155p.

NASA Contractor Report **172385**

Study of Fault - Tolerant Software Technology

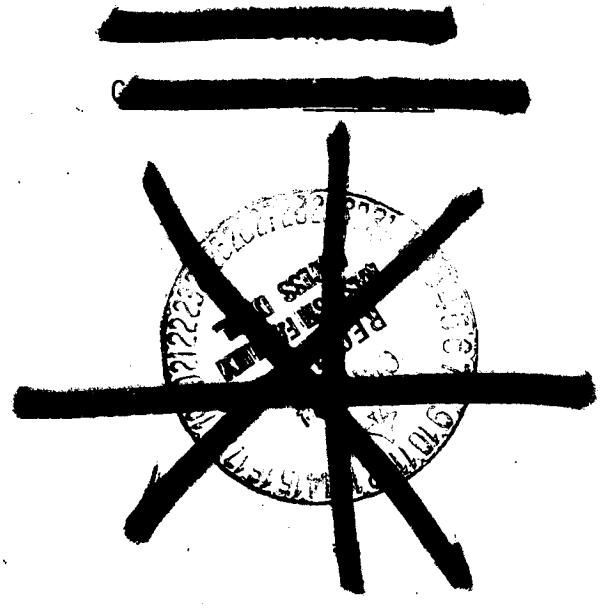
T. Slivinski, C. Broglio, C. Wild
Mandex, Inc.

J. Goldberg, K. Levitt
SRI, International

E. Hitt, J. Webb
Battelle Memorial Institute

Contract NASI-17412

SEPTEMBER 1984



National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report **172385**

Study of Fault - Tolerant Software Technology

T. Slivinski, C. Broglio, C. Wild
Mandex, Inc.

J. Goldberg, K. Levitt
SRI, International

E. Hitt, J. Webb
Battelle Memorial Institute

Contract NASI-17412

SEPTEMBER 1984

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

TABLE OF CONTENTS

1.0 Introduction

1.1 General

This report presents the results of a concerted study into the current state of the art of fault-tolerant software, and the implications of fault-tolerant software on future computer architectures, operating systems, and languages.

The study was conducted under the aegis of the National Aeronautic and Space Administration (NASA) Langley Research Center at Hampton, Virginia and included representatives from Mandex, Inc., SRI International, and Battelle Memorial Laboratories, (Columbus) as well as internationally known consultants. Research was performed under NASA Contract NAS1-17412 during the period 1 October 1983 through 30 April 1984.

Work on the study was under the overall direction of Mandex, Inc. with Dr. Thomas Slivinski serving as Project Director. Responsibilities were as follows:

- Battelle (Jeffrey Webb and Ellis Hitt) - responsible for assessment of the current state of the art of fault-tolerant software and quantitative evaluation techniques;
- SRI International (Jack Goldberg and Karl Levitt) - responsible for assessing the impacts of fault-tolerant software on computer architectures and operating systems;
- Mandex, Inc. (Christian Wild) - responsible for assessing the impacts of fault-tolerant software on higher level languages;
- Consultants (Thomas Anderson, University of Newcastle-upon-Tyne, and John Kelly, UCLA) - responsible for technical review and continuity;
- Mandex, Inc. (Thomas Slivinski and Carlo Broglio) - responsible for integrating and correlating the individual results and developing overall study conclusions and recommendations.

1.2 Background

The requirements for fault-tolerant software have been advanced since the early 1970's when such pioneers as Avizienis [AVIZ77] and Randall [RAND75] developed systematic techniques for applying redundancy to software. This interest arose (and continues today) from a variety of factors which make techniques for improving the reliability of software extremely important.

First and foremost, software costs and reliability are a major concern in the life cycle of systems. As hardware reliability has been improved and as the costs for hardware have been reduced, software faults have emerged as the most significant problem in new critical systems. Furthermore, as new applications become more complex, the reliability of the software increases in significance.

Second, over the years programmers had developed a collection of defensive programming techniques to assist in debugging the software and to deal with certain classes of software faults. What had not been accomplished is a systematization of these techniques into an orderly structure which could predict which techniques to apply, when to apply them, and to aid in the recovery of a consistent state. The concept of fault-tolerant software is to apply a systematic structure and to extend what has been largely an "ad hoc" approach.

Traditional software development techniques, employing extensive testing and debugging, had been shown to be very expensive and of diminishing value. The effectiveness of testing and debugging is limited by the Law of Diminishing Return and may even be bounded in the reliability that can be achieved. Methods which can improve the reliability of software beyond traditional testing are needed now more than ever.

1.3 Objectives

The objectives of this fault-tolerant software study were in two broad areas of interest.

The first was to determine the current state-of-the-art of fault-tolerant software. The study was specifically oriented toward answering such questions as: What techniques have been developed? What has been the experience with the techniques? Is one technique better than the others for various classes of problems or applications? Where is research work currently underway or planned? What results have been found? Most important, the study sought to answer the question: Can it be demonstrated that fault-tolerant software will improve the reliability of computer

systems? (i.e. is fault-tolerant software better than software which has been developed using traditional techniques?)

The second objective was to determine the implications of fault-tolerant software on computer hardware and software support systems. Specifically, the study sought to answer the following type of questions: What hardware features or support tools are needed for fault-tolerant software to operate efficiently and effectively? What facilities in the hardware, operating system and higher level application languages are needed to support each of the major techniques? What capabilities are desirable but not essential? Are the features, tools and environment consistent with the known future of computer architectures, or are substantial changes needed in these architectures to accommodate fault-tolerant software?

This second area led the study to look briefly at the Draper design architecture for the Advanced Information Processing Systems (AIPS). [CSDL83]

1.4 Summary of Results

This study concludes that fault-tolerant software technology is ready for application to new systems. While there is a great deal of development to be done and empirical data to be collected, all evidence indicates that this technology has progressed sufficiently that it is ready to move out of the laboratory into practical systems.

The major findings of this study are:

- Fault-tolerant software has been shown in limited experience to improve the reliability of systems;
- Fault-tolerant software has matured into a viable technology which is ready to be included in new systems;
- New hardware, operating systems and language developments are favorable to the incorporation of fault-tolerant software into new systems.

1.4.1 Fault-Tolerant Software and Reliability

Fault-tolerant software can improve the reliability of systems. This study found no evidence that, properly applied, fault-tolerant software will degrade the reliability of the resulting system and some evidence that its use will increase system reliability. In practical implementations, fault-tolerant software has been shown to detect errors in requirements, design and coding. Examples both in laboratory test beds and in operational systems have shown

reliability improvements. The modeling efforts that have been done all predict added systems reliability if the fault-tolerant software is properly implemented.

1.4.2 Fault-Tolerant Software Technology

Fault-tolerant software has matured into a viable technology. During the past ten years there has been an increasing use of fault-tolerant software in new systems and applications ranging from aerospace and military to nuclear power plant control and transportation. Regulatory agencies, such as the FAA and Atomic Energy of Canada, Limited, who are primarily concerned with public safety, have begun to require the use of dissimilar software and other fault-tolerant software techniques in certifying systems.

The concepts of fault-tolerant software have also undergone significant growth and refinement; techniques have been refined and analytical models developed. A second generation of research is underway at such institutions as UCLA, UVA, Newcastle-upon-Tyne, and governmental agencies such as the Army and Air Force.

What is currently lacking is the base of empirical data needed to verify and validate the results of the models and to transfer the technology from pure research into a development mode.

1.4.3. Implications on Future Technology

New systems architectures, operating systems, and languages are favorable to support fault-tolerant software. New systems support many of the notations and mechanisms needed to efficiently implement fault-tolerant software. These include encapsulation (concepts for constraining the effects of computations so that problems can be identified and corrected), hierarchy (the layering of hardware, operating systems and application functions into a well defined structure), and concurrency (federated and distributed processing and parallel actions).

In addition, lower hardware costs have increased the opportunities for building the mechanisms needed for efficiently achieving the redundancy needed to implement various fault-tolerant software techniques. Special purpose devices such as special memories, monitors, and even the use of additional general purpose processing units are now feasible.

In relation to languages, Ada and other newer languages, although not perfect, support many of the notations and constructs needed to write fault-tolerant functions into software.

The finding is that the efficient implementation of fault-tolerant software in newer systems has been greatly facilitated by the newer hardware, operating systems and languages.

2.0 Fault-Tolerant Software Overview

2.1 Introduction

This chapter presents an overview of the fault-tolerant software process. The model of this process provides the basic common steps that must be considered in any truly fault-tolerant software technique. This model is also the basis for descriptions and analyses used throughout this paper.

2.2 Software Faults

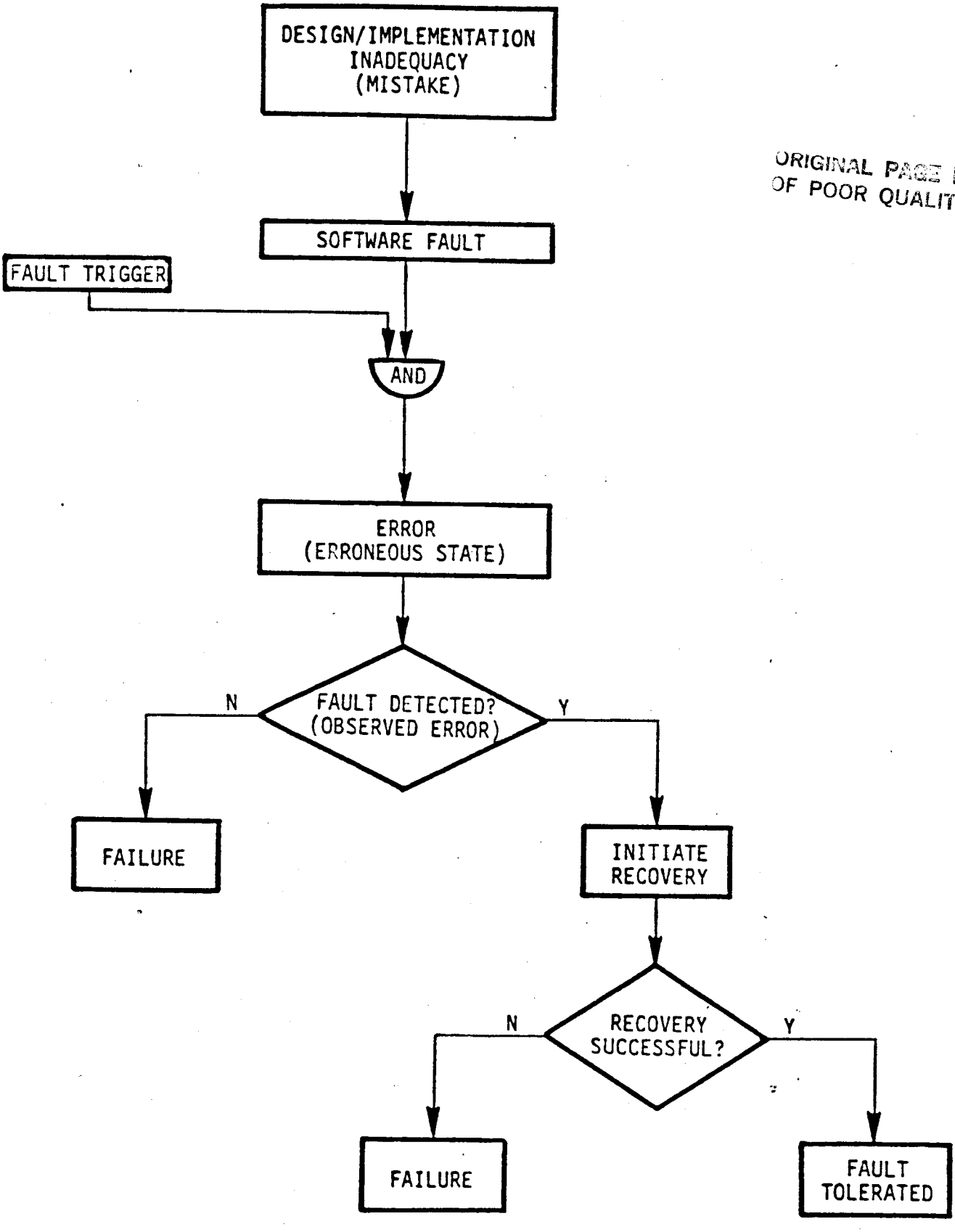
A software fault is any defect within a software component (e.g., a module, a procedure, a process, a collection of processes etc.). Software faults may be due to mistakes in specification, in translating specifications into a design, or in implementing the software design. A fault manifests itself as an erroneous system state or error. If an error cannot be tolerated, the system may fail. A failure occurs whenever the external behavior of the system does not conform to that prescribed by the system specifications, or more broadly perceived specifications. [ANDE81] Figure 2-1 represents the relationship between these terms.

Software faults are thus caused by human mistakes. This is in contrast with hardware faults which are caused by physical wear out as well as design mistakes. [Note: Some hardware faults will cause a change in the software code or sequencing; such faults are not defined as software faults in this study although other researchers include these as software faults (e.g., [SONE81]). The case where software accepts external-to-the-system faulted input as good input is a software fault. Although all software faults are due to human mistakes and, as such, might seem to cause only unanticipated faults, there can be anticipated software faults as well. Examples of this type of fault are divide-by-zero and overflow faults.

2.3 The Fault-Tolerant Software Model

A description of the general process of tolerating software faults has been developed to provide a unified interpretation of what fault-tolerant software does, as well as a set of common terminology. Fault-tolerant principles can be discussed as four phases: error detection, damage assessment, error recovery and fault treatment. [ANDE81]

FIGURE 2-1 FAULT-TOLERANT SOFTWARE EVENT RELATIONSHIPS



2.3.1 Error Detection

The identification that a fault exists can only be accomplished by detecting the effect of the fault, i.e., that an erroneous state exists. Detection of an erroneous state requires first, some form of redundant information and second, a means of analyzing this information. Because software faults are caused by mistakes in specifications, translation, design, or implementation, simply replicating programs and comparing results will not identify software faults because the same results will be produced in each duplicate. In order to detect software faults, it is necessary that the redundant versions be independent of each other, that is, of diverse design [AVIZ82] (see multi-version software and recovery blocks). Structural information about the internal design and construction of software can also be used to identify errors. Examples are, execution path monitoring and timing checks. Structural information can also be used to monitor the reasonableness of the intermediate and final results. Examples include inverse algorithm checks, range checks and rate-of-change checks. Redundancy is necessary but not sufficient in itself for error detection and recovery. Analysis of the redundant information is also required.

2.3.2 Damage Assessment

When an error has been discovered, it is necessary to determine the extent of the damage done by the fault before error recovery can be accomplished. Assessing the extent of damage is usually related to the structure of the system. Assuming timely detection of errors, the assessment of damage is usually determined to be limited to the current computation or process. The state is assumed consistent on entry. An error detection test is performed before exiting the current computation. Any errors detected are assumed to be caused by faults in the current computation.

One conceptual technique to isolate the effects of errors is the encapsulation of processes. Encapsulation is the concept of containing the effects of an action to only the objects to which that action has legal access. Through the proper enforcement of encapsulation the designer can assess the extent of the damage to a logical structure due to a fault.

All systems provide encapsulation to some degree. However, there is usually a significant difference between the structure intended by the source code of a software system (with its implied encapsulations) and its implementation. For example, most systems provide encapsulation for the abstraction of a process but not down to the individual procedures or subprograms which make up that process. Encapsulation is necessary to prevent one program from

destroying the address space of another program. However, little support is usually provided by the underlying machine architecture for the encapsulation implied by procedures. This lack of support can greatly complicate the assessment of damage caused by a fault. For example, assume a procedure contained a fault which occasionally generated arbitrary pointer values (addresses). The damage done by arbitrarily changing values anywhere in a program's memory space would be very difficult to assess. In this case the detection of the error may not occur until very much later during the processing. Even if an error was detected during that running of the faulty procedure, the extent of the damage would be difficult to determine.

2.3.3 Error Recovery

After the extent of damage has been determined, it is important to restore the system to a consistent state. (The term "state" refers to the state of the current computation. The current computation relates to the processing being performed at the intended encapsulation. This could be a procedure, a process or a collection of processes (domain) engaged in a conversation [RAND75]). This is the purpose of the error recovery phase. There are two approaches, backward and forward error recovery. In backward error recovery, the system is returned to a previous (presumably) consistent state. The current computation can then be retried with existing components (retry) (1) with alternate components (reconfigure), or it can be ignored (skip frame) (2). The use of backward recovery implies the ability to save and restore the state. This can exact great performance penalties if not carefully controlled.

Forward error recovery attempts to continue the current computation by restoring the system to a consistent state, compensating for the inconsistencies found in the current state. Forward error recovery implies detailed knowledge of the extent of the damage done, and a strategy for repairing the inconsistencies. While this may be possible in certain cases of anticipated faults, it is difficult to conceive of appropriate strategies in the case of unanticipated faults. Where feasible, forward error recovery is usually more efficient and less demanding on system resources than backward error recovery.

---

(1) This is only useful for transient timing on hardware faults.

(2) For example, in a real-time system, no processing for the current computation is accomplished in the current frame, sometimes called "skip frame".

2.3.4 Fault Treatment

Once the system has recovered from an error, it may be desirable to isolate and/or correct the component which caused the error condition. Fault treatment is not always necessary because of the transient nature of some faults or because the detection and recovery procedures are sufficient to cope with other recurring errors. For permanent faults, fault treatment becomes important because the masking of permanent faults reduces the ability of the system to deal with subsequent faults. As stated earlier, some fault-tolerant software attempt to isolate faults to the current computation by timely error detection. Having isolated the fault, fault treatment can be done by reconfiguring the computation to use alternate forms of the computation to allow for continued service. (This can be done serially, as in recovery blocks, or in parallel, as in N-Version programming.) Of course, the assumption is that the damage due to faults is properly encapsulated to the current computation and that error detection itself is faultless (i.e., detects all errors and causes none of its own).

3.0 Modern Software Fault-Tolerant Methods

The purpose of this chapter is to identify and categorize fault-tolerant software techniques. The next chapter, Chapter 4, will document the research and experiences with these techniques. Table 3.1 lists the major fault-tolerant software techniques in use today. In the discussion that follows, each fault-tolerant software technique is described with references to how it fits the general process as defined in Chapter 2.

3.1 Multi-Version Software

Multi-version software is any fault-tolerant software technique in which two or more alternate versions are implemented, executed and the results compared using some form of a decision algorithm. The goal is to develop these alternate versions such that software faults that may exist in one version are not contained in the other version(s) and the decision algorithm determines the correct value from among the alternate versions in the use of faults.

Methods to produce these alternate versions have been referred to as independent, diverse, dissimilar, or distinct design. These terms are used collectively to refer to the prominently supported method of producing alternate versions by using separate programming teams. Another suggested method is to explicitly make the versions different by examining them and forcing differences into the versions. Whatever means are used to produce the alternate versions, the common goal is to have distinct versions of software such that the probability of faults occurring simultaneously is small and that faults are distinguishable when the results of executing the multi-versions are compared against each other.

