specified in English. The primitive control mechanisms (decision points) were formulated to allow synchronization and decision making among the versions. It was also discovered that in order to prevent a faulty version from failing the entire MVS system, it is necessary to isolate the version. The Zero Generation Experiment demonstrated the feasibility of the MVS approach.

First Generation Experiment

The First Generation Experiment investigated the effects of specification languages on diversity [Kelly, 1982]. Eighteen programs were written with an average length of about 500 lines of PL/1 code. Seven programmers were supplied formal specifications written in OBJ, five were supplied non-formal PDL specifications. The remaining six programmers served as a control, working from a specification written in English.

Each program version handled exceptions internally to protect the system from failures that were encountered in the previous experiment. Exception handling allows the decision algorithm to base its decisions on more reliable values, i.e. those values flagged as bad are ignored.

The First Generation Experiment concluded that the multi-version software approach is a viable supplement to fault avoidance and removal. It was also found that specification errors are the most serious because they can lead to similar errors in the final versions. However, comparative testing of multiple versions has proven to be an excellent means of uncovering specification faults.

SECOND GENERATION EXPERIMENT

Encouraged by the results from the previous experiments in fault tolerant software, the NASA Langley Research Center began funding the Second Generation Experiment in 1984. This experiment was designed to evaluate the contribution of diverse program versions to fault tolerant software systems in a realistic aerospace application and to continue developing a multi-version programming methodology.

The Second Generation Experiment involves four universities, University of California at Los Angeles, the University of Illinois at Urbana-Champaign, North Carolina State University, and the University of Virginia, as well as the Research Triangle Institute (RTI), and Charles River Analytics (CRA). The specifications were written by RTI and CRA. Tools, acceptance tests and experimental coordination was provided by RTI. CRA has now assumed the role of customer and specification arbiter, and is providing flight simulation test data. Each university employed ten programmers to generate five program versions and is now contributing to the analysis phase of the experiment.

The Protocol

Work on the specification was begun in late 1984 after the application was chosen. The principal investigators held several meetings at which draft specifications were presented and subsequently corrected in an attempt to create a specification that was both typical of industrial practice and appropriately sized for the experiment.

Five independent programming teams at each of the four universities generated software from a common specification. A controlled software development process, uniform across all four universities, was designed to reflect standard industry practice. Additionally, programmers were not permitted to discuss any aspect of their work with members of other teams. Work-related communications between programmers and a central project coordinator (specification expert) were conducted via Unix™ mail. Copies of each question and answer pair were locally rebroadcast to all programming teams.

The experiment included ten weeks for software generation. These were organized into five phases:

1. *Training phase*: The programmers attended a brief training meeting. An introductory presentation was made summarizing the experiment's goals, requirements and the multiple version software techniques. At this meeting, the programmers were given written specifications and documentation on system tools.

2. *Design phase*: At the end of this four-week phase, each team delivered a design document following guidelines provided at the training meeting. Each team delivered a design walkthrough report after conducting a walkthrough which was attended by silent observers including the site's principal investigator.

3. *Coding phase*: By the end of this phase, programmers had finished coding, conducted a code walkthrough and delivered a code walkthrough report.

4. *Testing phase*: Each team was provided four sample test data sets. No two teams received the same test cases. Two weeks were allotted to this phase.

5. *Preliminary acceptance test*: Programmers formally submitted their programs. Each program was run in a test harness. When a program failed a test it was returned to the programmers with the input case on which it failed, for debugging and resubmission. By the end of this two week phase, all twenty programs had passed this preliminary acceptance test.