The comparison function executes as decision algorithm once it has received results from each version. The decision algorithm selects an answer or signals that it cannot determine an answer. This decision algorithm and the development of the alternate versions constitutes the primary error detection method. Damage assessment assumes the damage is limited to the encapsulation of the individual software versions. Faulted software components are masked so that faults are confined within the module in which they occur. This way the occurrence of a fault(s) is transparent to the outside environment. Fault recovery on the faulted component may or may not be attempted. Figure 3-1 depicts the above definition of multi-version software.

TABLE 3.1 CATEGORIZATION OF FAULT-TOLERANT SOFTWARE TECHNIQUES

Multi-Version Software

- N-Version Program
- Cranfield Algorithm for Fault-Tolerance (CRAFT) Food-taster
- Distinct and Dissimilar Software

Recovery Blocks

- Deadline Mechanism
- Dissimilar Backup Software

Exception Handlers

- Hardened Kernel
- Robust Data Structures and Audit Routines
- Run-Time Assertions*

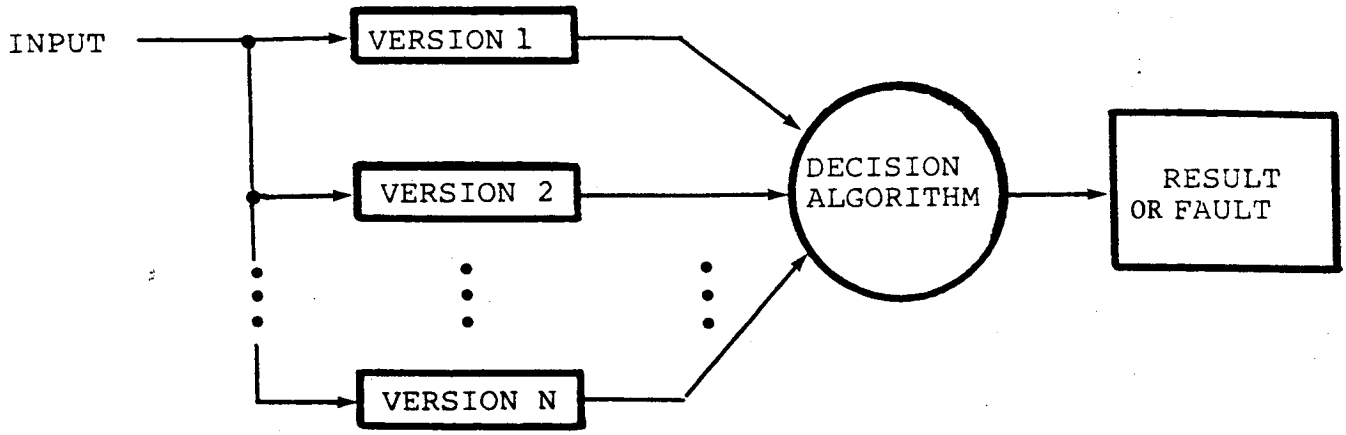
Hybrid Multi-Version Software & Recovery Block Techniques

- Tandem
- Consensus Recovery Blocks

---

* Not a complete fault-tolerant software technique as it only detects errors.

FIGURE 3-1 MULTI-VERSION SOFTWARE



3.1.1 N-Version Programming

N-Version programming is one of the earliest organized methods to introduce redundancy into software and is today one of the more developed techniques. Conceptually N-Version programming is analogous to static redundancy for hardware fault-tolerance. $N \geq 2$ functionally equivalent, independently generated versions are provided input from a system supervisory program called a driver. The driver executes a comparison algorithm on the versions' results. [CHEN78a] A majority vote for $N \geq 3$ is a typical example of a comparison algorithm.

N-Version programming is supposed to reduce the probability of a common fault among the versions and thereby increase the overall software reliability. This improved reliability is accomplished by having the versions independently generated by N separate individuals or teams. It is preferable that these individuals or teams have diverse training and experience. To help increase the independence, it is recommended that different programming languages, algorithms, data structures, and implementation techniques be used in each version. [AVIZ82]

Usually, independent programming teams are given a common specification. Use of a single common specification implies that truly independent versions are not produced because that errors in the specification will occur in all versions. Errors which are common to all N versions will not be detected in N-Version programming or in any fault-tolerant software technique. This is true in all fault-tolerant software and non fault-tolerant software techniques. However, Kelly has researched and experimented with N specifications written in different specification languages to attempt to overcome or at least reduce problems with a single specification. [KELL82] The specification

should be as complete and as unambiguous as possible (of course) and impose as few restrictions on the variety of implementations as possible. Additional information that must appear in the specification necessary for implementation of N-Version software includes: definition of cross-check points (cc-points), comparison vector (c-vectors) data structure, comparison status indicators (CS-indicators), synchronization mechanism, the comparison algorithm and the response to the possible outcomes.

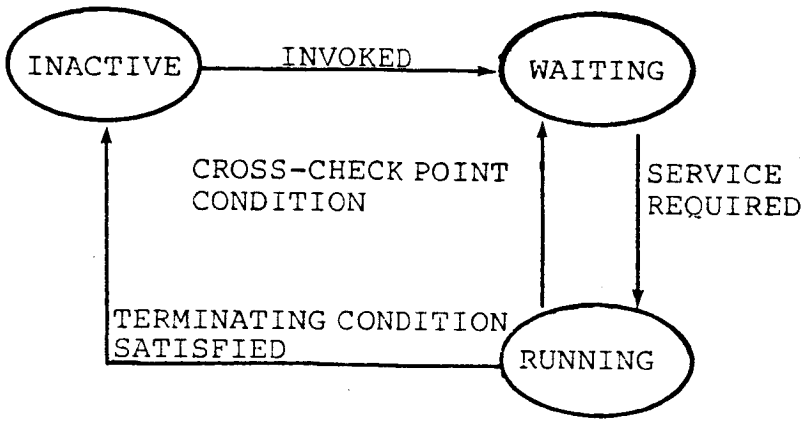
The cross-check points define when the versions will compare results and status. The most common cross-check points are common-input points, output-use points, and transaction-commit points. [MAKA82] This synchronization of the versions is a restriction on the implementation and may be another mitigating factor in the independence of versions. The comparison vector includes fields for the result data to be compared and comparison status indicators that are results of any additional assertions used within an individual version.

The driver controls synchronization activities of the versions. [CHEN78a] Initially, a version is in an inactive state. When invoked by the driver, it enters into a waiting state where it waits for a synchronization signal, representing a request for service from the driver. When this signal is received, it transfers into a running state. If any terminating condition is signaled by the status in the c-vectors, then the execution of this version is terminated and it returns to the inactive state. Otherwise, it generates a c-vector when a cc-point is satisfied. It then uses a synchronization signal to notify the driver that a c-vector is ready, and finally returns to the wait state. The state transitions for a version are illustrated in Figure 3-2.

The driver also handles the decision algorithm. The decision algorithm may be a majority vote for N>2, or some other strategy. For a voting scheme involving multiple correct values, the allowable range of discrepancy from each version, the data sensitivity of the algorithm, and the limitations of the hardware representations must be taken into account in order to develop a system discrepancy range for comparison of numerical results.

N-Version programming systems may execute the versions serially in a single processor (N-Version serial), or execute each version in parallel on N loosely-coupled processors (N-Version parallel). Pratt, Knight, and Georgory [PRAT83] make the observation that "damage assessment is handled by the assumption that damage will be limited to the versions in the minority when the vote is taken [then] ... to ensure this is true, the versions must be physically separated. Clearly, this is not easily achieved for parts of programs such as subroutines.

FIGURE 3-2 STATE TRANSITIONS OF A VERSION



In practice, this limits the application of N-Version programming to the system level and precludes its inclusion in technologies like software components."

For N-Version parallel, additional hardware communication channels must exist point-to-point with each processor each driver is to handle the decision algorithm. The communications channels must be able to rapidly transmit c-vectors for decision making by using a developed protocol.

3.1.2 CRAFTS (Foodtaster)

The Cranfield Algorithm for Fault-Tolerant Software (CRAFTS), or Foodtaster, as it is sometimes referred to, was originated by Morris and Shephard of the Cranfield Institute of Technology in England. It was proposed as an integrated hardware and software fault-tolerant scheme applicable only to real time systems that perform process control of a continuous output function. [MORR81] The following description of CRAFTS will concentrate on the fault-tolerant software aspects.

CRAFTS requires two versions of the software and multiple processors (for the purpose of discussion, three is assumed). There is one version of the software referred to as the basic program, which attempts to totally fulfill the functional specification, and an alternate of the basic program. (How this alternate is derived is discussed later.) Both software versions execute serially in each of the three processors, thus requiring double the processing throughout. The software operates on time-skewed data, executed by each of the two software versions in different processors. For example, in Figure 3-3, data sampled at N-1 is executed upon in processor 2 by the alternate and in

processor 3 by the basic program. This provides both physical and temporal redundancy.

Error detection is performed by comparing the basic program's and alternate's results calculated from the most recently sampled data shown as N in Figure 3-3. If the results agree within some tolerance, then CRAFTS assumes there is no software error and operation continues. If the results disagree, estimates of the basic program and alternate outputs (shown as B_N(EST), A_N(EST) in Figure 3-3) are made by extrapolating the past three results. It is the use of extrapolation that restricts the use of CRAFTS to continuous output functions.

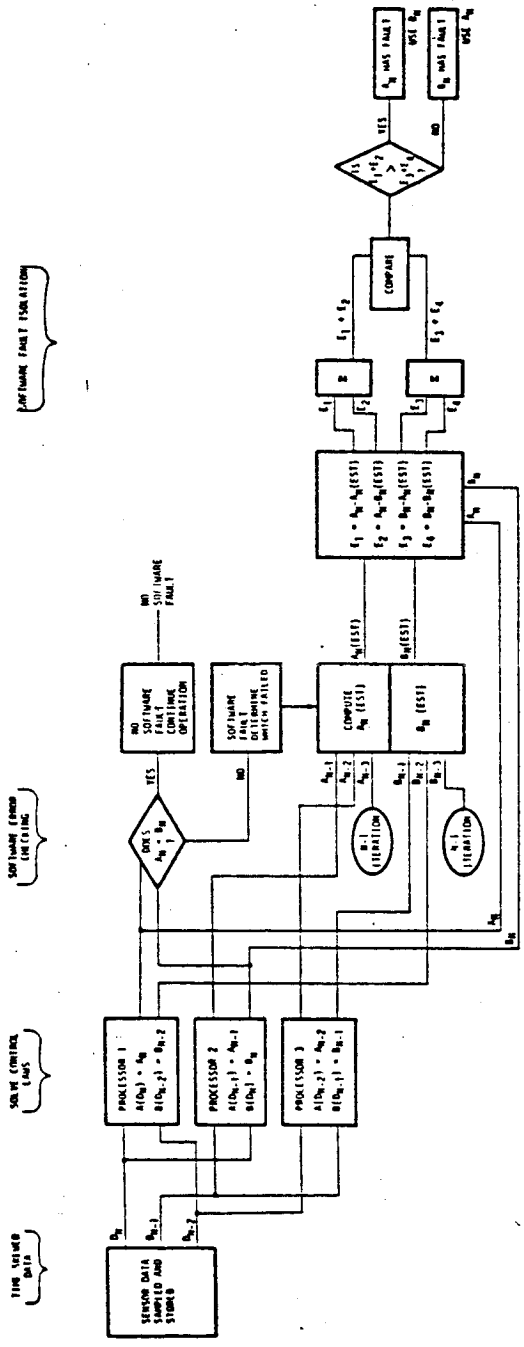
Damage assessment, error recovery and fault treatment are performed the same as in multi-version software. The comparison of the calculated results (B_N, A_N) with the estimated results (B_N(EST), A_N(EST)) determines which software program's result will be used. If the difference is too large a fault is signalled. It is unclear from [MOOR81] whether the decision to use a particular program's results is temporary or permanent.

Alternate program versions are generated utilizing the iterative nature of realtime systems. Morris shows how the output of the n+1 iteration can be given in terms of the n iteration output and n+1 and n inputs. This is referred to as "difference equations." Logic equations and conditional branches can be made different by applying DeMorgan's Theorem and complementary logic respectively. These methods are offered as a means of producing independent software versions, although they only operate at the expression or program statement level. It is doubtful if algorithm differences will occur. It is suggested that the generation of the alternates could be automated in order to avoid the costs of n developments. While difference equations for arithmetic and trigonometric expressions (the latter suffering from accuracy problems) were produced, "it became apparent that the difference equation method produced unacceptably complex programs when applied in situations more complex than arithmetic and trigonometric expressions." [MORR81]

Recently, Milco International extended and refined the original CRAFT Foodtaster Method. At this time, this work is proprietary and no results have been published. Milco's approach does not appear to necessarily require interprocessor communications for fault detection, isolation and recovery. The method "seems only suitable for control applications as in the original Foodtaster". [SMYT84] The software versions are arithmetically derived to be different so that no correlated faults exist. The goal is to automate the dual software derivation process, which has been generalized beyond the original Foodtaster concept of difference equations. Fault detection and fault recovery are

performed by an "estimator." The estimator determines which values to accept upon detection of a fault.

FIGURE 3-3 CRAFTS FAULT-TOLERANT SOFTWARE OPERATION AT SAMPLE N



3.1.3 Distinct and Dissimilar Software

The third variant of multi-version programming is the use of distinct and dissimilar software. Dissimilar software has been used on an ad hoc basis for a number of years and has proven itself quite successful in shortening the testing and debugging stage of software development. Dissimilar software and distinct software are terms widely used in industry to refer to multi-version software systems which, to date, have utilized two alternate versions. The second version is used to detect errors through a comparison of results from the two versions.

3.2 Recovery Blocks

The second major technique shown in Table 3.1 is recovery blocks and its subcategories - deadline mechanism and dissimilar backup software. For convenience, we categorize any fault-tolerant software technique, which has the general form asserted by Randell, as recovery block method. [RAND75] This scheme for software fault-tolerance can be regarded as analogous to hardware fault-tolerant "stand-by sparing." As the system operates, checks are made on the acceptability of the results generated by each software component. Should one of these checks fail, a spare component is switched on to take the place of the faulty component. The spare component is, of course, not merely a copy of the main component. Rather it is of independent design, so that there can be the possibility that it can cope with the circumstances that caused the main component to fail. Of course, recovery blocks satisfy this general description. The deadline mechanism [CAMP79] and dissimilar backup software are also included here because of their similarity with recovery blocks. Each of these is described later in this chapter.

"A recovery block consists of a conventional block (i.e., software component) which is provided with a means of fault detection (an acceptance test) and zero or more stand-by spares (the additional alternates). A possible syntax for recovery blocks is as follows:

"<recovery block> ::= ensure <acceptance test> by

<primary alternate>

<other alternates> else error

<primary alternate> ::= <alternate>

<other alternates> ::= <empty> | <other alternates>

else by <alternate>

<alternate> ::= <statement list>

<acceptance test> ::= <logical expression> "[RAND75]

The following is based on [RAND75]. To perform a recovery block operation, the primary alternate (which corresponds to the block of the equivalent conventional program) is performed. An acceptance test is run to determine whether the alternate has performed acceptably. If the primary version fails to complete or fails the acceptance test, the system state is restored to that current just before entry into the primary version and an alternate version is performed. If the primary version passes the acceptance test, all alternates are ignored and the statement following the recovery block is executed. If the final alternate does not pass the acceptance test, the entire recovery block is considered to have failed, so the block in which it is embedded fails to complete. Recovery is then attempted at that level.

All fault recovery is accomplished by automatic reversal to a previously established recovery point. When a process has accepted the results of a recovery block, recovery can only take the form of a more global process reversal to the beginning of a recovery block which has not gone through the acceptance test.

The valve of the recovery block scheme depends on the practicality of producing useful acceptance tests and alternates and on the cost of providing means for resetting the system state.

The acceptance test ensures that the operation performed by the recovery block satisfies the program which invoked the block. Therefore, the test must be performed by references to the variables accessible to that program rather than variables local to the recovery block. The surrounding program can continue with any of the possible results of the operation. The acceptance test confirms that the results are within the range of acceptability, disregarding which alternate can generate them.

The test does not have to be a formal check on the "correctness" of the operation performed by the recovery block. Instead, the designer decides upon the stringency of the test. Ideally, the test should ensure that the recovery block meets all specifications depended upon by the program test that calls it.

Some acceptance test failures may not be legitimate because of design inadequacies within the test itself; in effect, false alarms. The acceptance test may even suffer an error during execution and fail to complete. Such errors are treated as failures in the enclosing block. (These failures should be rare, since acceptance tests are intended to be simpler that the alternates they check.)

In evaluating an acceptance test, any modified non-local variables must also be available in their original form because of the possible need to reset the system state. To give the acceptance test more strength and efficiency, the test should be able to access these non-local variables in either their original or modified value. An additional facility available in an acceptance test should be a means to determine whether any of the modified variables have not been accessed within the test.

"The primary alternate is the one which is intended to be used normally to perform the desired operation." [RAND75] Other alternates will perform the operation in some other manner. Although they may perform less economically they should be simpler. So as long as one of the alternates succeeds, the operation will have been completed.

Since maintenance actions to enhance a software system will always be required, a method of overcoming any unreliability of the new version has been proposed [MELL83]: retain older versions of the software component as secondary alternates and make the new version a primary alternate. This way, the enhancements of the new version will be performed, but if a fault is encountered, availability is maintained by use of the older versions.

By making system state resetting fully automatic, programmers are shielded from the problems of resetting the system in fault recovery. No special restrictions exist and no special programming conventions must be followed. To be specific, the task of explicit preservation of restart information can be avoided. In this way, the recovery block structure provides a framework enabling extra program text for error detection and recovery action to be added to a conventional program. Even though the program will increase in size, its reliability will increase also.

Since a process is always backed up to the state it had reached just before entry to the primary alternate, only modified, non-local variables have to be reset. "This

mechanism detects, at run-time, assignments to non-local variables, recognizing when an assignment to a non-local variable is the first to have been made to that variable within the current alternate. Thus, precisely sufficient information can be preserved." [RAND75]

Melliar-Smith [MELL83] lists the assumptions upon which recovery blocks are based. If these assumptions were completely justified, then recovery blocks would be able to produce any desired level of reliability. However, these assumptions are rarely unflawed leaving opportunities for software unreliability. For recovery blocks, the assumptions are: a correct specification, recognition of faults, faults manifest themselves within a recovery region, the independence between alternate blocks, and the independence of N alternate blocks from the acceptance tests.

3.2.1 Deadline Mechanisms

The deadline mechanism [CAMP79] adopts the same structure as the recovery block but it performs a specific type of fault detection assertion. The deadline mechanism has scheduling mechanisms that include hardware timers to ensure timely responses of the software component.

The deadline mechanism associates two algorithms (i.e., alternates) with each software component. The primary alternate produces a better quality service than the other alternate. "Better quality service" means it fully implements the software component's specification. The alternate is a simpler, more deterministic algorithm which will produce a result in a known amount of time. This "simpler" alternate will generally only provide partial functionality of the software component, thereby reducing it to a degraded mode, at least for the current computational iteration.

The deadline mechanism itself is a centralized scheduler and supervisor which replaces the acceptance test of the recovery block technique. It ensures that either the primary or alternate completes before a time deadline.

The deadline mechanism was proposed to tolerate faults that manifested themselves as timing faults. The mechanism can be used to complement the fault-tolerant capabilities of recovery blocks. This aspect has been suggested by Hecht [HECH76] as necessary for real-time applications.