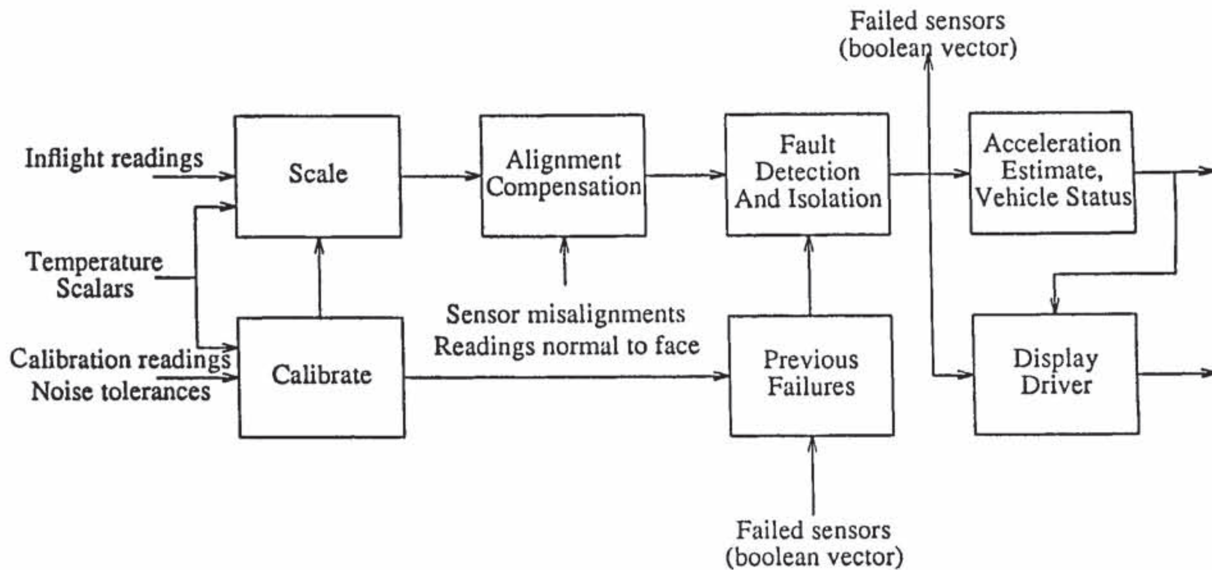

Fig. 1. System data flow diagram

The original specification is a 60 page document (including tables and figures) written in English [CRA, 1985]. The additional question and answer pairs that were broadcast among the programmers tripled the bulk of the specification. These specifications resulted in twenty Pascal programs which averaged over 2500 lines of code ranging from 1600 to 4800 lines.

A long and careful validation phase including extensive preliminary testing of the versions followed the 10-week software generation phase. During validation, many errors and ambiguities in the specification (including the electronic communications) were revealed. The specification has now been restored to a single document, a document that has benefited from the scrutiny of more than 50 motivated programmers and researchers. Additionally, many software tools and detailed testing procedures have been developed for the verification, certification and analyses of the 20 versions.

It is interesting to note how a small number of errors in the original specification led to numerous ambiguous and contradictory addenda in the form of question and answer pairs. The cause for this confusion appears to be attributable to the requirement that the central coordinator answer every question personally, which put a great deal of pressure on the central coordinator to answer quickly rather than well. The number of questions (over 250) posed by the 40 programmers was overwhelming. Although most questions derived from only a handful of errors and ambiguities in the original specifications, each was phrased differently so that simple affirmative or negative responses were interpreted to have extraneous and contradictory ramifications. In an attempt to give definitive explanations that would curb the need for further questions, a series of ten "announcements" was issued. However, even some of these announcements were revoked. In the end, the specifications had grown unwieldy and imprecise, rather than clear and precise.

The Application

A Redundant Strapped Down Inertial Measurement Unit (RSDIMU) is part of an integrated avionics system. This unit contains eight linear accelerometers mounted on the four triangular faces of a semioctahedron. Each accelerometer measures the component of acceleration along its axis. This fault tolerant configuration requires a special component to manage sensor redundancy and to reconfigure the system in the event of sensor failures. Each application program serves this management function, computes acceleration estimates, and drives a display panel that provides the non-interactive user interface to the system.

A significant amount of linear algebra, particularly matrix transformations, is involved. The original specification was simplified to fit the schedule of the experiment. There are eleven reference frames of interest (coordinate systems corresponding to the earth, vehicle, sensors, etc.), four of which are non-orthogonal. The programs are provided raw data measurements from the eight linear accelerometers. They are required first to identify faulty sensors and then to compute a statistical estimate of vehicle acceleration based on the redundant set of operational sensors. Final system status and acceleration estimates are reported by a digital display panel as specified by an input parameter. A block diagram of the system is shown in Fig. 1.

Goals

The list of objectives for the Second Generation Experiment is extensive. Experimentation concerns have been focused on the development of multi-version systems, primarily design and testing issues, and the modeling and analysis of these systems. The following lists some of our primary research concerns.

Reliability improvement of multiple versions over single versions. For ultra-high reliability applications, the multiple version approach to software engineering is being employed as a substitute for the conventional single version approach [Bishop, 1985; Hills, 1985; Martin, 1982; Taylor, 1981]. Empirical measurements of reliability and failure rates for both single and multiple version systems are being gathered and analyzed.

Modeling reliability with similar errors. Statistical data on similar errors is being gathered to validate a model of reliability that incorporates the effects of similar errors [Dorato, 1986; Eckhardt, 1985]. MVS systems are predicated on the assumption that when errors occur, their effects will generally be masked by correct results produced by the other versions in the system. Any system whose components exhibit independent failures is easily modeled statistically, and the reliability improvement of the system over an individual component is dramatic. A primary concern of this experiment is to quantify the effects of similar errors in independently developed, redundant software versions.

Qualitative investigations into the causes of errors. Similar errors are of particular interest to the study of MVS systems. If, for some subset of the input domain, a majority of the versions produce a similar error, rather than the correct response, then the benefits of the MVS system are negated and our confidence in an erroneous response amplified. It is therefore critical to understand the causes of similar errors in order to facilitate their avoidance or removal. The causes of software errors are of general interest in software engineering, and the existence of multiple software versions makes it easier to detect, locate and categorize sources of errors.

The role of recovery. Recovery is a way to prevent errors that occur early in the computation from propagating [Tso, 1986]. Decision points (cross-check points) are the interfaces between program versions and the supervisory system [Avižienis, 1985]. They are used to communicate intermediate and final values to the decision algorithm and may be used to return values for recovery. Recovery involves the injection of "consensus" data values into a faulty version, where the consensus values are determined by the decision algorithm of the MVS system. These values are then used in subsequent computations. In this way versions are recovered as they fail. Otherwise, failed versions must be excluded, causing the system to degrade. The alternative involves a passive supervisory system that compares values and generates a consensus, but does not return these values to disagreeing versions.

Granularity of comparisons. In order to establish the consensus opinion among several versions, the decision algorithm must compare and classify computed values. The simplest scheme involves bit-wise comparisons of results. This approach is too unforgiving under most circumstances, requiring, e.g., exact equality of computed real-valued outputs. At the other extreme, if all output variables are lumped together for a single decision, then the number of faults tolerated by the system will decrease drastically. In this experiment, the level of resolution used by the decision algorithm is determined from the semantics of the outputs.

Testing. Multi-version software development offers interesting new approaches to testing. Particularly, automated testing procedures may be developed which simply look for discrepant responses from the versions. This approach does not involve the expense of predicting correct output values, and therefore allows much more extensive testing. This approach is particularly appealing for rapid software development.

CURRENT RESEARCH ISSUES

All of the aforementioned topics are currently receiving attention at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. Attempts at developing metrics for the various dimensions of diversity have yet to produce results. However, considerable insight into the relative impact of these dimensions has been gained. Most notable is the role of specifications. The need for complete and unambiguous specifications is even more acute for multi-version systems than for single version systems. In general, this project has fostered an increased awareness of the critical role of software engineering methods in the development of highly reliable software systems.

At this point, preliminary analyses of the experiment and the twenty program versions is complete. Our primary research concerns have been twofold: identifying and defining the key factors in developing highly reliable software systems, and analyzing the twenty redundant versions developed in the Second Generation Experiment. Complete and unambiguous specifications, and a thorough acceptance test have been identified as two particularly critical factors in the development of highly reliable software systems.