3.2.2 Dissimilar Backup Software

In its complete form, dissimilar software has three main components: a fully functional primary software system, run-time assertions embedded within the primary software system, and a dissimilar, secondary software system that performs

only the essential subset of the requirements. The primary software system executes the system function during fault-free operation. Run-time assertions are used to detect faults in the operation of the primary software system. Upon detection of a software fault, the primary is aborted and the secondary software system begins operation.

3.3 Exception Handlers

Traditionally, exception handling involved a combination of a hardware exception (fault) monitor and a software routine in which the function was to compensate for the fault. Examples of these exceptions are divide-by-zero and arithmetic overflow. The fault detection was application independent while the software handler was application dependent. (One typical form of fault recovery was to restart the system with the only fault treatment being that the function was no longer scheduled for future execution).

Exception handling today encompasses a variety of techniques and methods to detect conditions in which the computer has reached an improper state, and to correct this state. These include use of hardened kernels to protect or encapsulate errors, robust data structures which protect the data integrity and run-time assertions.

Cristian [CRIS82a and CRIS82b] has developed a model of exception handlers in a hierarchy of modules using the concept of data abstraction. For each data abstraction, exceptions have to be specified as a response to run-time attempts to violate its inherent invariant properties. These anticipated faults can be handled by forward fault recovery techniques. No specified fault treatment is offered. Unanticipated faults, i.e., design faults, can be handled by a default exception handler using automatic backward recovery. Cristian shows how recovery blocks can be modeled by his default exception handler. [CRIS82a]

3.3.1 Hardened Kernel

Hardened kernel is a term used to describe a software organization aimed at minimizing the complexity of the system software by limiting its operational requirements to essential functions. [RANE83] Essential functions are encapsulated as the kernel of the software system. All additional software that completes the system specification interface to this kernel. Run-time assertions monitor the performance of the entire software system. If a software fault is detected in the additional software, the function is aborted and not used again. The assumption is that the kernel is the minimum software required, simple, and therefore more reliable, than the entire software system. Also,

it is executed constantly and therefore very few software faults should remain.

3.3.2 Robust Data Structures and Audit Routines

Robust data structures are techniques for providing a self-identifying structure to the necessary information so that if an error occurs, the information structure can be constructed. A robust data structure is a data structure which contains redundant structural information. If erroneous changes occur, then errors may be detected and possibly corrected. Error detection may be accomplished concurrently by the software component accessing the data structure or by periodically executed audit routines (which checks the consistency among the redundant information). If an error is detected, it may be repaired using forward recovery. The redundant information is used to reconstruct consistent structure. Fault treatment is not directly addressed by the robust data structure method of software fault-tolerance.

3.3.3 Run-Time Assertions

Run-time assertions are a unified, descriptive way of viewing all software fault detection mechanisms. Table 3.2 provides a list of run-time assertions.

The acceptance test in recovery blocks is an assertion about the results of a software component. It is assumed that the acceptance test is more reliable than the software component it is monitoring. The voter of a multi-version software asserts that the majority will be correct. Reasonableness checks use specification knowledge to find parameters out of range, faulty rate of change or invariant unique mathematical relationships that are not maintained. A watchdog timer asserts that if a software component does not complete within a fixed time then a software fault has occurred. Capability architectures and tagged memory assert that a component should not reference beyond its initially set memory region or interact with parameters of unlike type, respectively. Control flow checks assert that program sequencing should not go through any previously undefined paths. [YAU80]

3.4 Hybrid N-Version and Recovery Block Methods

None of the current fault-tolerant software completely solves the software reliability problem. Multi-version and recovery blocks have received most attention because of generality. A common problem is producing independent alternate versions of software components such that correlated faults are either eliminated or reduced to an accept-

TABLE 3.2 RUN-TIME ASSERTIONS

Acceptance Test

Voter

Exception Monitor

Reasonableness Checks (range, rate of change, unique mathematical relationships)

Watchdog Timer

Access Control

- Capability Architecture
- Tagged Memory

Sequence Monitoring (Path Checking)

Analytical Redundancy

Observers

ably low level. Multi-version software and recovery blocks also have their own individual problems due to their structural implementations. Hybrid techniques are proposed to exploit those advantages and avoid those disadvantages that are inherent in multi-version software and recovery blocks.

3.4.1 Tandem

Tandem [SONE81] is a method suggested by Soneriu. It has a fault detection conceptually similar to N-Version programming, and the facilities for backward recovery like the recovery block. Figure 3-4 shows the functional flow of the tandem technique.

Tandem requires two separate processors. Initially, a common recovery point is set on only one processor or different recovery points can be set if a recursive cache is available. Two alternate versions of the software component are executed on the different processors. A validation test compares the results. If the results are the same, then the recovery point is purged and the output becomes available for the next operation. If the results differ, however, recovery is accomplished by rolling back to the system state saved before execution of the current operation. Fault treatment is handled by selecting another set of alternate

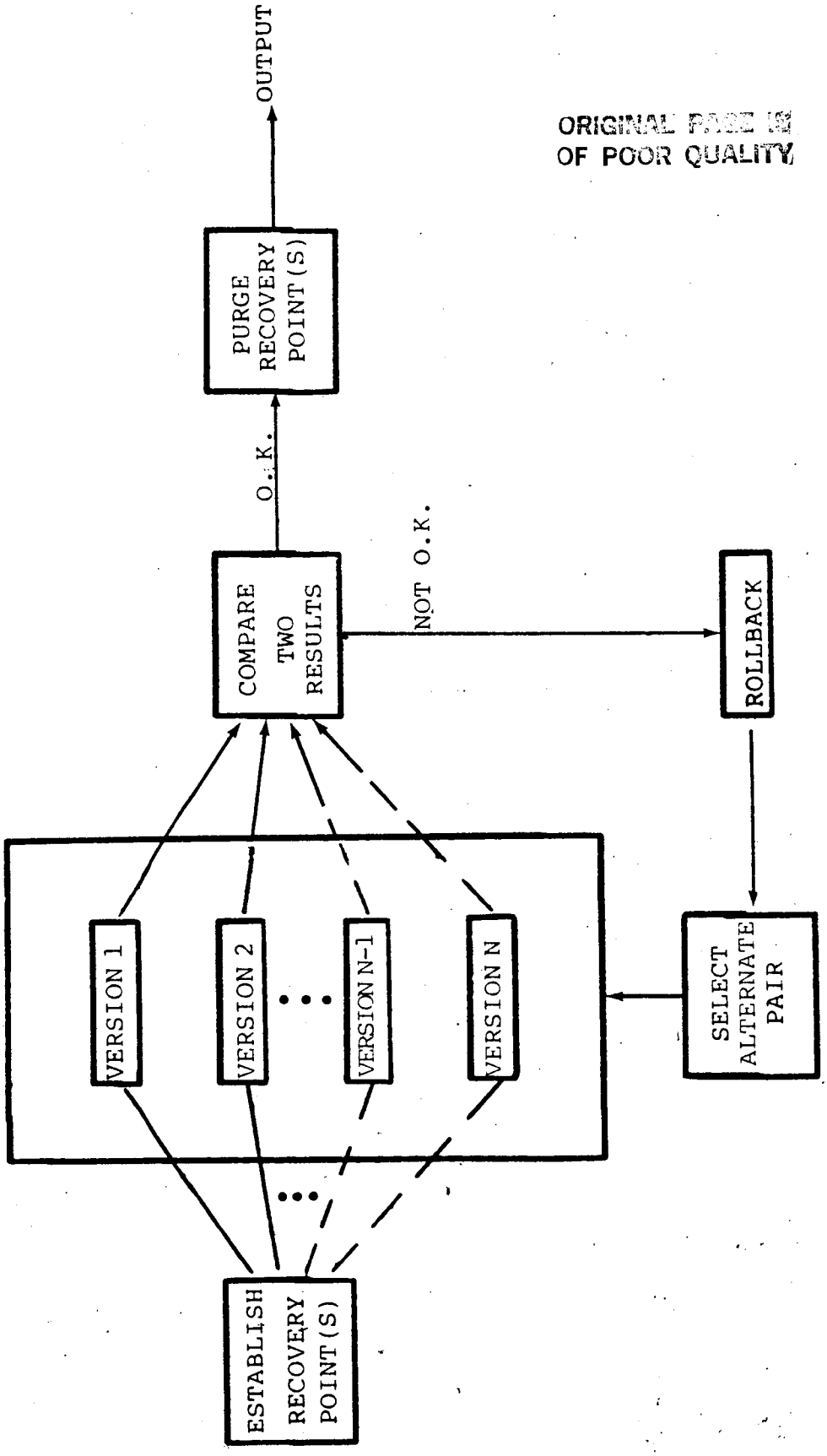
versions. This election is repeated until all combinations of two alternates fail to give an acceptable result. If this occurs, the Tandem operation indicates a failure of that software component. The advantages of this technique over multi-version programming is that less resources are required to have a fail-operational system. Tandem requires only two processes while a multi-version software unit would require at least three processors. The advantage this technique has over recovery blocks is that the fault detection mechanism is performed by comparison which is simpler, executes in less time and is considered a more general error detection mechanism as compared to the acceptance test. Tandem does have some disadvantages when compared with recovery blocks. The reliability of a simple comparison, in the face of the potential for correlated errors, has not been shown to be higher than an acceptance test criteria.

3.4.2 Consensus Recovery Block

The consensus recovery block method [SCOT83b] attempts to lessen the importance of the acceptance tests in recovery blocks. The acceptance tests have been identified as the most crucial component in recovery blocks, yet there is no general methodology to design and analyze the effectiveness of acceptance tests. [MELL83 and HECH79] Also, the consensus recovery block method addresses the problem of the discrepancy range for an exact comparison of alternate versions, as well as multiple correct answers (e.g., best vs. first fit memory allocation), as in multi-version software.

As shown in Figure 3-5, the consensus recovery block method requires the development of n independent versions of a program, an acceptance test and a voting procedure. Initially, all versions execute and submit their outputs to a voting procedure. Since it is assumed that there are no common faults, if to more versions agree on one output, that output is designated as correct. If there is no agreement, that is, the versions supply incorrect outputs or multiple correct outputs, then a modified recovery block is entered. Each version is examined sequentially in a predetermined order. The output of the "best" version is subjected to the acceptance test. If that output is judged acceptable, it is treated as a correct output, and system execution continues. If, on the other hand, the output is not accepted, the next best version's output is subjected to the acceptance test. This process continues until an acceptable output is found, or the n outputs are exhausted. Notice that there is no requirement for input state recovery since all versions execute in a parallel fashion as in an N-Version programming system. This technique might be more aptly termed consensus multi-version software.

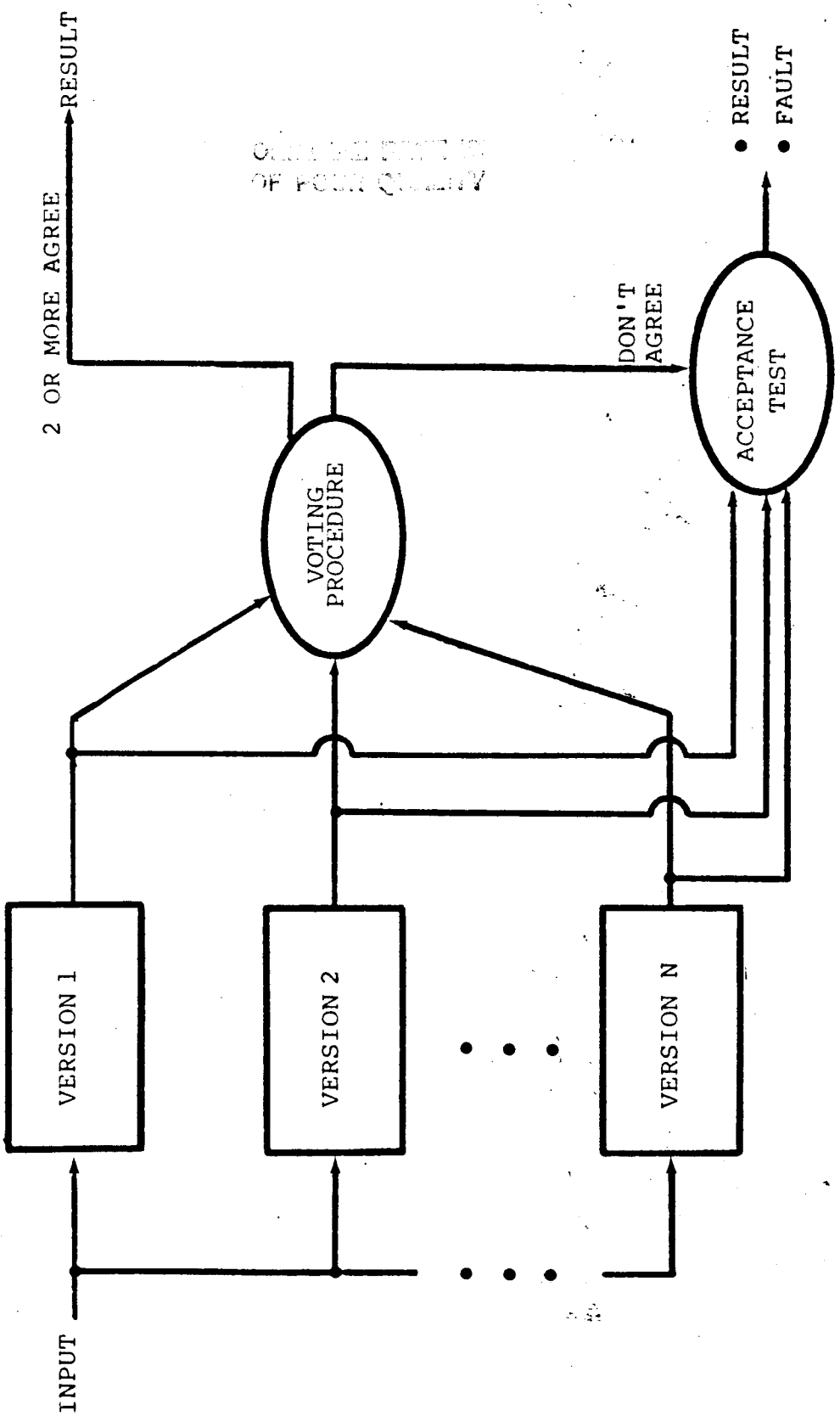
FIGURE 3-4 TANDEM - A HYBRID TECHNIQUE



3.5 Summary of Techniques

In summary, it was found that the techniques of fault-tolerant software have been developed conceptually from the earlier disciplines of hardware fault-tolerance and common programming techniques that had been used for a number of years in reliable systems on an unstructured or ad hoc basis. The categorization of the techniques into the four broad areas shown in Table 3.1 is not a final division of this technology; all techniques appear to be part of the more generalized model that is presented in Chapter 2. The divisions presented are for convenience only. Conceptually, it was found that the technology is continuing to evolve with newer techniques or methods being developed as combinations or refinements of older methods. However, the basic four steps outlined in the generalized model appear as the fundamental framework for all work.

FIGURE 3-5 CONSENSUS RECOVERY BLOCKS - A HYBRID TECHNIQUE



4.0 Current Experience with Fault-Tolerant Software

In this chapter the experience that has been gained using fault-tolerant software techniques and current research efforts that are underway or planned is reviewed. This information was accumulated through a review of published results and, since published results tend to be a year or so old, with telephone and personal interviews of key researchers. The thrust of this effort was to cover all the fault-tolerant software techniques that have been proposed and obtain several pertinent examples of the experiences with each technique.

4.1 Experience with Multi-Version Programming

It was found that there is an extensive amount of practical and research experience in multi-version programming, although only one implementation of a complete N-Version programming system could be found in operation. Several industries including commercial aircraft, transit systems and nuclear power plants have utilized dissimilar software in fail-safe applications.

4.1.1 Space Shuttle Experience

The NASA Space Transportation System, better known as the space shuttle, uses multi-version software to tolerate software faults in mission-critical functions. Computing resources are provided by four primary computers which are composed of identical hardware and software. A single backup computer, which has an alternate version of the mission critical functions (i.e. flight control, guidance and navigation), operates in parallel with the primary computers. The primary computer software was developed at IBM. The backup computer software was developed by Rockwell, Draper Laboratory, and Intermetrics. The software is being independently maintained by Rockwell. Error detection is accomplished primarily by the crew, but also by the backup computer and the primary computers themselves. Damage assessment and error recovery are ignored. Fault treatment is performed by the crew when they initiate a switchover from the primary computer control to the backup computer control. [TROY84]

During software development, numerous software faults and requirement faults were uncovered because of the existence of the alternate (backup) version. No faults have been detected during the operation of the shuttle. However, prior to launch of the first flight, a fault in the synchronization between the primary and backup existed. If the backup computer had not been in the system, this fault would not have occurred. In this example, fault-tolerant software

reduced availability because of the added complexity, but reliability was not decreased. The system performed according to specifications when the computers disagreed. On the other hand without the backup computer, there might not have been sufficient confidence in the primary computer's ability to fly the shuttle.

4.1.2 Airbus Industries Experience

The Airbus Industries A310, a production commercial aircraft put into service in April 1983, has a slat and flap control system developed by Marconi Avionics that uses a dissimilar architecture [MART82]. The slat and flap control system is totally fly-by-wire and executes a fail-passive function. Fail-passive is a type of fail-safe operation where the safe state for the system is to remain "frozen" in the previous state.

The use of dissimilar software was driven by four major factors (not necessarily in the presented order):

- Certification authorities in England, France, and Germany recommended consideration of "dissimilar monitoring", which is a historic precedent in developing safe systems. This was suggested because of uncertainty as to how to analyze the safety of a digital system;
- Marconi desired to propose to the customer a program that was financially and technically advantageous compared to the opposition;
- Marconi itself desired a method to reduce the excessive cost of producing flight-crucial software;
- The unique requirements of the system, i.e., fail-passive, lent itself to a dual-dissimilar architecture.

Marconi's experience was that it was able to reduce the costs of software testing by testing the versions against each other. There was no significant cost improvement, however, because Marconi did more extensive testing than might have been necessary, since dual-dissimilar software was new to Marconi and the certification authorities. Overall, the certification process, a very costly process, was easier to conduct than if a single version of software had been produced.

David Martin, a Marconi engineer involved in the development of the slat and flap control system, believes that the use of dissimilar software will continue "but it's not the answer to everything. You are not going to use it for every system. ... It's another weapon in the armory

[against software unreliability]". He also doubted the use of fail-operational dissimilar software if three versions are required. He felt that would probably be more costly than if a single version were extensively tested to achieve same level of confidence in the software system. [MART84a]

4.1.3 Boeing 737-300 Experience

The flight-control system for the Boeing 737-300 aircraft is a dual-dissimilar system developed by the Sperry Corporation [WILL83]. One version performs all functions, while the second version performs only critical functions. The reliability of the non-critical software is accepted with a single version of software. The certification authorities' view is that if it is not evident that the redundant channels are independent, then exhaustive (more extensive) testing would have to be performed. The two versions were programmed by independent programming teams using a common specification. The probability of similar software faults was considered essentially eliminated. Error discrepancies in the algorithms is well understood so that comparison tolerances can be accurately set. During software testing, the versions were compared at limited but specific points (e.g., module integration tests, hardware and software integration tests) because of the concern that testing against each other reduces the independence of the two versions. The dissimilar architecture resulted in a cost savings during the verification phases because of the certification authorities' viewpoint.