Testing

Random and "hand" testing have been used extensively at UCLA to evaluate the versions' performance both redundantly (in various combinations) and individually. In our effort to automate the error detection process, we have found definitions of correctness elusive. The major obstacles to defining correctness are the classification of responses into consensus groups, and the resolution of multiple correct output values.

In order to measure reliability, one must first define the correct behavior of a program. Clearly a program behaves incorrectly if it fails to deliver the desired service. But consider a program with an identifiable and inadvertent fault, that is some deviation from the prescribed algorithm. Now consider that the numerical significance upon the output is insignificant, i.e., the program behaves adequately when run alone. Subtle numerical errors present a special challenge to automatic error identification and classification.

Defining equivalence. When two or more versions simultaneously generate equivalent values for a given output variable, then these values are deemed similar [Avižienis and Kelly, 1984]. In order for the decision algorithm to determine a consensus, it must decide which results are similar. Boolean and integer results are typically required to be identical. Character strings may be allowed cosmetic variations such as spacing and capitalization.

Real numbers have proven more difficult to classify. For the purposes of this experiment, it is necessary to define

equivalence classes among 20 output results. The versions produce outputs that cluster, but that are not randomly distributed. In fact, the use of different algorithms by the versions leads to subclustering. Statistical literature does not appear to address the determination of a "center" value from a sample distribution of this sort. One obvious solution is to use the median value. Unfortunately, there are special cases when this is clearly not the best solution, such as when the dominant cluster lies completely to one side of the median. If there is a largest cluster, we take the median of that cluster as the consensus. Every value within a fixed (application dependent) interval about that median is considered equivalent.

Complex data types also require special attention. An example of a complex data type is the set of bit patterns used to drive 7-segment digital read-outs. Making a decision on each bit individually might generate a nonsensical display; treating all seven bits as a single entity excludes the contribution of a version that intends the correct display pattern, but has made an error in the details of its representation.

The general problem may be viewed as one of granularity or semantics. Consensus may be determined at the bit level, at the basic type level, or at the level of the complex types. A purely semantic approach might suggest determining consensus at the highest level, although a computational perspective might suggest a finer resolution. One extreme approach is to view all final output variables as a single result that is either correct or incorrect [Knight and Leveson, 1985]. Under this interpretation, a triple modular redundancy system reaches no consensus if two versions fail on different variables, despite the fact that there is a consensus on each variable. For example, suppose that the system is supposed to produce the two results, (A,B), and that the versions actually produce (A,B), (A,X), and (Y,B). By treating the output as a single result, this system has no consensus despite the fact that there is a consensus on each of the output values. At the other extreme, bit-wise comparisons do not tolerate insignificant differences in real numbers or cosmetic differences in text [Kelly, 1982]. We are currently using a decision algorithm that determines consensus at a semantic level, comparing most values at the granularity of the basic types.

Resolving Multiple Correct Results. Multiple correct output values have been a significant obstacle to automating our testing process. In this application, one way to identify a faulty sensor is to observe that its readings are noisy, i.e., that the standard deviation over a set of values exceeds some prescribed threshold. Should the noise level be precisely at this threshold, two algorithms could conceivably produce different results, one indicating that the sensor is noisy, the other indicating that it is operational. The effect of slight numerical differences is seen by the supervisory system as totally contradictory responses.

This problem may be solved by introducing an additional decision point that uniformly returns to the versions a value of the standard deviation. The versions may then compare this value to the threshold. Note that this approach assumes that the supervisory system performs recovery, i.e., it does not passively observe the versions' outputs. The generality of this approach has not yet been determined. There is a significant cost associated with the decision-making process when it involves interprocess communications. There is also some concern that decision points limit design diversity. Thus the number of decision points is kept to a minimum.

Recovery.