John Williams, a Sperry engineer involved in the development of this system, believes that fault-tolerant software can be used in fail-operational systems and will appear in future systems. One big reason for its incorporation is because of the cost savings in reliability and safety analyses. [WILL84]

4.1.4 Boeing 757 Experience

Digital control systems in the Boeing 757/767 aircraft have examples of both similar and dissimilar architectures. Two systems, the yaw damper and the stabilizer trim systems, were implemented using dual-dissimilar architectures. [MCWH84] Both are fail-passive systems.

The reason dual-dissimilar was chosen was because of a distrust of digital systems by the certification authorities who viewed the use of dissimilar architectures favorably. Boeing had hoped to reduce some of the verification and validation effort by using dissimilar architectures. However, in this case the certification authorities required extensive validation of each channel. This doubled Boeing's

analysis efforts and made the system more costly than the estimated cost of a single version software system. Another reason for the selection of a dual-dissimilar architecture was the requirement for high integrity (reliability) and fail-passive operation.

The two development teams were separated but the software versions were "made" to be more distinct. This was accomplished, for example, by having one version use a table look-up while the other used polynomials or, using DeMorgan's Theorem, to change logical expressions.

The service records of both systems are extremely good. To date, there have been no system faults and only a few nuisance disconnects. [MCWH84]

4.1.5 AECL of Canada Experience

Atomic Energy of Canada, Limited (AECL) has designed two dissimlar shutdown systems and placed one of the system types into service for nuclear power plants. [GILB84] The system implements two dissimilar channels each of which is triplex. Dissimilarity is the "cornerstone" of the regulatory requirements. Independent programmers developed the programs. The service experience of the one system type has been that two software bugs were found during final system validation and one bug was detected in actual operation. This bug was a round-off error in one version that was detected by the faulted version itself before it was detected by the other version.

Ray Gilbert at AECL stated that he believes that dissimilar software would not be used if it were not required by the regulatory agencies. He believes that as the complexity increases, the development cycle to produce separate versions becomes counter-productive to software reliability. Gilbert believes "It is better to produce one version and make it as simple as possible and thoroughly test it than to count on different people to program differently". [GILB84]

4.1.6 Other Multi-Version Software Experiences

Dual dissimilar software was used to implement a reactor shutdown system at the Halden nuclear reactor. Done in conjunction with the Finnish Technical Research Centre, two different designs were written in two different software languages. They experienced specification ambiguities and one software error common to both designs. [TAYL81]

The LM Ericsson Company in Sweden has developed several fail-safe systems using dissimilar software for the railroad industry. These systems include an interlocking system in Gotegorg, train warning and train stopping system in Taiwan,

and automatic train control and speed supervision systems for Scandinavian and Norwegian state railways. Other means of insuring reliability of software, e.g., manual verification, testing, and simulation, were felt to be inadequate for a safety-related function. Therefore, dual dissimilar software was produced by two independent programming teams. Specification faults were detected, especially where the specification did not allow for unexpected situations. [VONL79]

Westinghouse Transportation Division has developed two generations of automatic train protection systems for the Metro in Sao Paulo, Brazil. These systems utilized similar dual hardware, with dissimilar software written in assembly language. During the software development, software faults were discovered by comparing the results of the two different versions. Pierre Zuber, a Westinghouse software engineer involved in these projects, believes there are two main obstacles in the way of using fault-tolerant software: First, it is difficult to produce independent software versions because of the limited way to produce a result, especially for real-time systems. Second, there is added difficulty in maintaining more than one software version. [ZUBE84]

Using its variant of the CRAFT Foodtaster, Milco International has implemented a simulation of the B-1 bomber's flight control terrain-following system using dual hardware and dual software versions, i.e., both versions in both processors has purportedly been implemented. [SMYT84] Current results obtained from the simulation of the B-1's terrain-following system had "100 percent fault detection, and recovery for injected software faults". These results were obtained for both-time skewed and non-skewed data. Therefore, the usefulness of time-skewing to detect software faults requires further investigation. [SMYT84]

4.1.7 Current Research in Multi-Version Programming

Most current research in multi-version software is being conducted at universities. The University of California, Los Angeles (UCLA) has been a leading promoter and research bed in multi-version software. UCLA is currently developing an experimental laboratory using twenty VAX-11/750's which will operate on a local area network. A special operating system has been developed, called LOCUS, to support N-Version software experiments. Currently, efforts have been focused on getting the laboratory operational. Another research topic being pursued is the classification of software faults. [AVIZ83]

Several experiments have been conducted at UCLA to investigate the feasibility and utility of N-Version software. Experiments conducted in 1976 and 1977 were designed

to study the applicability to problems and ease of implementing N-Version programs and collect qualitative and quantitative data on effectiveness of 3-version software.

MESS (Mini Text Editing System) was a program assignment for the graduate seminar course E226Z, offered at UCLA in the spring quarter of 1976. A total of 27 programs were available for the study. All but two were found to contain design faults. From the experience and the results of the MESS experiment, the following conclusions were reached: the methodology used to implement N-Version programming is relatively simple and can be generalized to other similar applications; the results attained from executing 3-version programs are encouraging. The effectiveness of 3-version programming seems to warrant further investigation. The 3-version redundancy was successfully applied at subroutine (module) level, rather than at complete program level. This result shows that selective application of N-Version redundancy to certain critical parts of longer programs can be a practical alternative. [AVIZ77 and CHEN78b]

RATE (Region Approximation and Temperature Estimation) was a program for computing dynamic changes of temperatures at discrete points in a particular region of a plain. The RATE problem was given as a programming assignment for the graduate seminar course, E226Z, offered at UCLA in the spring quarter of 1977. Three different algorithms were specified to accomplish the function. One algorithm was given to each programming team. Out of a pool of 16 programs, the four best plus three written by Chen and Avizienis, formed the seven programs for the experiment. The seven programs were instrumented, grouped into 12 combinations, and tested with 32 test cases each. Of the total 384 cases tested:

- 290 cases contained no bad versions,
- 71 cases contained one bad version,
- 18 cases contained two bad versions and,
- 5 cases contained three bad versions.

Naturally, the 290 cases of three good versions generated acceptable results, and the 23 cases containing two or three bad versions generated unacceptable results.

The results obtained from the MESS and the RATE experiments are of mixed nature. There are several encouraging points: the methodology for implementing N-Version programming was relatively simple and can be generalized to other similar applications; in some cases, 3-version programming has been effective in preventing failure due to defects localized in one version of code; and N-Version programming can be a practical approach if it is selectively applied.

On the other hand, there are some negative points: in the environment of some operating systems, certain implementation defects of a version may cause its associated 3-version program to be aborted by the operating system; and if missing program functions are the predominant software defects, then N-Version programming may not be an effective approach. [CHEN78a]

The primary aim of later experiments by Kelly [KELL82] and Avizienis [AVIZ82] was to investigate software specification techniques and how this affected N-Version programs. An airport scheduler specification was written in PDL, OBJ, and English. The 18 individual versions' stand-alone tests of 100 test cases resulted in a range from 35 percent to 95 percent yielding good results. All triad combinations of the 18 versions were executed in an N-Version organization. Cases of all three versions producing an incorrect result occurred .1 percent of the time, while two bad versions out-voting the good versions occurred 2.4 percent of the time. These cases would result in total failure of the system. Two good versions correctly out-voting one bad version occurred in 27.1 percent of N cases. This would result in the function to be available when a single version may have given a faulty result. The conclusion was that N-Version software had an increase in reliability over individual version software and the increase was substantial. [KELL82] John Knight at the University of Virginia is currently starting an experiment to validate the fundamental principle of multi-version software, that is, that the programs be independent of each other. He will be starting with versions of a program which must pass 200 randomly generated tests to be used in the experiment. Previous similar experiments (Kelly [KELL82] and McAllister [MCAL83]) did not require that the individual version's reliability be that good. Knight expects results to be available in the summer of 1984. [KNIG84]

David McAllister at North Carolina State University has been conducting experiments in correlated faults in multiple versions of a program. He found that their initial attempt to predict the reliability of N number of versions was too optimistic. He wishes to repeat the experiment using programs with individually higher reliability. Nelson's reliability model was used to obtain the reliability of each version by executing 100 test cases. The reliability of the individual programs was as low as 17 percent. [MCAL83]

4.2 Experience with Recovery Blocks

As with multi-version programming, no practical example of a complete system built using recovery block method was found. The nearest to a realistic implementation is the work currently being performed by the Computing Laboratory at the University of Newcastle-upon-Tyne in Great Britain

for the British Navy. The work is very recent and has not yet been published. This information was supplied by Dr. Thomas Anderson who acted as a consultant to this study. The application is a naval command and control system that was constructed to commercial standards. The application involves both concurrent processing as well as real-time processing constraints. The system is a fairly large application - approximately 8000 lines of code (50K bytes of machine code). Although the experiment is not complete, some of the results include a significant increase in reliability (approximately 2.5 times increase) and a 10% performance degradation due to overhead needed for recoverability. [ANDE84]

The Computing Laboratory at the University of Newcastle upon-Tyne has also conducted numerous other research projects in recovery blocks. Randell [RAND75, 78A, 78B, 78C, 79] proposed the formal structure of recovery blocks and has researched recovery in distributed systems. Anderson [ANDE76a, 81a, 78] is also extensively involved in computing system reliability research. Cristian [LEE78, 79] and Wood [WOOD80], are also all involved in research in recovery blocks.

In work for the Army, System Development Corporation experimented with distributed recovery blocks in a shared memory multi-processor used in a real-time closed loop control system. The system is designed to look at fault-tolerant software in missile tracking applications that are highly time and reliability critical. Results showed that recovery blocks provided fault-tolerance for selected real-time processes with low impact on CPU utilization and critical response times. [WELC83]

There has also been experience with deadline mechanisms. Campbell, et. al., [CAMP79] report the results of a simulation of the deadline mechanism. Wei [WEI81] discusses the application of deadline mechanism to two real-time systems, the Annular Suspension Pointing System (ASPS), which is a computer system controlling a platform on the Space Shuttle, and on the widely used satellite based on Multimission Modular Spacecraft. Both systems' software was reimplemented using Path Pascal and deadline mechanisms. Simulations of the Path Pascal version verified the applicability of the deadline mechanism in real-time applications.

The REBUS (Resident Backup Software) is an example of dissimilar backup software. It is being implemented in the F-8 digital fly-by-wire program being conducted by Draper Laboratory for NASA/Dryden.

4.3 Experience with Exception Handlers

Boi, et. al., [BOI81] discuss the exception handling in the ISAURE system, a small business application oriented system. The system was modular and hierarchically organized. Exception handling attempted to mask the faults at the level of occurrence. If masking could not be accomplished, the exception was propagated upward in the hierarchy to be handled at that level. Willett, et al, [WILL82] focus on the design of recovery strategies, forward and backward, that have been adopted in the Bell Telephone No. 4 Electronic Simulating System (ESS). These recovery strategies dealt with program anomalies and bad data. A total of 38 different recovery strategies have been developed, 15 of which serve the common need of several units, while 23 serve particular units or unique fault situations. It was unclear how many of these recovery strategies were necessary to handle hardware faults. The recovery of data is performed by audit programs which reconstruct (forward recovery) the data from associated information or by various levels of reinitialization (backward recovery). Reinitialization is faster but has a larger negative system impact, i.e., more calls may be disrupted. Software fault detection (exception monitor) checks job scheduling and sequencing for frequency and execution time, and watchdog timers perform sanity checks.

The Rockwell/WESCOM 580 is a family of multiprocessor controlled telephone switching system. The expected cost associated with multi-version software or recovery blocks is not justified for a telephone switching system. [DE81] However, the high reliability requirement of these systems resulted in the need for extensive error detection and recovery. Error detection mechanisms include a watchdog timer, memory access monitoring, software range and index checks, and periodically executed audit routines. When an error is detected, forward error recovery is attempted first in all cases except watchdog timer detected errors, which involves backward recovery. Fault treatment is performed off-line.

With regard to the hardened kernels, Boeing has developed the MOSS (Minimum Operational Software Set) concept and has used it in a C-135 flight control system simulation and Compass Cope remotely piloted vehicle project. They plan to continue research on this concept use in flight-critical applications. [MART84b]

The classic example of the use of robust data structures and audit routines is their application by Bell Laboratories in the data structures of their electronic switching system [BEUS69 and WILL82]. More recently, Black, Taylor, and Morgan [BLAC80 and 81] looked into the theoretical and practical use of robust data structures. They describe the addition of redundancy to data structures as a way to

improve a software system's ability to detect and correct faults in the control information. The approach is applicable to both forward and backward recovery techniques.

The Bell System's TSPS is a telephone line switching system. It implements redundant information in the data structures. A software audit program periodically checks the data structures. Upon detection of an error, forward recovery techniques correct the damage. While correcting damage losing some data is acceptable as long as the system can be kept operational. TSPS installations averaged between 10 and 100 actual corrections per day even after several years of operation. An automatic restart, i.e., backward recovery, occurred about once every two months. [CONN72]

4.4 Summary of Experience with Fault-Tolerant Software

As with any technology which is in the early stages of its evolution, examples of complete implementations fault-tolerant systems in the real world are limited. In fact, no true industrial instance was found where a complete fault-tolerant system was implemented (including the automatic correction). Many instances, were found where the techniques of fault-tolerant software have been used successfully in partial solutions. It was found that the developers who use fault-tolerant software techniques in their systems continue to thoroughly test their software so that the detection of additional errors has not necessarily occurred. No examples were found in which the last stage of the generalized model, fault treatment, was an integral portion of an operating system. However, it was found that portions of the techniques have been fairly widely used in error detection, and that the only examples of complete systems are in the laboratory or are too new to provide complete analysis.

There were, however, several significant results from the experience gained to date:

- First, there is no evidence that fault-tolerant software will degrade the reliability of the resulting system, and many of the projects surveyed reported that improved reliability (although not quantified) was achieved;

- Second, a trend was observed in which regulatory agencies are requiring software developers to use redundant software to improve the reliability in safety critical functions, and in which developers of systems using the dissimilar redundant software are being allowed to implement only critical subsets in the alternative, thus taking advantage of the inherent benefits of fault detection;

- Third, it was found that found that the empirical data needed to quantitatively evaluate the performance of the fault-tolerant software implementations has, with the exception of the research world, not been kept, and that, therefore, statements on improved reliability, cost benefit of fault-tolerance versus traditional software development, etc., could not be made beyond the personal observations of the individuals involved in the project.

5.0 Basic Principles and Design Approaches for Architecture

5.1 Introduction

This chapter addresses the impact of fault-tolerant software on the architecture and hardware design of future real time space computers. Its discussions are intended to complement those in the chapter on fault-tolerant software primitives, which emphasize the language support aspects of a fault-tolerant software architecture. In successive topics, we will discuss:

- Objectives,
- Architectural scope,
- Basic principles and design approaches.

These discussions are followed by chapters on fault-tolerance techniques and hardware implications, software fault-tolerant operating systems, and a summary of hardware mechanisms.

5.2 Objectives

System and application programmers currently use a wide variety of ad hoc techniques to protect systems from the harmful effects of software errors. In various forms, these techniques employ special types of redundancy, fault detection, damage assessment, fault treatment, and state recovery. Fault-tolerant software is an attempt to organize these techniques into a systematic design methodology. [ANDE81]. Several particular techniques have been described and several general purpose schemes, combining sets of more basic techniques, have been proposed. The most well-known are recovery blocks [RAND75 and HECH76] and N-Version Programming. [CHEN78a and KELL83] These techniques and schemes are evolving (in fact, those two schemes should be viewed as instances of a more general scheme), but the use of substantial and systematic redundancy in run-time software remains controversial, due to:

- The lack of field experience,
- The weakness of current cost-effectiveness models,
- Uncertainty about whether the kinds of faults addressed by present techniques have major significance.

Nevertheless, interest in systematic methods for fault-tolerance in software is increasing, and it is hoped that information on architectural organization and hardware costs

provided in this report will contribute to the development of fault-tolerant software as a general methodology that can be practiced as part of normal system development.

Since computer programming and architecture are evolving and interconnected arts, the discussion of computer hardware is presented in the framework of new concepts of computation that have been developed in recent years. This discussion will make continual reference to specific principles, mainly Encapsulation and Hierarchy. These principles are only partially reflected in current programming practice and architecture (e.g., the Ada and SmallTalk languages, the Intel iAPX 432 microprocessor and the Unix operating system), but it is expected that they will be increasingly adopted over the next five years.

In the same time period, it may be expected that next generation software concepts, which are presently being vigorously pursued in numerous research programs, will start to emerge rapidly, in order to support higher level programming and to exploit VLSI technology. These concepts, variously known as Applicative Programming [BACK78], Equational Programming, etc., are associated with new architectures that feature very high orders of parallel computation and a high degree of asynchrony [TREL82].

This discussion, therefore, attempts to distinguish hardware techniques that are applicable to current new architectures, within a methodological framework that reflects new and evolving trends in software and computer architecture. One of the findings is that the design approaches needed to support fault-tolerant software are compatible with modern trends in computer architecture. This encourages the view that support of software fault-tolerance need not add an undue amount of logical complexity in new computer system designs.

Quantitative evaluation of the proposed techniques was not attempted. The benefits are functions not only of the performance improvement provided by particular mechanisms, but also of their frequency of use. The costs are functions of how well a particular technique fits into a given architecture and logical realization, which is difficult to estimate outside the context of a particular machine. Also, the mechanisms suggested in many cases are only meant to suggest feasible approaches and not optimality. Many questions therefore remain open for further research.

5.3 Architectural Criteria and Scope

This section discusses criteria for choosing and evaluating architectural features, and scope of architectural types and levels. Recommendations include programmer flexibility, low performance burden, and close integration of

fault-tolerance and computational support functions. A wide scope of relevant architectural types is recognized, and the issue of applying fault-tolerance to the operating system itself is examined.

5.3.1 Criteria

We assume that the primary goal for the techniques to be considered is to improve software reliability in high-performance computers used in diverse, multifaceted, and mission-critical applications in aircraft and space vehicles. Such applications will include a full range of problem types and algorithms. Applications will cover a wide range of response times, sizes of data sets, and data volatility, i.e., expected lifetime of internal state information (quite different for control and data processing applications). The need for hardware fault-tolerance is assumed. We also assume that software fault-tolerance techniques will be used on an experimental basis to some extent, at least during the first few years of system life.

These environmental features imply the following criteria for the support of fault-tolerance in software:

- Programmers should be allowed flexible choice of software fault-tolerance techniques for application to particular program functions.

  Features added to support software fault-tolerance should have low impact on performance and low visibility to the programmer;

- Techniques employed for software fault-tolerance should be compatible with those employed for hardware fault-tolerance. To the greatest extent possible, they should not weaken the protection against hardware faults, they should enhance hardware fault diagnosis and recovery, and they should be jointly coordinated for optimum use of system resources to meet the reliability requirements of various mission functions.

These criteria have guided our selection of techniques and their evaluation in the study. They are, of course, ideals, and inevitably subject to compromise.

5.3.2 Architectural Scope

This section discusses the architectural issues covered in this chapter in three regards: architectural type, architectural level, and architectural scope of software fault-tolerance techniques.

The range of possible architectures for future aircraft and space applications is very large. Generic architectural types of this range include:

- Dedicated-Function Uniprocessing: A single computer, standing alone or slaved to another computer;
- Multi-task Uniprocessing: A single computer with full memory and I/O capabilities, serving multiple mission functions;
- Applicative Multiprocessing: A large set of simple processors sharing a multiported memory; distributed task control;
- Shared-Memory Multiprocessing: A set of processors, possibly with local memory, each with local program control; both physical and logical memory space are shared;
- Federated Multicomputing: A set of computers, possibly heterogeneous, each with its own memory and program control, possibly sharing a common file space; central system control is either fixed or movable among computers;
- Distributed Multicomputing: A set of computers, possibly heterogeneous; distributed system control.

The results of the study are relevant to all of these types. Lesser emphasis is given to uniprocessing, because of its limited significance in systems and also to applicative multiprocessing (i.e., it is not immediately relevant to the coming generation of air/space systems). It is assumed that some level of hardware fault-tolerance will be needed in all applications of interest, and we discuss the interaction of hardware and software fault-tolerance design issues.

In this wide range of system types there will be a great variety in detailed hardware characteristics, reflecting different design decisions about the extent of distribution of control, and the mechanisms for inter-module and inter-computer communication, the memory addressing concept, etc. This has limited the degree of detail available for describing the hardware implications of the support schemes considered.

Different definitions exist for the level of the term "architecture". One common definition narrows it to the set of machine functions seen by the system programmer. Broader definitions recognize it as a hierarchy of machine functions of different levels of responsibility or abstraction, and as a particular physical and logical organization of functional elements. These definitions allow more meaningful descrip-

tions of the functionality that links the hardware to the application programs and of the software and hardware design options that are available to the system designer.

We shall use the broader definitions of architectural level. We assume a hierarchical structuring of machine functions in which the top level is the functionality presented to application programs, and the bottom level is the functionality provided by "machine" language. We also recognize various possible functionalities and configurations of hardware, although we do not go very deeply into issues such as bus structure and protocol, and the physical partitioning of machine functions. We believe that this emphasis is appropriate for this study, given the wide range of possibilities for these design features in the set of architectures of concern. We definitely consider the internal and external functionality of the operating system.

Most discussions of software fault-tolerance have focused on its use in improving the reliability of application programs. It has been argued that such techniques are of less interest than system software, because the latter is stable and is subject to greater scrutiny, hence it may be expected to exhibit greater intrinsic reliability.

Two other considerations may have obscured the relevance of the architectural scope of software fault-tolerance to operating system software. First, most discussions of software fault-tolerance have assumed that approximate solutions are acceptable, which, with few exceptions, cannot be the case for system software (one cannot, for example, allocate one resource simultaneously to two consumers). Second, system software is typically very poorly structured, so it is difficult to apply any systematic design method to the operating system as a whole, as distinguished from particular functions.

Neither consideration is fundamental. In regard to approximate solutions: if needed, multiple versions or alternates of system software could be designed within the constraint of producing exactly the same results, so that the occurrence of any discrepancy implies a software error. Alternatively, one might design a hierarchy of system versions such that lower level versions would correctly apply system resources, but with less efficiency and hopefully with greater reliability (error detection could be provided by an acceptance test, as in [HECH82]). Many of the resources typically managed by an operating system may be replicated, with each replica managed by a separate software version of the operating system. With regard to system complexity: recent advances in operating systems (notably kernel-based designs) have introduced a high degree of design structure, so that upper levels of the system have the same relation to lower levels as application programs have to the operating system. This formal structure tends to

facilitate application of general software fault-tolerance techniques to operating system programs.

Special problems do pertain, (as will be discussed in the section on software fault-tolerant operating systems) but it appears that there are significant opportunities for beneficial application of fault-tolerant software to operating systems, given an appropriate system structure.

5.4 Basic Principles and Design Approaches

In applying fault-tolerance to a complex system, there is a danger that the new mechanisms may introduce additional sources of failure due to design and implementation errors. It is important, therefore that the new mechanisms be introduced in a way that preserves the integrity of a design, with minimum added complexity. It is very important that some unified principles of system organization be applied in the architectural design.

The concepts of encapsulation and hierarchy have evolved in recent years as extremely powerful and widely applicable principles of computer system design and programming. Briefly, encapsulation is an approach to organizing computational objects and activities in clusters with well-defined boundaries, and with central management of external interactions. Hierarchy is an approach to organizing computations into simple dependency relationships (specifically, a linear and unidirectional order of dependency, as in tree-type graphs).

The initial motivation for these principles was to simplify the process of making design decisions during system development. Subsequently, many other benefits have been derived, such as convenience of error containment and correction, secure data access and portability. Encapsulation and hierarchy are attractive for fault-tolerance because they offer ways to achieve simplicity and generality in the realization of particular fault-tolerance functions. They also offer powerful organizing approaches for integrating fault-tolerance into overall system design.

Examples of use of the encapsulation principle to enhance intrinsic reliability and fault-tolerance include:

- Organization of data and programs as uniform objects, with rigorous control of object interaction;
- Organization of sets of alternate program versions into fault-tolerant program modules (e.g., Recovery Blocks and N-Version program sets);

- Organization of consistent sets of recovery points for multiple processes (e.g., Conversations [NEUM80]);

- Organization of communications among distributed processes as atomic (indivisible) actions (e.g., methods for avoiding fragmentation of distributed processes using Remote Procedure Calls, as in [RUSH83]).;

- Organization of operating system functions into recoverable modules.

The hierarchy principle is coming into widespread use in system design. Notable examples are layered protocols for communications systems (e.g., the OSI module), structured operating systems (e.g., Multics and Unix), and structured software methodologies (e.g., HDM, Constantine, HOS, Jackson). Examples of use of the hierarchy principle to enhance reliability and fault-tolerance include:

- Organization of all software, both application and system type, into layers, with unidirectional dependencies among layers;

- Integration of service functions and fault-tolerance functions at each level;

- Use of nested recovery blocks to provide hierarchical recovery capability;

- Organization of operating system functions so that only a minimal set at the lowest level (a "kernel") need be exempted from fault-tolerance;

- Integration of global and local data and control in distributed processors.

Randell, Anderson, and Lee [RAND75, RAND75, and LEE83] have proposed the concept of recursive structure for fault-tolerant systems, based on a principle of recursive construction ("Fault-tolerant systems should be constructed out of generalized fault-tolerant component system" [RAND75]). They suggest a generalized component consisting of a package of functions for normal and abnormal activity, in which the latter functions are responsible for exception handling both within the component and at lower levels, and the former are able to signal need for exception handling to the abnormal functions at the same or next higher level of the hierarchy.

Continuing reference will be made to these principles in the detailed discussion of fault-tolerance techniques.

Use of kernel-based structure for operating systems offers a good method for applying software fault-tolerance techniques to the operating system, because most operating system functions are encapsulated into service modules. The main difference between application modules and operating system modules is that the domain and range of system module functions are the state of the computer, rather than the state of the system in which the computer is embedded, and the values of system module functions tend to be primarily logical rather than numerical (but that is not an exclusive property of the system modules).

In practice, the kernel itself is logical rather than simple (it must decide what process the hardware should pay attention to, provide communication between processes, etc.), and it must be very fast. It is natural, therefore, to avoid redundancy in the implementation of the kernel, i.e., to make it a trusted process, because it may be possible to verify its design and because redundancy would tend to reduce its speed. In principle, however, redundancy may be applied to the kernel and to all levels below it. Some protection might be obtained by periodic testing. These issues are discussed in greater detail in the chapter on Software Fault-Tolerant Operating System.

6.0 Fault-Tolerance Techniques and Hardware Implications

6.1 Introduction

Although the recovery blocks and N-Version schemes are the most well known proposals for software fault-tolerance, they should be seen as particular combinations of the more general software fault-tolerant schemes of error detection, damage assessment, error recovery, and fault treatment.

The schemes are themselves in the process of evolution, and with experience, it is likely that new combinations will evolve, and that future application and system programmers will choose their own combinations of basic functions. The discussion of hardware implications in this section, therefore, is aimed at the various individual techniques on which a general methodology of fault-tolerant design may be based.

This chapter describes a set of specific fault-tolerance techniques and their individual implications for hardware design. A summary of the hardware implications is given in in Section 6.8. Some of the techniques are intended for both application and operating system fault tolerance, while others are more well suited to one class or the other, as noted. The discussion of particular fault-tolerance techniques is preceded by a discussion of hardware implications of the encapsulation principle, which, while not a fault-tolerance technique in itself, is a fundamental technique for fault isolation and reconfiguration.

The techniques described are:

- Encapsulation,
- Processor redundancy assignment,
- Fault detection and correction logic,
- State recovery,
- Run-time assertion checking,
- Watchdog timing,
- Watchdog processing,
- Robust data structures.

6.2 Data and Program Encapsulation

6.2.1 Fault-Tolerance Function

Data encapsulation is an application of the general principle of encapsulation to data. Data are organized in sets, each with its own mechanism for controlling access to individual data items. The set may be given a name, representing a type of data, so that individual elements represent instances of the type, e.g., month (January, February,...). In some designs, strong rules are enforced as to what types of data may be combined in an operation. The access mechanism may have some associated rules under which access for reading or writing are allowed, and may have other associated information that may be useful in relating the set to other data sets, e.g., data types, ownership, and times of creation or modification. Use of an access mechanism helps to increase reliability by:

- Eliminating the kind of incorrect data accesses that can result when data is accessed directly on the basis of a computed address;
- Allowing detection of certain software errors, in which data of different kinds, e.g., alphanumerics and floating point numbers, might inadvertently be combined;
- Facilitating relocation of data, by centralizing the mapping of data names to physical locations;
- Limiting the effects of erroneous data modification.

Program encapsulation applies the encapsulation principle to the structuring of programs and interprogram communication. Program variables are divided into two groups, only one of which is accessible to other programs. Externally, the behavior of the program is defined only in terms of the externally accessible variable; the internal variables are effectively invisible, or "hidden". This is an example of the general principle known as Information Hiding [PARN72], which has many benefits in software practice, such as support of hierarchical structure and portability. As an example, Ada packages are encapsulated programs [TAYL80]. As in the case of data encapsulation, program encapsulation limits certain kinds of software errors, such as inadvertent flow of control and data reference. The encapsulation mechanism also permits association of information that defines relationships among encapsulations, such as access rights, alternate program versions, times of creation and use, and the like.

Object encapsulation is a unified encapsulation technique under which all the various computational entities

such as data, programs, processors, channels, etc., are treated uniformly, i.e., as so-called objects (which may be thought of as abstract machines). Each object has a data set and a set of functions through which data are accessed and modified. Several architectures have been developed around this principle, extended by such features as capabilities, which are tokens of access rights that support secure information sharing [GILO83].

6.2.2 Hardware Implications

Data encapsulation implies use of some data structures, such as tables, for mapping names of data sets into physical addresses, and for accessing individual members of data sets. Use of associated information to explicitly define the type of the data tags in association with the names for checking correctness of the access request. Since such requests occur at almost every instruction, special hardware is needed for both address translation and access validation.

Mechanisms to support program and object encapsulation can be rather elaborate, involving manipulation of data structures such as stacks and heaps (collection of possible multiple instances of data types) [GILO83 AND SMIT81]. Discussion of these mechanisms is beyond the scope of this report, but it is worth noting that one architectural effort (RISC1) is motivated by the observation that "The procedure call/return is the most time-consuming operation in typical high-level programs" [PARN72]. We note further that Ada provides most of the good qualities of encapsulation and hierarchy discussed, with the notable exception of robustness with respect to failed processors in multiprocessor (or distributed processor) systems [KNIG83].

A technique for detecting memory management errors that is particularly useful for Ada packages is the use of "signatures" (a kind of checksum) for checking that dynamic structures such as stacks and heaps are consistently modified. The elements of a data structure (either the data elements themselves or tags associated with blocks of elements) are treated as symbols in a code space. The signature is a single symbol that is a function of a set of all symbols in a set. The signature is updated upon the additional or removal of symbols, and following a change, the set function may be compared with the signature for consistency. Appropriate hardware consists of a signature "adder" and a small processor that acts as an observer of memory I/O data. Another use of signatures and monitors is discussed in the sections on watchdog processors. Other techniques for memory protection are discussed in the section on robust data structures.

6.3 Processor Redundancy Assignment

6.3.1 Fault-Tolerance Functions

This section discusses the configuration of multiple processors to support various combinations of hardware and software fault-tolerance. These configurations may be considered fixed for a given machine or virtual within a processor set. Techniques for fault detection, masking and recovery are also discussed. Implementation of those functions may justify use of yet additional processors.

Single and multiple processors can support several modes of redundant computing, such as:

- Dual processor redundancy for hardware fault detection;
- Triple processor redundancy for hardware fault masking;
- N-fold processor redundancy (N greater than one) for software fault masking, as in N-Version programming;
- Single processor, serially employed for alternate program versions, for acceptance-based software fault masking, as in recovery blocks;
- N-fold processor redundancy, employed in parallel for alternate programming versions, for acceptance-based software fault masking (a parallel version of recovery blocks).

In practical systems, both hardware and software fault-tolerance may be required. Various combinations are feasible, for example:

- Combined hardware and software fault-tolerance: Sets of dual or triple processors execute multiple software versions concurrently. Hardware and software fault detection also proceed concurrently. The number of processors is the product of the hardware and software redundancy factors;
- Hardware fault-tolerance plus serial acceptance test: Dual or triple processors are configured for hardware fault-tolerance. They execute identical replicas of a selected version, and switch between versions based on acceptance tests applied to version outputs;
- Software fault-tolerance plus periodic hardware test: Dual or triple processors execute different program versions concurrently to provide software fault masking. The processors are subject to periodic

testing for hardware fault detection, by self-test or comparisons between processors. (Note: this use of concurrently executed redundant software periodic hardware fault checking is directly symmetric to the use of concurrent hardware-fault-tolerance and periodic software testing in the preceding scheme).

Several mixtures of roles are advantageous in the use of available processor resources, for example:

- Dual hardware and dual software fault-tolerance: In a four-processor set with two program versions, the versions are executed on separate processor pairs. Members of each pair are compared to detect hardware errors and members of different pairs are compared to detect software errors. (Note: additional processor resources can be used to upgrade fault-tolerance from detection to masking in hardware or software, as needed);

- Shared processor-dual hardware and dual software fault-tolerance: In a three-processor set with two program versions, one version is executed on one processor, and identical replicas of the other version are executed on a pair of processors; outputs of the pair are compared to detect hardware faults. If those outputs are identical, any unacceptable deviation between the outputs of the two versions implies the existence of software fault in one of the versions or a hardware fault in the solo processor. Separation of these causes would be done by reconfiguring the processors and applying further tests. This scheme involves an exchange of information between hardware fault and software-fault detection processes. The complexity of this exchange might be considered to be a source of error. The benefit obtained is a saving of processors, or, in the case of a degraded set of processors, an opportunity to use available resources to maximize fault detection or masking. For dual-version software, three processors are used instead of four. For triple-version software, four processors are used instead of six.

6.3.2 Hardware Implications

There is a major architectural choice between fixing the assignment of processors to specific redundancy roles and making the assignment dynamic. Fixed assignment offers simplicity in data communication and in task scheduling and dispatching. Dynamic assignment offers several attractive benefits, including:

- Economy in the number of processors required for a given order of hardware and software fault-tolerance.

This is beneficial both in the initial sizing of machine resources and in making the best use of processor resources in the event of processor loss;.

- Flexibility in the use of the level of hardware and software redundancy according to reliability needs and experience. For example, certain critical functions may require high orders of redundancy. If these occur infrequently, fixed assignment of processor resources would be very wasteful. Also, if a particular processor has an unusual history of reporting discrepancies between software versions, implying possible internal faults, it might be advantageous to assign it to a hardware fault-tolerant pair, in order to clarify its hardware "credentials".

Changes in assignment of roles to processors will occur infrequently, and should require no special hardware facilities. Decisions about the dispatching of data to processors according to their assigned roles may, however, occur frequently, e.g., for each quantum of data transferred (file, message or work), depending on how the architecture provides for communication among processors, memories, channels and error detection and correction modules. Clearly, information as to redundancy configuration, assigned roles, and perhaps data paths, must be used to control data flow. This information may be distributed and applied in several ways, as in the following schemes:

- Shared memory/distributed dispatching: Processors with local data stores and autonomous dispatching capabilities make independent calls for data from a central store. Each processor's operating system contains all information about configuration and role assignment. The rate of change of context information is low, so no special hardware appears to be necessary;

- Shared memory/centralized dispatching [MATS83] A set of processors, with relatively little local data store and no dispatching capability, processes data that is dispatched from a shared memory by a central controller. Communication between the controller and the processors will require appropriate hardware support. The controller itself constitutes a special hardware facility. This scheme is appropriate to a situation where a large number of processors is needed for concurrent processing. Centralizing the configuration and assignment logic helps to simplify the processors. This concept is carried out strongly in Dataflow computing systems, where sequencing of parallel and sequential operations is supported by both the programming language and by the machine hardware. Flexible application of machine resources

to multiple replications or multiple versions may be organized at the language level;

- Distributed processing: Configuration control, scheduling, dispatching and processing are controlled and conducted at each node of a distributed processing network. Information needed for redundant task dispatching and intercommunication may be communicated through normal mechanisms. Synchronization of distributed results presents a special problem, as discussed in the following section.

6.4 Fault Detection and Correction Logic

6.4.1 Fault-Tolerance Functions

The following functions are widely applicable in hardware or software fault-tolerance, and are candidates for special hardware implementation because of their frequency of application: [SHRI82]

- Consistent data replication and distribution: Standard fault-tolerance schemes such as dual and triple redundancy for hardware faults and the two multi-version schemes for software fault-tolerance base their fault detection logic on the assumption that output data of corresponding replicas of versions are derived from identical input data. This assumption is invalidated by several kinds of faults. The following problems may occur in practice;

  Inconsistent replication in tightly coupled systems: In distributing copies of a unique datum to several processors, there are several kinds of faults in practical distribution systems (e.g., different threshold levels in receivers on a common bus) that may induce differences in the received data. This problem is solvable by the use of one of the interactive consistency algorithms [MAKA82], which requires several time-consuming exchanges of input data among the receiving processors, with voting;

  Excessive data dispersion in distributed systems: An attractive strategy in distributed process-control systems in which input data vary slowly is to allow redundant programs, either identical or multi-versioned, to accept input data that are approximately the same but not necessarily identical; for example, samples of a single source at slightly different times or samples at the same moment of time from different sensors that read the same physical variable.

In comparing the outputs of processes that use different inputs, care must be taken to distinguish those differences in output due to input differences from those due to to processing faults. Making this distinction might be enhanced by careful control or reduction of the amount of dispersion in the values of the several input samples. Recent investigations [PATT82] suggest that the logic of such processing must be application dependent.