Five decision points were specified for the RSDIMU application. To avoid restricting design diversity, programmers were not told where to place decision points in their programs. The sequence in which the decision points occurred was specified, and it was required that the variables of each decision point be computed but not yet used when the decision point is reached. The programmers were also required to use the (possibly modified) values returned by the decision algorithm in subsequent computations.

The preliminary acceptance test of the twenty versions did not test recovery. It ensured that the decision points were placed in the right sequence, but output values were checked at the end of the execution of each version. The validation phase revealed that the versions do not handle recovery properly.

The faults can be classified into two categories: incorrectly located decision points, and unused returned values. Some teams inserted decision points at the point where a value was first calculated. Under some circumstances, such as the detection of a sensor failure, values would be later revised to reflect the failure. The corresponding decision points were located too early in the procedure. Some versions were found to use computed values before passing them to the decision algorithm. These decision points occur too late. Finally, some versions ignore the returned values. They maintain redundant variables whose values are not updated to reflect the values returned by the decision algorithm.

These faults can be detected easily by special tests. In the verification phase of the experiment, the output values are checked at the decision points. This testing detects the incorrect placement of decision points. Also specific tests are included that deliberately return new values to the decision points. The results of the next decision point are then checked to verify that the returned values were actually used.

Specifications

Another continuing area of research in the fault tolerant software group at UCLA is formal specifications. MVS systems require complete and unambiguous specifications. No details concerning program functionality may be left unspecified. For example, it is not sufficient to state that whenever a particular exception occurs, an "appropriate" error message should be printed. Decision algorithms are not (yet) sophisticated enough to generally recognize semantically equivalent statements. The use of formal, high-level specification languages is a promising approach to the production of high quality initial specifications.

UCLA has been rewriting the existing English specification in the formal specification language Larch [Guttag, 1985]. We have received considerable assistance from the language developers, J. V. Guttag of MIT and J. J. Horning of the DEC Systems Research Center. The simple and elegant syntax of Larch is able to capture most of the functional requirements of this application without introducing excessive implementation bias which might limit diversity. The experience of rewriting the specification in Larch has clearly shown that when one is

forced to express the requirements formally, ambiguities and inconsistencies become more apparent [Tai, 1986]. The formal language prevents the expression of ambiguous statements, but there is still a dearth of software tools needed for consistency checking. We are now considering how to integrate tables and figures with formal Larch specifications for increased readability.

CONCLUSIONS

The benefits of a redundant development process clearly extend beyond those anticipated for the final MVS system alone. While MVS systems tolerate dissimilar errors, the multi-version approach to software engineering facilitates the avoidance and removal of faults. The validation and verification phases of this experiment benefited significantly from the redundant programming efforts: errors in the specifications were more likely to be detected; a single version that correctly computed a particular output was sufficient to cause an investigation into the discrepancy between that version and the other versions which contained a fault(s) on that output. The investigation uncovered faults among the other versions and the testing tools.

Preliminary analyses of the twenty versions have provided some insight into the causes of similar errors. The most prevalent cause has proven to be the specifications. Boundary conditions, exceptions, and other such design and coding errors appear to be both less frequent and less correlated. This observation strongly supports multi-version software as a means of diminishing the number of faults introduced during the design and coding phases of development.

Although similar errors are potentially a significant obstacle to MVS systems, and specifications prove to be their primary source, it is reassuring to note that the redundant development process was to credit for the discovery of a large number of specification errors that would otherwise have passed unnoticed until much later, at which time their correction would have been much more costly.

In the process of testing the 20 versions, we have been faced with many interesting challenges that are of direct relevance to the design of decision algorithms. Many of our solutions have been designed specifically for this application. However, development of a theory of decision algorithms is underway, which will lead to the design of application independent decision algorithms.

Empirical results from this experiment will be published by the cooperating institutions after the verification, certification, and final analysis phases are complete. These will include measures of reliability improvements, statistics on software errors, and more detailed discussions of the causes of errors.

Acknowledgements.