A technique of possible importance in distributed systems is to attach time information ("time stamps") to data samples, in order to avoid time confusion due to internal and possibly unpredictable time delays;

- Hardware fault detection and error masking: Experience has shown that the time required for error detection and voting in three-fold and five-fold redundant systems can be very significant in high-performance process-control applications. Feasible hardware implementations for these functions have been described; [SHIR82]

- Adjudication: In both the recovery blocks and N-Version programming schemes, outputs are used by the system only if they pass a test of "acceptability" using user-defined test functions. In the standard N-Version scheme, acceptable outputs are combined according to some appropriate rule and forwarded to the system. In the standard recovery blocks scheme, control is passed to alternative programs. The former is a form of forward error recovery, and the latter is a form of backward error recovery.

  The functions of acceptability testing, output combination and transfer of control have together been called the "Adjudication" function. This represents a unification of the recovery blocks and the N-Version schemes. The complexity of the adjudication function can vary considerably, depending on the complexity of the primary program and upon the degree of software fault coverage desired. For example, an acceptance test for a program may constitute a single line of code or it may require execution of a program of the same size or even larger (written appropriately, an acceptance test might be usable as another version). The time allowed for adjudication also may justify use of separate processors. For lengthy acceptance tests, pipelining of version execution and acceptance-test execution may be advantageous:

  The outputs of concurrently executed N-Version programs require logical synchronization due to inevitable differences in the execution time of different

implementations. [MAKA82]. In tightly-coupled systems, this is no different from other process synchronization problems. In distributed systems, where node or link failure is a significant possibility, the synchronization process for combining results derived from different nodes must consider the possibility that some input may never appear. Special hardware other than timers does not seem necessary;

- Numerical error control: Significant errors in numerical computation can result due to the characteristics of the algorithm used and of the particular form of finite arithmetic employed in a given computer. Such errors may result in large deviations between the outputs of program versions that may use different numerical methods for the same function. Improved rules for floating-point arithmetic that reduce this effect for important computations have been proposed; [CART83]

- Context management: In some cases, most notably in operating systems, the amount of information that is potentially changed by an operation is very large, and may be generated by many different subprocesses. Examples are processor-to-resource mapping data and status bits. Version comparison or acceptance testing may require considerable effort in the collection of information from the various subprograms that develop or use it. For high speed systems such as the resource managers of operating systems, it would be very advantageous to be able to call up all state variables for a version very quickly for comparison with the proposed values from other program versions. It would be further advantageous to be able to distinguish only those state variables that have changed since last inspection. [ANDE81]

6.4.2 Hardware Implications

The special functions described were selected because in the cases where they are applicable, they occur very frequently, and therefore could have harmful impact on performance. Their implications for special hardware design are exemplified in the following:

- Requirements for consistent replication of input data and output data comparison and voting imply a need for direct logic for three-way voting and comparison rapid data exchange between processors;

- Requirements for time coordination of data samples in distributed processing systems imply a need for time stamping of data samples;

- Requirements for adjudication of multiversion software results imply a need for general-purpose processors for computing acceptance tests and output combining rules; downloading of microprograms specialized to specific applications, and pipelining of primary computations and adjudications may be valuable;

- Requirements for very rapid comparisons of large number of data elements such as the various control signals found in operating systems imply a need for:

  Rapid context retrieval.

  Rapid extraction and comparison of most-recently changed variables;

- Requirements for controlling errors in numerical computation imply a need for improved rounding algorithms for floating-point arithmetic units.

6.5 State Recovery

6.5.1 General Issues

State recovery is a fundamental function in fault-tolerant computing. Faults usually cause state variables of an application program, or of the machine itself to be set to some improper values that not only may give incorrect outputs, but may also prevent correct future computation. If fault tolerance involves machine reconfiguration, it will have to be initialized to the correct state. The goal of state recovery in fault-tolerant computing is thus to assure that, following faults and possible fault correction actions, all application and machine state variables have values of state that enable correct computation to proceed.

If input data and initial state information for a computation are retained throughout a fault event, it is possible, in principle, to repeat the computation to obtain the the correct results. Such recovery is called backward error recovery. In a real-time system, the time and resources required for this repetition may prevent the servicing of new input data. If that service is more important than correcting past results, it may be preferred to act as quickly as possible to set system state variables to values that will allow computation to proceed. Such recovery is called forward error recovery. Some combination of forward and backward error recovery may be advantageous, in which past-future computed results may be sacrificed in some proportion.

Backward recovery is necessary in applications where some state values have long-term importance. Forward recovery, in which accuracy in some values may be sacrificed, may be acceptable in systems (e.g., sensor-driven systems) in which state values are quickly derived from, and strongly determined by the most recent data. Practical real-time systems will require both forms of recovery.

It is perhaps unfortunate that the recovery blocks approach has been associated exclusively with backward recovery and that N-Version programming exclusively with forward recovery. In some applications where it is essential that state variables be preserved, one may need the output-combining aspect of the N-Version scheme together with some recovery capability. A particular problem that occurs in the N-Version scheme is that if each version maintains such state variables internally, it may be a non-trivial matter for one version (or a cooperating set of versions) to correct a damaged state variable in another version, because the data representations used by the versions may be different (and deliberately so!); for example, data might be organized in various equivalent structures, such as lists, trees and hash arrays. The reliability advantage of using different representations must be balanced with the performance cost of transforming representations in order to enable interprocessor cooperation in fault detection and data restoration.

6.5.2 Fault-Tolerance Functions for Backward Error Recovery

The key issue in implementing backward error recovery is efficiency in time and computing effort. For example, in selecting a previous system state (i.e., the values of the set of all system state variables) for return, the one nearest in time that allows full regeneration of lost results is clearly desired. In order to minimize redundant computing resources and restoration time, it would also be desirable to distinguish only those state variables that need to be modified from among the large number of state variables that may be involved in the particular computation.

The first goal, finding the nearest valid recovery state, has been subject to considerable investigation [NEUM80 and ANDE81] For single processes, useful recovery. states are known as recovery points, the value of which are determinable to a recovery routine with little difficulty. For sets of alternative programs employing the same input data, as in the recovery blocks scheme, the proper recovery point is also clearly defined.

Complications arise in dealing with sets of communicating processes, since all processes that may have contributed to an unacceptable computation may be suspect. Two

approaches have been considered. In the first, a set of processes are encapsulated in a compile-time entity known as a conversation [NEUM80], which is treated as an atomic recovery module; that is, when any recovery is needed, all processes return to the points at the entry of the module. In the second approach, records are kept of all actual interprocess communications, and the records are traced backwards until a consistent set of recovery points is found. This approach requires imposition of a discipline of interprocess communications in order to avoid pathological conditions that could result in a massive unraveling of computations. [RAND83a]

The goal of economical state recovery itself requires setting correct values to only those variables that have changed between the recovery point and the point of error. In the technique known as the "recovery cache" [MELL83 and LEE80] the names and initial values of state variables that change following a recovery point are recorded in a special memory (a cache). In order that this action does not slow normal computation, the cache performs an associative search on the names of the state variables it contains, so that for each state variable accessed in the computation, it is immediately determined whether or not that variable has previously been modified.

A possible alternative to the use of this special mechanism is to use serial-access data structures, such as hash tables or balanced trees, to record initial values for the actual state variables visited in the program. Despite the efficiency of such structures, the computational load may be significant and may justify special hardware.

6.5.3 Forward Error Recovery Functions

Less attention has been given to forward than to backward error correction. The most well-known form of forward correction is fault masking, such as voting, in triple modular redundancy, or version-output combination, as in N-Version programming. This technique cannot in itself assure that all state variables that may be needed in future computations have been correctly established following an error. Correctness in this case does not necessarily mean state values that correctly represent past history, but rather values that are consistent with the application (e.g., values of altitude, velocity and acceleration that are feasible combinations for a given aircraft). An additional objective might be that the new states should be reasonable approximations to what they would be if the error had not occurred.

The problems of determining consistent points in the program from which future computations may proceed, and of determining the subset of all state variables that need

restoration, are essentially the same as in backward recovery. It is much more difficult, of course, to determine what the new values should be. Many heuristics can be imagined, but it would seem that the rules for determining new values must depend on the application. In practice, it may be important to compute the new values very quickly, in which case special processor support may be justified.

6.5.4 Hardware Implications

Requirements for management of compile-time-encapsulated sets of recovery processes imply the need for special operating system service, but do not require special hardware.

Requirements for dynamic reconstruction of consistent recovery points in sets of processes imply the need for maintenance of records of interprocess communication events. In some applications, this service may be a significant performance load on the operating system, and may justify the use of a separate processor.

Requirements for forward error correction in real-time systems imply a strong possible need for special hardware to combine the outputs of multi-version programs. The combinations will be application dependent, so in a multiple application environment, the hardware should be programmable, and programs that specialize its function for particular applications should be changeable during operation.

In real-time, high performance systems, requirements for restoration of changed state variables in backward recovery or for estimation of lost state variables in forward recovery imply the need for a Recovery Cache (a name-associative store), or a special processor to maintain associative-like data structures.

6.6 Assertion Checking

6.6.1 Fault-Tolerance Functions

Assertion checking is a kind of acceptance testing, applied to interior points of a program. [ANDE79 and LEVE83] A logical expression is associated with selected program points that asserts some relationship among program variables that should be true whenever program control resides at that point. It has been shown that such assertions can be written for all programs with a single thread of control. Such assertions can be used to verify analytically the correctness of program code. During program execution, evaluation of an assertion expression to a value TRUE implies correctness in the program and in the hardware used up to that point. The assertion checking

technique addresses only the error detection function of fault-tolerance, and does not prescribe a specific recovery technique. Results of assertion checks must, of course, be reported to an error-handling process.

In practice, it is difficult and time consuming to write assertions that completely define the applicable relationships. Some experiments [ANDE79] have indicated that useful error detection may be obtained using incomplete specifications of the kind that a reasonably skilled programmer could declare intuitively. The expressions are typically simple computationally, but it is conceivable that they may become more powerful (and complex) as experience is gained with this technique.

A possibly significant burden on performing might be experienced in retrieving all the system state variables that might be relevant to an assertion. Research is needed to determine the significance of this effect.

6.6.2 Hardware Implications

Use of the current style of assertion checking does not impose a significant burden on performance. However, the scheme has potential for considerable elaboration, with consequent performance implications. Special hardware could be justified for:

- Tagging assertion statements as special data types,
- Employing extra processors for parallel evaluation of assertion functions,
- Buffering global state variables (e.g., in a conventional cache) to reduce the performance burden of data access.

6.6.3 Watchdog Timing

6.6.3.1 Fault-Tolerance Functions

Watchdog timing is a form of assertion checking in which the semantics of the check on software correctness are reduced to the dimension of time [ORNS75]. This kind of check is useful because many software and hardware errors are manifested in an excessive time taken for some operation. Examples include loss or improper flow of control and incorrect synchronization of multiple processes. The technique is also attractive because programmers usually are able to declare reasonable bounds for program steps without much difficulty.

In typical practice, timing is applied to only one process at a time. For high-performance systems with appreciable process concurrency and substantial nesting of process calls, the use of multiple time-checking intervals is attractive. Such intervals might be nested or overlapped, in the general case.

Several levels of testing complexity may be employed. Perhaps the simplest cases are the measuring of the time required to execute a program loop or to accomplish a call-return exchange between a pair of processes (procedures, subprograms, etc.). In this case, control remains at or returns to the point in the program at which the time measurement is initiated. The time limit for such measurement would be declared in the program body.

A more complex case is that in which the program is required to move from one control point to another within a given period of time. The complexity arises when there are several possible destinations and several possible paths to a given destination. Consider source point A, destination points B, and C, and intermediate points D and E, with paths ABC, ABD, and AED. The three paths may have significantly different appropriate time bounds. In departing from A, three time measurements may be initiated, corresponding to the three possible paths; then as the program selects a particular path, measurements for logically excluded paths would be terminated. This approach requires that information concerning selection of a particular path should be located so that it appear prior to the timing out of alternate paths.

The foregoing discussion is not intended to offer a definitive solution, but merely to indicate the possible complexities of implementing a powerful watchdog timing facility.

Some enrichment may also be profitable in the handling of error reports. For example, if transient, low-importance errors are frequent events, some time-out reports might be accumulated, and even dismissed before notifying the main error handler.

6.6.3.2 Hardware Implications

The watchdog timing facility should be as independent as possible of the software and hardware it protects. For the simplest case of only one measurement at a time, the initiating program (application or operating system utility) must communicate to the operating system a time period and a reset command, and the operating system must provide a time measurement and alarm service.

For multiple concurrent time measurements, a separate processor may be justified, including:

- A time clock, perhaps with several output frequencies in order to minimize measurement activity for widely varying time intervals;
- A set of time registers with associated information, such as program source and distribution points, logical reset conditions and counting rate;
- Programs for processing error reports.

6.6.4 Watchdog Processing

6.6.4.1 Fault-Tolerance Functions

Watchdog processing is yet another form of simplified assertion checking, in which the semantics of the check are reduced to program path traversal. The technique as described in the literature [NAMJ83] does not cover the dimension of time, but it could be employed in conjunction with watchdog timing. The basic idea of watchdog processing is to check that the control path followed in a program execution is legal, i.e., logically consistent with the program text.

Several strategies are conceivable, but all involve use of one processor that observes instruction flow in another processor. One technique employs the concept of a "signature", which is a symbol whose value is a function of a string of symbols, usually of the same size. Examples from current practice are sum-checks and cyclic redundancy checks (CRC), used to check the integrity of data storage and retrieval. In the proposed scheme [NANJ83], a check symbol is associated with each program node at complication, and a deterministic signature is computed for each path. During execution, check symbols are accumulated into a running signature, and information is derived that recognizes the path that is actually traversed (there are several alternatives for dealing with loops). The final signature is compared with the precomputed path signature. In one implementation, check data are embedded in the program. In another, a graph scheme for the program is stored separately, together with node check symbols, and a path is followed according to branch information derived from observing program-counter behavior in the primary processor. The use of signatures for checking for correct updating of special data structures (e.g., stacks and heaps) is discussed in the section on data encapsulation above.

6.6.4.2 Hardware Implications

Some form of separate processor is required with access to instruction and operand data of the primary processor. For the scheme in which check symbols are embedded in the program, only a logic unit for signature accumulation, together with a simple programmable controller are needed. For the scheme in which a separate program schema graph and associated check symbols are maintained, a substantial amount of additional memory may be needed, depending on the complexity of the program graph.

6.7 Robust Data Structures

6.7.1 Fault-Tolerance Functions

Data bases containing complex data structures such as trees and multiply-linked lists are vulnerable to damage from errors in the programs that update them. Various schemes have been proposed for employing redundancy to protect data bases from errors of various kinds. The notion of Robust Data Structures [RAND83a and SHIR82] has good theoretical foundations, and seems to be of practical value in some commercial products. In that scheme, redundant links between data elements are employed in a way that allow error detection and correction for some kinds of error. In practice, a separate inspection is conducted of the data base, at a frequency that fits the experienced error rate.

This scheme is considerably more economical than one in which several program versions maintain independent data bases, but it assumes that errors will not occur that preserve consistency. As with many other assumptions about software errors, more experimental results and practical experience are needed.

Another approach to memory error detection is the use of signatures, which is described in the section on encapsulation above.

6.7.2 Hardware Implications

Checking and correcting of data structures can be done independently by a dedicated processor. This technique is in current use [MELL83].

6.8 Summary of Hardware Implications

This section summarizes the findings on hardware implications of the fault-tolerance techniques. The findings are organized here according to architectural feature, including

basic system functions and special logic units, memories, and monitor units. Motivations and explanations of the use of these functions are given in the discussions of the preceding section.

6.8.1 Basic System Features

The following features are not specific to or essential for fault-tolerance, but they are strongly supportive of reliable programming and they tend to simplify the design of fault-tolerant mechanisms:

- Mechanisms to support program encapsulation (e.g., Strong Data Typing and Program Domains) for the purposes of intrinsic reliability and limitation of error propagation, including:

  Name-based memory management,

  Capability labels and machine support of label inspection, such as hashing and table management;

  Stack management to support rapid context change and interprocess parameter passing;

- Mechanisms to support hierarchical design in operating systems, (e.g., kernel and ring-based design), including:

  Multiple register sets and high speed access to memory to support rapid context change;

  Special logic (e.g., microprograms) to support rapid control of basic operating system functions, e.g., interrupt processing, memory management, process scheduling, interprocess communication and type checking;

- Mechanisms to support time and data synchronization in distributed and federated systems, including:

  Time and source-identification labels attached to inter-node data (Time Stamping),

  Special processors for time-keeping (Fault-Tolerant Clock Synchronization).

6.8.2 Other System Features

- Support for flexible processor allocation among replicas, versions and tests:

For architectures employing a pool of simple processors, special controllers for dispatching tasks to processors and directing processor outputs to hardware fault detection units; modes may include parallelism and pipelining;

For distributed processors, special modes for inter-node communication to accelerate collection of multiple versions and replicas;

- Special logic units for rapid execution of fault-tolerance functions, including:

  Identity comparators and voters for hardware fault detection and masking;

  Programmable logic for version-output combination, e.g., inexact voting;

  Voters and rapid data transfer facilities to support consistent input data replication (Interactive Consistency) for software and hardware redundancy;

  Devices to compute signature check symbols and to check the integrity of dynamic memory structures and flow of program control (as in the watchdog processor technique);

  For process-control applications with requirements for minimal transport delay (sensor-to-actuator delay time), special microprogrammed processors for rapid combination of multiple-version computed outputs, using application-dependent combining functions;

- Special memories for state recovery, including:

  Name-associative cache memories for backward and forward error correction in programs with large state-variable sets.

- Special processors for on-line monitoring of computing activity with access to data busses, program counters and memory units and special logic support, including:

  Access to data busses, program counters and memory units;

  Fast recognition of special symbols that denote monitorable data;

  Multiple time-out serrvice (Watchdog Timers); special timers to support time measurement;

Program-control path checking (Watchdog Processors); special logic units to support signature accumulation;

Memory error detection and correction (Robust Data Structures);

Memory error detection (Signature Checking).

7.0 Software Fault-Tolerant Operating Systems

7.1 Introduction

This section discusses techniques for achieving system software (i.e., operating systems) that is tolerant to design flaws. The major themes addressed are:

- A framework that is suitable for both system software and the applications software that it supports. Moreover, the framework supports the different approaches to software fault-tolerance: Recovery blocks, N-Version programming, and combinations of these two approaches;

- The identification of the reliability kernel, i.e., that portion of the system that provides the basic mechanisms that the rest of the system will use to achieve software fault-tolerance. The reliability kernel will not be fault-tolerant and, consequently, should be correct, or "trusted", in order for the redundancy in the rest of the system to be managed so as to achieve fault-tolerance. The reliability kernel is hardware fault-tolerant to the system and to the security kernel of a system intended to provide data security;

- Hardware features that support the objective of minimizing the performance penalty associated with providing software fault-tolerance. Such hardware support is essential for system software which even in the absence of fault-tolerance can exact a heavy overhead burden.

One straight-forward approach to using different versions of operating system software for fault-tolerance is to encapsulate separate versions of the operating system, together with the set of application programs it serves in a separate computer. The resulting architecture would be similar to the SIFT mutlicomputer concept [GOLD80 and GOLD84]. This approach masks faults occuring anywhere and permits recovery for all programs that do not have an internal state or have only a few state variables. The approach is quite suitable for achieving full software fault-tolerance (masking and recovery) for all application programs suitable for execution on SIFT, but only masking for the operating system of SIFT. Of particular interest about this approach is that it does not require any trusted system software. Embellishments to the basic concept to support recovery for the operating system entail a nontrivial albeit modest size reliability kernel. However, even with these improvements the SIFT-based approach is limited; for example, it requires a multicomputer organization and it also

places demands on the organization of application programs that might be unacceptable.

Towards a more flexible framework for fault-tolerant system programs, a hierarchical operating system is presented. The lower levels of the system (which can be viewed as the reliability kernel) provide the mechanisms needed by the higher levels. The mechanisms of the kernel support the fault-tolerant needs of the higher levels and provide the basic building blocks the higher levels require to achieve their functionality. It is convenient to view the kernel as all levels below some distinguished level, the distinguished level being decided on the basis of the tradeoffs between cost and reliability benefits attendant to applying fault-tolerance to levels below it. The kernel complexity is dependent on such properties of the overall system architecture as the degree of distribution of computation, what are the key properties of system programs relative to fault-tolerance and what hardware support is vital to performance. Key hardware features are those in support of:

- Multiprocessing hidden from most of the system software and all of the application programs;
- Encapsulation (providing error confinement) through the use of tagging, capabilities, or descriptors, as in some designs for secure systems;
- Context, switching (i.e., the establishment of domains that define the objects accessible to a process); the establishment of conversations;
- Acceptance test computation;
- Identification of recoverable data sets;
- Level registers that assist in the processing of hierarchically structured systems;
- Exception handling.

7.2 SIFT-Based Software Fault-Tolerance

Briefly, the application of the SIFT concept to software fault-tolerance is as follows. The underlying machine organization is a multi-computer, where the individual computers are interconnected by a network that allows direct communication between each pair of computers. The basic computational unit in SIFT is a task, i.e., a unit that computes outputs that are each a function of the task inputs; a task may or may not retain state information between invocations. Fault-tolerance is achieved by replication and voting at the level of tasks. Each task is assigned to 3 or more computers. If the outputs of an instance of a Task A are

required as inputs to an instance of a Task B, the following steps occur: The (3 or more) task A outputs are broadcast to all computers (although only those running task B actually require them). The broadcast values are voted by all computers. The majority-voted value computed in each computer is used as the input for the execution of task B in that processor, the collection of disagreeing inputs to the voter are further processed in an attempt to identify computers that might have failed, and a computer judged to have failed is logically removed from the configuration (by being essentially ignored by all good computers).

The physical separation of the computers in SIFT achieves confinement of a fault to single computer. This confinement coupled with the replication and voting assures the masking of any single fault; a higher degree of fault-tolerance is obtained when the task replication is five or greater. Transient faults are distinguished from permanent faults by determining the persistence of errors caused by faults.

Let us now consider how the SIFT architecture can be used to achieve software fault-tolerance, primarily through the use of N-Version programming. In the current use of SIFT all of the programs implementing a task are identical and can be assumed, in the absence of hardware failure, to yield identical results for each invocation. For the assumption of identical programs, exact-match voting is used. To achieve software fault tolerance, nonidentical programs will be associated with each task. It cannot be assumed that each of these programs will yield exactly the same output -- even when presented with identical inputs. Hence it will be necessary to use approximate voting. It has been previously noted that voting in SIFT is time-consuming, especially if the replication is greater than three. Since approximate voting will be more expensive than exact-match voting, a hardware or microprogram implementation of the voting function would be very desirable. Thus, it is clear that errors in the application programs are masked.

What about achieving fault-tolerance for the SIFT operating system? The key is for the specifications of each of the operating systems run by the computers of SIFT to be identical, but for the implementations to be different. What does it mean for the specifications to be identical? Our design verification of SIFT [MELL83] is with respect to a collection of design specifications that are input/output specifications for the SIFT operating system. These specifications provide constraints on the task schedules, assure synchronization of the computers, identify key tables (e.g., noting working and nonworking computers, prevote and post-vote buffers, and task dependencies), among other properties. It is not difficult to suggest different implementations. For example, one instance of the scheduler could use

a table with a linear search, another with a logarithmic search, yet another with a hash representation. (Of course, the execution times for these instances will not be identical. The slowest instance will, then, pace the other instances.)

Under the assumption of implementation independence, any single software error is masked or has no impact on the system's behavior. If there is an impact, the output of one or more) application tasks on a single computer will be in error, and voting of the application outputs will mask this error. Note that the error could be manifested as a task failing to compute any output. Furthermore, there is no trusted component concept for the interconnection network itself: the physical isolation and the absence of any significant sharing of data among computers means all software is redundant and checkable. However there is one major deficiency of the approach: The computer suffering a fault in its operating system might not recover from the error following the fault. For example, if the error results in the damaging of a schedule table, the table is doomed to remain in error.

An embellishment to SIFT can be used to permit recovery of the operating systems but at the expense of requiring some trusted software. Recovery can be effected by each computer checkpointing its key tables for restoration following an error. Checkpointing can be achieved by each computer broadcasting the table values to other processors; actually, only changes incurred since previous checkpoints need be of concern. Since the internal representation of the table may be unique to each processor (to obtain protection from software faults), it will be necessary to translate the representation to a form that is consistent with that of the other processors. (1)

When a computer is thought to have suffered an operating system error (by, say, a majority of the other computers determining that the computer is behaving erratically), the process of recovery is as follows:

- The other computers, acting in concert, restart the clock of the failed computer. Hardware is clearly required here, in particular, a hardware voter the inputs of which direct lines from each of the other computers and the output of which is a reset line to the computer's clock;

---

(1) The SIFT-based approach to fault-tolerance is similar to the distributed approach to security suggested by Rushby [RUSH83] where the physical separation of processors assures the data in a processor is easily protected from abuse by other processors.

- The other computers broadcast the key tables to the failed computer;

- Once restarted, a computer runs a bootstrap program that reads (and votes) the table values. Now the failed computer is ready to be accepted back in service.

The reliability kernel consists primarily of the clock, the clock restart mechanism, and the bootstrap program. The voter is not necessarily in the kernel since it can suffer occasional faults without disabling its computer.

Additional embellishments are possible. For example, it might not be necessary to completely restore a computer's state upon noting an error. The origin of the error (i.e., the identity of a corrupted table) might be determinable by detailed analysis of the behavior of the failed processor. The failure to produce an output for just one or a few tasks would seem to indicate an error in the schedule table. The recovery,then, would involve restoration of only this table.

Even with the extension to allow recovery for operating system software, the SIFT-based approach is still not deal. In particular:

- In order to be compatible with SIFT, the application programs must require no internal state (or have a sufficiently small number of state variables such that they can be passed as inputs) and must run completion each invocation; in SIFT an invocation corresponds to a subframe interval of a few milliseconds;

- The scheduling discipline of SIFT requires the schedule tables to be preplanned;

- A multicomputer organization is required to obtain the benefits of fault-tolerance with minimal trusted software. As we discuss below, the SIFT concept can be realized on a single computer by appropriate multiplexing of resources, although the software to effect the multiplexing must be trusted; (2)

- Although all errors associated with a single computer are maskable, recovery from certain single faults might not be possible. Errant behavior is noted only when the results of an application task execution

---

(2) Again, the security kernel analogy is relevant. When resources are not physically separated, the security kernel becomes larger since the separation must be effected by software.

differ. It is certainly possible for a fault to impact an internal table, that would result in some future task suffering an error, but not impact the immediate task. The key issue here is that the application tasks being in agreement is not a strong enough test to guarantee all subordinate programs are free of errors. A finer granularity of checking is likely to be required.

7.3 Toward a General Framework for Reliable System Software

In this section we describe preliminary work toward a general framework in support of fault-tolerant system software. The major themes addressed are:

- The particular properties of operating system software that pose problems for fault-tolerance;
- A hierarchical structure in which each level provides the mechanisms that the next level requires to achieve fault-tolerance and its general functional requirements;
- Support for the different techniques at each level: N-Version programming, recovery blocks, and user-controlled recovery (as described by Parnas [PARN72]);
- The portion of the system that is to be trusted (i.e., the reliability kernel) in the sense that it is not fault-tolerant. Relative to the hierarchical structure, it is convenient to view a particular level as the boundary between kernel and nonkernel software: All levels below this distinguished level constitute the kernel. It appears that certain functions cannot be protected by redundancy; others, particularly if simple, might also be trusted since the cost of the redundancy might outweigh the benefits;
- How to embed distribution of computation into the design in a manner that is transparent to the user. Distribution in a hierarchical system can occur at any level -- or, in general, at a number of levels.

7.3.1 Problems Associated with Operating Systems

Perhaps the most distinguishing features of operating system software, different from the conventional application programs associated with aircraft or spacecraft, are a large internal state, extensive concurrency (real or virtual), and the need for high performance. Each of these features poses

problems with respect to the current approaches to fault-tolerant software.

The large internal state poses particular problems for the N-Version programming method. There is no assurance that the vote (approximate or identical) on output values is sufficient to detect error in the internal state; it is not practical to vote over the internal states of the replicas since having independent realizations, the state spaces will likely be quite different. It might be possible to define an abstraction function that will unify the state spaces of the different versions, thus enabling the vote to detect certain errors in the internal state. For example, different representations of a table (corresponding to different versions) might all have the same number of entries; a vote on the number of entries would be helpful in detecting many errors.

It is clear that the extra steps associated with software fault-tolerance, not required for conventional programs, exact a performance penalty. Parallel processing can help to reduce the performance drain. For example, each of the replicas in N-Version programming could be assigned to separate processors. Also, pipelining and lookahead could be used where tasks are not necessarily independent. For example, the computation of acceptance tests could proceed in parallel with conventional task processing, provided the system is prepared to backup if the acceptance test fails. A more subtle performance drain, however, seems more difficult to eliminate. An optimized program, particularly part of an operating system, interweaves the functions of what we call decision and commitment. A typical computation relates to deciding on the availability of a resource. A typical commitment would assign this resource to some process. Since commitments might be difficult to undo, the requirement of fault-tolerance dictates that the decision be checked prior to any commitment, although in an optimized program the two phases overlap.

7.3.2 Hierarchical Approach to Operating System Fault-Tolerance

Anderson and Lee [ANDE81] describe a general approach to hierarchical fault-tolerance that embodies all of the schemes that have been studied. The following are the possible approaches:

- A level, suffering a fault, is able to use its internal mechanisms to effect recovery. The mechanisms could be an alternate program, masking through N-Version program, or even hardware masking techniques. In either case, the next level is unaware of the fault;

- A level is unable to effect recovery, the general consequence being loss of computation or data. In this case, the level suffering the fault is required to recover to a consistent state and to notify the next higher level that an error has occured; the notification can often be through the mechanisms of an exceptional return, particularly if a programming language (e.g., Ada) supporting exceptions is used. The higher level will then use the facilities of the lower level to recover itself as best it can. As a simple example, consider the lower level to be a scheduler that has as its internal state a table of current schedules. If this table is in error, the next level (say the file system) could retrieve a previous version of the schedule table stored as a backup file.

Numerous efforts have shown that an operating system can be conveniently structured as a hierarchy of abstract machines, e.g., SRI's PSOS (Probably Secure Operating System). [PSOS] Typically, each of the levels in the hierarchy is viewed as a manager of some particular resource. For example, PSOS contains managers of such resources as: pages, segments, directories files, processes, I/O devices, procedures, etc. They key problem of concern to us is how to embed fault-tolerance into such a collection of resources. It is our view that the lower levels of the system should be trusted, primarily to avoid the performance penalty associated with making them fault-tolerant. Thus, we envision an operating system consisting of a kernel managing primitive resources: memory (primary and secondary), dispatching of processes, I/O devices, low-level communication protocols, and a recovery cache. The more complex (or virtual) resources of the operating system would then be built out of these resources and would be fault-tolerant.

The notion of generalized, hierarchically-structured resource managers should encourage a general style of design of acceptance tests. A resource manager is responsible for planning the allocation of resources to consumers (i.e., the next higher level) using some objective functions, such as priority, request order, urgency, and economic use of resources. Such planning may require complex computations, but the results should be checkable against the objective functions in a straight-forward way; for example, if the assignment of a set of resources to consumers must satisfy a priority criterion on the consumers' requests, a simple check can be made that the priorities of the requests corresponding to the assigned resources are in monotonic order. Similarly, although arriving at an efficient assignment of memory space may be very complex, given a wide range in sizes of requests and a very fragmented memory space, a check on the total size of allocations relative to requests would be simple to apply. It should be noted that the acceptance tests at level L(i) applied to the procedures of

L(i-1) that it calls can be derived from the specifications associated with L(i).

The notion of system functions as resource managers may seem, at first, not to address the important function of error handling. The following interpretation may provide a basis for considering that function as a resource management function. Consider a multilevel system wherein each level has a complement of exception returns, one of which is returned when the level cannot complete an operation. In other words, by returning an exception condition a level indicates to its caller that its proposed solution to the allocation problem at hand did not satisfy the criteria of acceptability. That may be because the problem was too difficult, or because the program that implemented the solution was faulty. The task of the upper level program that receives this exception call is to find some new way of solving the resource allocation problem, e.g., by reducing the complexity of the problem itself or by assigning it to a different lower level manager. This response is thus a kind of resource allocation function, and can itself be subjected to a test of acceptability. This approach can be extended to form a hierarchy of exception handling procedures.

7.4 Hardware Support for Fault-Tolerant Operating Systems

The hardware features that can mitigate the performance penalty associated with fault-tolerance are described below:

- Capability-based addressing. Each object in the system is associated with a unique capability. A resource manager can access only those objects for which it has a capability. Thus capabilities provide a form of redundancy on access, the benefit of which is to confine errors to just those objects a resource manager requires. The hardware implications of capability-based addressing are minimal, particularly if modern architectures such as the Intel 286 or 386 are used. The capability must be mapped into a real address if the object in question is a section of memory (e.g., a page or segment), into an I/O device, or into a procedure call (if the object is virtual) and has a realization only in terms of a program that manipulates it. There are several ways to implement the mapping in an efficient manner;

- A hashing function, implemented in hardware, that maps capabilities to integers. The algorithm of the hash function must be carefully chosen to permit efficient implementation and to minimize conflicts in the hash table. The hash function would be part of the kernel;

- A global object table that maps capabilities into addresses. At any instant, a portion of the table is in fast access memory, (e.g., an associative memory) the remainder in slower memory. That portion in fast access memory would contain the working set capabilities of the currently executing process. Some hardware support for the rapid loading of the associative memory would be desirable;

- The assignment of permanent capabilities to the I/O devices, registers, and certain protected regions of memory;

- Fast domain switching. A domain of a process is the collection of capabilities currently available to the process. Domain switching occurs when a call (usually a procedure call) involves a change in the domain of the process. For example, when a call is made to the segment manager, the domain would become the segment passed with the call and the table containing the mapping of segment capabilities to real addresses. When the segment manager returns, the new domain will not contain the capability to the mapping table. The hardware needed here is for a single instruction that establishes domains from the capabilities passed and capabilities internal to the called procedure;

- Hierarchy management. The propagation of calls and the return of exceptions is inimical to a hierarchically constructed operating system. Often a level will be bypassed, to avoid multilevel interpretation. For example, a call to the segment manager to write and read segments must be fast. The way to achieve efficiency is to have a partition of the segment manager as a primitive resource manager implemented in hardware. If this low level returns an exception (e.g., because the segment referenced is not present in hardware), then control is returned to the segment manager for handling. The key, then, is for the hardware to keep track of the calling and called levels, and also the identity of exception returns;

- Signature computation. We have previously noted that much of what a resource manager in an operating system does is in terms of tables. Since these tables are likely to be large, some economy is needed on computing acceptance tests involving these tables. For example, it will not be possible to compare the contents of two tables. The solution is to use digital signatures. The signature for a hash table could be some function of all locations that are occupied. Of course, careful selection of a function is required in order to minimize the probability of a

valid and invalid state having the same signature. Hardware support would be helpful here to permit the updating of signature in parallel with the updating of the table.

7.5 A Strawman Concept for a Distributed Computer Supporting Software and Hardware Fault-Tolerance

The following is one of several possible architectural concepts that might be used to examine the feasibility of techniques for efficient support of software fault-tolerance. It represents an amalgam of current advanced architectures rather than a radical departure. For convenience, we employ the acronym SFTA (Strawman Fault-tolerant Architecture).

SFTA is a distributed multicomputer employing a common file system, in the style of UNIX United. Programs in any computer may call for access to any file element without knowledge of its physical location. A global operating system (with copies in each computer) is responsible for reconfiguration and initialization of computers; its actions require a consensus among participating computers.

Addressing in SFTA is based on capabilities, which define access rights to addressable objects and domains of computational activity. The conventional notion of capabilities may be extended to include other information of a protective nature appropriate to fault-tolerance and distributed processing, such as time stamps, version set and object size.

The local operating systems are recursively structured as a layered hierarchy of virtual machines, each of which has a full set of error handling facilities. Errors are thus reported and handled hierarchically. The hierarchy is implemented as a set of utility modules, most of which take the form of resource managers. The modules are served by a kernel machine (probably implemented in microcode) which provides basic services of machine attention and communication. The utility modules may be subject to software fault-tolerance. The kernel may or may not, depending on its complexity. Use of fault-tolerance in operating system utilities requires clear separation of the decision and commitment phases of computer resource management.

Software errors are detected at two levels, on the basis of external consistency, i.e., by comparing results of several versions or by acceptance testing, and internal consistency, (by checking process, through assertion checking, path checking and data structure consistency and excess-time checking). We note that the two levels of checking are mutually supportive, both in error detection and in validation of acceptability. External consistency checking

requires programmed use of general purpose processors, either the ones used for regular computation, or additional ones, if high speed is required. Internal consistency checking is done by special purpose processors acting as monitors of internal computer activity, with access to data, memory address signals, and time sources. Various tags may be carried in the data stream of normal processing in order to identify appropriate data for the monitors.

Forward recovery and backward recovery are facilitated by special hardware. For high-speed control computations, the error masking function of forward recovery is provided by special processors, probably microprogrammed, that implement application-dependent functions for error detection and output generation. Both backward and forward recovery make use of special hardware that speeds up the restoration of improperly modified program variables to their initial state. This may employ a cache memory (for highest speed) or special logic to implement associative retrieval in conventional memories.

Hardware faults are detected using duplicated processing and detected and masked using triplicated processing. Continuing reduction in the cost of microprocessors may make it preferable to use directly-connected, hardware fault detecting processor pairs rather than to configure pairs out of individual processors. Special logic is provided for these hardware fault-tolerance functions that require high speed, such as voting, clock synchronization and interactive consistency/source congruence.

The proposed concept leads to configurations with a relatively large number of processors for:

- Concurrent processing of software versions,
- Recovery control,
- Acceptance testing,
- Forward error masking,
- Monitoring of internal consistency,
- Input-output.

Assuming the use of low-cost processors, the processors would consist of directly-connected hardware fault detecting pairs. The number of processors could be in the range of eight to twelve pairs. An appropriate internal architecture for a computer with such a large number of processors would be a pool of processor pairs, communicating through a high-speed redundant network, e.g., a multiple ring. Information of interest to the monitor processors would be distributed in the network. Spare processor pairs would be provided for each type of processor.