The work reported here is part of a large joint effort involving the collaboration of many individuals. It is with pleasure that we acknowledge the efforts of the following people that have contributed to this research: D. Eckhardt and L. Lee of the NASA Langley Research Center, R. Campbell of the University of Illinois, J. Knight of the University of Virginia, D. McAllister of North Carolina State University, A. Caglayan of Charles River Analytics, J. McHugh and L. Lauterbach of the Research Triangle Institute, and the 40 programmers who produced the program versions. This research has been supported by NASA grant NAG1-512.

REFERENCES

Avižienis, A., "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," pp. 450-464 in *Proceedings 1975 International Conference on Reliable Software*, Los Angeles, California (April 21-23, 1975).

Avižienis, A. and Chen, L., "On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution," pp. 149-155 in *Proceedings COMPSAC 77* (1977).

Avižienis, A. and Kelly, J.P.J., "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer* Vol. 17(8), pp.67-80 (August 1984).

Avižienis, A., Gunningberg, P., Kelly, J.P.J., Lyu, R.T., Strigini, L., Traverse, P.J., Tso, K.S., and Voges, U., "Software Fault-Tolerance by Design Diversity; DEDIX: A Tool for Experiments," pp. 173-178 in *Proceedings IFAC Workshop SAFECOMP'85*, Como, Italy (October 1985).

Bishop, P., Esp, D., Barnes, M., Humphreys, P., Dahll, G., Lahti, J., and Yoshimura, S., "Project on Diverse Software - An Experiment in Software Reliability," *Proceedings IFAC Workshop SAFECOMP'85* (October 1985).

Charles River Analytics and Research Triangle Institute, "Redundancy Management Software Requirements Specification for a Redundant Strapped Down Inertia Measurement Unit," Version 2.0, (May 30, 1985).

Chen, L. and Avižienis, A., "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," pp. 3-9 in *Digest of 8th Annual International Symposium on Fault-Tolerant Computing*, Toulouse, France (June 1978).

Dorato, K., "Coincident Errors in N-Version Programming," *Master Thesis*, UCLA, Computer Science Department, Los Angeles, California (June 1986).

Eckhardt, D.E. and Lee, L.D., "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transaction on Software Engineering* Vol. SE-11 (12), pp.1511-1517 (December, 1985).

Grnarov, A., Arlat, J., and Avižienis, A., "On the Performance of Software Fault-Tolerance Strategies," pp. 251-253 in *Digest of 10th Annual International Symposium on Fault-Tolerant Computing*, Kyoto, Japan (1980).

Guttag, J.V., Horning, J.J., and Wing, J.M., "Larch in Five Easy Pieces," Report No. 5, Digital Equipment Corporation Systems Research Center, Palo Alto, California (July 24, 1985).

Hills, A.D., "Digital Fly-By-Wire Experience," Proceedings AGARD Lecture Series (143) (October 1985).

Kelly, J.P.J., "Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach," CSD-820927, UCLA, Computer Science Department, Los Angeles, California (September 1982).

Knight, J.C. and Leveson, N.G., "Correlated Failures in Multi-Version Software," *Proceedings IFAC SAFECOMP'85*, pp.159-165 (October 1985).

Martin, D.J., "Dissimilar Software in High Integrity Applications in Flight Controls," pp. 36.1-36.13 in *Proceedings AGARD-CPP-330* (September 1982).

Ramamoorthy, C.V. *et al.*, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Transactions on Software Engineering* Vol. SE-7(6), pp.537-555 (November 1981).

Tai, A.T., "A Study of the Application of Formal Specification for Fault-Tolerant Software," *Master Thesis*, UCLA, Computer Science Department, Los Angeles, California (June 1986).

Taylor, R., "Redundant Programming in Europe," *ACM SIGSOFT* Vol. 6(1) (January 1981).

Tso, K.S., Avižienis, A., and Kelly, J.P.J., "Error Recovery in Multi-Version Software," in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France (October 1986).

Voges, U., "Application of a Fault-Tolerant Microprocessor-Based Core-Surveillance System in a German Fast Breeder Reactor," *EPRI-Conference* (April 9-12 1985).