SFTA is only one of many possible architectural concepts. It is presented as an example of a coherent framework for studying design tradeoffs and possible unifications among the various hardware techniques discussed in this chapter.

8.0 Higher Level Languages and Fault-Tolerant Software

In this chapter the ability of higher level languages to support fault-tolerant software is investigated. Design principles are discussed in the context of fault-tolerant software. The level of application of fault-tolerant software involves an engineering tradeoff which is a function of the cost and coverage of the fault-tolerant techniques. The integration of accepted design principles for software engineering with the goals of fault-tolerant software is the development of a unified approach to fault-tolerant software known as Idealized Fault Tolerant Components (IFTCs). The relationship between IFTC and the major fault-tolerant software techniques is discussed, as are extensions of IFTC to handle watchdog processes. A set of fault-tolerant primitives which need to be supported by any given system for fault-tolerant software are derived. The use of the programming languages such as Ada and C to support fault-tolerant software is also investigated.

8.1 Software Design Principles and their Relationship to Fault-Tolerant Software

The principles of fault-tolerant software should be integrated into the software design process in a manner which minimizes the increased complexity of the design. In this regard, it is believed that the incorporation of the strong design principles of hierarchical decomposition and encapsulation, which are already accepted within the software engineering community, is required for the successful utilization of fault-tolerant software. Since the goal of these design principles is to aid in the production of reliable software by controlling the growth in complexity for large systems, their integration with fault-tolerant software design principles is highly desirable.

Hierarchical decomposition (also known as stepwise refinement or layers of abstract machines) attempts to control complexity by confining communications between functions in a well-defined manner. Encapsulation (related to the concepts of information hiding, modularity and abstract data types) attempts to limit the visibility of information between the implementation of a function and its environment. A particularly strong form of encapsulation is known as an atomic action. In an atomic action there is a clear distinction between what is known inside and what is known outside the atomic action. Thus, the atomic action cannot influence or be influenced by its environment until completion. In other words, an atomic action appears indivisible to its environment. For purposes of fault-tolerant software, the semantics of an atomic action are "all or nothing". That is, an atomic action will run to completion with success or it will have no effect on the state of the compu-

tation (1). These semantics are adopted because it may be better to discard the current computation than to try and patch up the state after an error is detected. One of the major goals of fault-tolerant software is to maintain a consistent state even in the presence of faults.

The encapsulation of a computation provides a convenient unit of reconfigurability for fault-tolerant software. In general, this unit can be any of the following:

- Primitive language function (e.g. division operation);
- Procedure;
- Process;
- Conversation between processes or a transaction;
- Domain of functionality comprising a set of processes and data objects with their supporting resources (could be used to implement a hardened kernel or a minimal set critical functions for a dissimilar backup);
- Entire computer complex (this is the last resort, the entire computing system is expendable).

The unit of reconfigurability relates to the resources which one is willing to discard if a fault occurs. Timely error detection intends to isolate faults to the current computation within the encapsulation unit. If this is accomplished, then the damage is limited to the resources encapsulated within the current computation. If the current computation is a well-defined function, then it may be reasonable to make a priority judgement about the extent of the damage done by the fault. When a judgement is possible, error recovery is facilitated.

In general, the decision about the granularity of reconfiguration is an engineering one based on costs and requirements. If the unit of encapsulation is large, then there is the potential to discard a large amount of resources (in the form of time, code, processing) when an error is detected. Also, it may be difficult to pinpoint the damage done and, indeed, error detection itself may be difficult and/or time consuming. The principle of atomic action will force certain levels of reconfigurability when dealing with resources

---

(1) The term state refers to the state of the current computation. The current computation relates to the processing being performed at the intended encapsulation. This could be a procedure, a process, or a collection of processes (domain) engaged in a conversation. [RAND78d]

which are difficult or impossible to recover. An obvious example is the decision to launch the space shuttle.

As the unit of reconfigurability is reduced, there is an implied need for increased error detection and support for encapsulation at these finer levels of granularity. This support implies a penalty in performance or increased support from the underlying architecture. Thus, we envision a tradeoff between the granularity of reconfigurability and the cost of encapsulation. The engineering of this tradeoff is application dependent.

8.2 Idealized Fault-Tolerant Components (IFTC)

Although many of the software fault-tolerant techniques were developed independently, they share many similar features. We can consider each technique as a particular packaging of the required ingredients of error detection and recovery. The suitability of the particular packaging depends on the requirements of the application. No single technique seems to be dominant in all cases. Idealized Fault-Tolerant Components (IFTC) are an attempt to incorporate fault-tolerant software principles into the accepted practices of hierarchical design and encapsulation, with minimum impact on the normal software development process. IFTC provide unified frame-work for many of he proposed fault-tolerant software techniques. [RAND83] and [LEE83] IFTC are a combination of modular hierarchical structuring techniques, atomic actions, and exception handling. IFTC give the designer control over the amount of error detection and the kind of error recovery techniques most appropriate for the application. Investigations of the formal semantics of IFTC is also underway. [CRIS82] and [CRIS84] Use of Ada to implement these ideas has also been reported. [SANT83] Figure 8-1 illustrates the concept of IFTC.

An IFTC is a well-encapsulated structuring unit for the organization of fault-tolerant software systems. Service requests are made to IFTC in the normal way (i.e., by procedure call or message, etc.). If the requested service can be provided without faults, the IFTC gives a normal response. During the servicing of the request the IFTC may itself make requests of subcomponents. These subcomponents may also be IFTC. Thus, IFTC provide a recursive structuring principle.

If a fault occurs during the processing of a request and the error caused by that fault is detected, an exception is raised within the IFTC. An exception handler is called when an exception is raised. If the IFTC can handle the error internally, then the fault is masked and activity returns to normal servicing. Error recovery can be forward or backward. Figure 8-2 shows how forward error recovery can be handled with an IFTC.

FIGURE 8-1 IDEALIZED FAULT-TOLERANT COMPONENTS (IFTC)

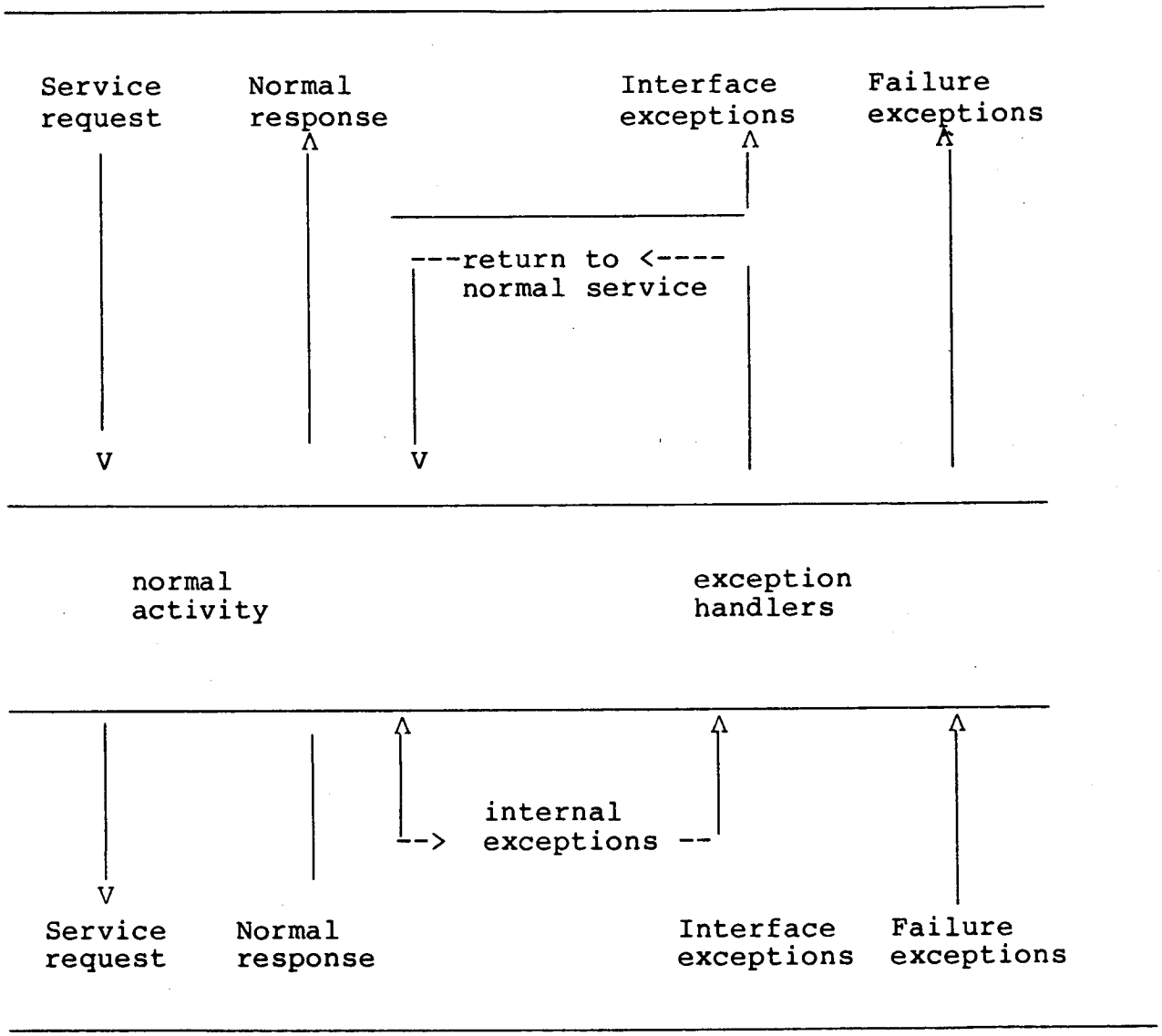


FIGURE 8-2 FORWARD ERROR RECOVERY

```
    [inconsistent_assertion -> forward_correct1];
```

or

```
    F[x ->forward_correct2];
```

For example

```
    [ b = 0 -> b = very_small];
    x := a/b;
```

or

```
    x := a/b  [zero_divide -> x :=MAX];
```

The statements between the brackets define the exception handler. The section before the "->" is either an assertion handler or a propagated exception. The statements after the "->" are the action portion of the exception handler. Internal checks for consistency or interface preconditions are made by assertions ("inconsistent_assertion" and "b=0" in Figure 8-2). Subcomponents can also return exceptions (subfunction "F" returns exception "X" and function "/" returns "ZERO_DIVIDE"). In the case shown in Figure 8-2, forward error recovery masks the fault and the computation continues normally. In some cases, as shown in Figure 8-3 the fault can be ignored.

FIGURE 8-3 IGNORING AN EXCEPTION

```
log_next_value(x) [couldn't_log -> ];
```

In this case, no action is taken in the exception handler, so the exception is ignored (in the example in Figure 8-3 it doesn't matter if some of the entries are not logged).

The interface specification for IFTC lists all the exception conditions which are propagated by the IFTC. Exception handlers must be provided for exceptions listed, or a default exception handler is invoked. This default handler implements an atomic action rollback. The default

exception "failure" can be invoked to respond to unanticipated exceptions. The exception handler can propagate the exception up the hierarchy using the "signal" command. Also available is the "reset" command which restores the internal state of the computation to that upon entry to the IFTC. Implicitly attached to every IFTC is the default exception handler shown in Figure 8-4.

FIGURE 8-4 DEFAULT EXCEPTION HANDLER

```
[ -> reset; signal(failure)]
```

8.2.1 Watchdog Processing and Idealized Fault-Tolerant Components (IFTC)

One aspect of fault-tolerant software in which treatment is not apparent in the above description of IFTC is the handling of time-outs. Actually, time-outs can be considered a special case of the class of watchdog processes. The problem in handling watchdog processes is providing the appropriate mechanisms for interaction within the hierarchy of IFTC. Let us concentrate the discussion on the problem of time-outs. The time-out activity is a concurrent process which can control the execution of the process being timed. In the case of a time-out, the timing process must be able to abort the execution of the timed process, restore the run-time environment to a consistent state, and return control to the exception handler. Semantically, a time-out should be treated like an assertion or acceptance test which failed. The implementation is quite different because of the nature of the concurrent monitoring by the timing process. In the case that the calling and called IFTC are on different processors or computers, extensive housekeeping operations must be performed to restore the system to a consistent run time state. Buffer queues may have to be flushed, late messages will have to be discarded, and the serving IFTC will have to be resynchronized. This housekeeping will be done by either the underlying support system and architecture, or by the appropriate mechanisms provided to the application layer.

The intended semantics of the aborted computation affect the nature of the restoration activity. If the computation was a procedure call, then the restoration might be to execute the "reset" command and signal the exception "time-out". If the computation was a remote procedure call, then the restoration would be to notify the executing processor of the requested abort operation and have the remote machine

abort the remote procedure, execute a "reset" command and signal the exception "time-out". However, because of the delays in communications, it may be necessary to simulate the propagation of the exception "time-out" from the remote procedure call in order to attempt appropriate action on the requesting processor in a timely fashion. However, this simulation can lead to the following situations:

- The remote procedure is aborted, state is reset, and time-out is propagated. The supporting system on the requesting processor will need to ignore the propagated "time-out" since it has already been simulated. It is required that the supporting systems be aware of at least some of the semantics of IFTC;

- The processor on which the remote procedure call is being executed fails, and the underlying system attempts to restart the computation on another processor. The request to abort the remote procedure will have to be redirected to the new processor. Redirection will require that the underlying system have the mechanism for resolving names in the presence of a changing hardware environment;

- An abort for time-out is requested but the remote procedure has already finished. The underlying system should recognize that this has occurred and ignore the effect of the remote computation. This may involve the use of time-stamps or other mechanisms to uniquely identify each request for service;

- Several processes are involved in the computation. This will be discussed in Section 8-4.

Under certain circumstances, resetting the state may be quite involved. If the aborted process is changing shared data structures, then is important to put these shared variables in a consistent state before tasking alternate action. The easiest way to insure the integrity of the shared data structures is to treat changes to them to be atomic actions. [CRIS82] The "reset" command of an IFTC will accomplish this. However, when aborting remote computations with a simulated "time-out" exception, it may be difficult to reset the shared data structure. If the requesting process tries an alternate computation, then the shared data structure must be reset and the "aborted" computation must not be able to access the shared data structure any more. Because of communications delays, assuring that the proper actions take place under the constraints of real-time deadlines can be very difficult.

One case in which access to shared data structures is desired is a critical section. The semantics of a critical section consist of mutually exclusive access to the shared data structure. A critical section should be considered an

atomic action on the shared data structure. Aborting a critical section requires that the shared data structure be reset to its initial state. The underlying system must support these semantics for aborting remote computations. If this cannot be guaranteed in the general case, then appropriate restrictions must be placed on the nature of remote computation to guarantee consistency.

Another concern with proper timing is whether the calling or called IFTC should specify the timing limit. In the case of a timed Recovery Block [K1978a], we would like to allow enough time to execute the alternate (or in some cases the primary). This statement implies we know how much time the alternate might use. If the programmer can place some limit on the number of loop executions (as could be done using Dijkstra's variant function for proving loop termination [DIJK76], then the compiler run-time support system should be able to estimate the time for execution. [WEI81] Thus, it may be possible to place loose but useful limits on the execution time of components. This timing would be a function of the called routine. However, in real-time systems, the reasonableness of the execution time may not be as important as the meeting of a deadline. The calling routine is usually in a better position to specify the deadline. Another argument for including the timing function with the calling routine is in the case of hardware failures where the calling and called routines are on different machines. Timing on the machine of the calling routine would allow for continued timely operation when the computer supporting the called routine failed.

In Figure 8-5 two possible methods of specifying timing constraints are given.

FIGURE 8-5 SPECIFYING TIMING CONSTRAINTS

Using Wei's syntax:

```
within one second do
     service next position.compute
     else next position.approximate
```

Integrating with previously defined syntax for exception handling:

```
next position.compute [not within (one second) ->
     reset; next position.approximate];
```

The first syntax was given by Wei in his dissertation. [WEI81]

Extending this mechanism to the general case of watchdog processes (such as control flow monitoring) seems straight forward but needs more investigation. While the semantics for watchdog processes is consistent with an assertion test (although done concurrently) and the syntax for handling exceptions from watchdog processes can be integrated into IFTC as shown above. The operational aspects of watchdog process are quite different. To execute a computation, the exception handler must be examined for those exceptions propagated by watchdog processes and those watchdog processes must be started concurrently with the execution of the monitored computation.

8.2.2 Communicating Processes and Idealized Fault-Tolerant Components (IFTC)

The difficulty of providing error recovery in the case of communicating processes is well known. [RAND75] However, several researchers have indicated that the rather simple and repetitive nature of real time control software may allow for simplifying restrictions to be placed on the nature of the communications. [ANDE83] The implicit recovery points provided on entry to an IFTC do provide a basis for defining conversations. When an error is detected, an exception can be propagated to all processes with which the current computation has had an exchange of information. These exceptions can be further propagated. Either some mechanism must exist for the computations themselves to pass on the exceptions, or the support system must keep track of the history of communications and automatically propagate these exceptions. If the underlying system keeps a history of communications it can also determine a consistent set of recovery points (a recovery line) dynamically.

8.3 Idealized Fault-Tolerant Components (IFTC) and the Major Fault-Tolerant Software Techniques

In this section we will try to indicate in what sense IFTC are a unifying mechanism for fault-tolerant software. In the last section, the use of IFTC for forward error recovery, atomic actions, ignoring faults, and time outs were discussed. In this section, the relationship between IFTC and the major proposed fault-tolerant software techniques identified earlier will be discussed.

8.3.1 Multi-Version and Idealized Fault-Tolerant Components (IFTC)

One of the advantages of multi-version fault-tolerant software is the possibility of using the technique transparent to the application programmer. Thus, each programming team is given the same specification to produce its

version. The specification accepts input from some driver and returns the output to the driver. The synchronization of the various versions and the consensus function is hidden in the driver itself. In the simplest case, except for the added requirement for the cross check points (cc-points), the programming of the versions itself requires no additional mechanism to handle the fault-tolerant aspects. However, in general, it will be necessary to restore the state of versions deemed in error. So the question of state recovery must be addressed. Also, if the individual version itself discovers an error, it should notify the driver so that account of this exception can be used in the consensus function. In addition, if the consensus function itself fails, it would be nice to handle this occurrence in a unified manner. Figure 8-6 shows how the driver for multi-version might be written using IFTC.

FIGURE 8-6 MULTI-VERSION AND IDEALIZED FAULT-TOLERANT COMPONENTS (IFTC)

```
PROCEDURE three_version (input:  IN input_t;
          output:  OUT output_t IS
        output1, output2, output3 : output_t;
BEGIN
        output1 := version1 (input);
        output2 := version2 (input);
        output3 := version3 (input);
        output  := consensus (output1, output2, output3);
END

PROCEDURE three_version (input:  IN input_t;
          output:  OUT output_t) [no_consensus] IS
        output1, output2, output3 : output_t;
BEGIN
        output1 := version1 (input) [->output1 := ignore];
        output2 := version2 (input) [->output2 := ignore];
        output3 := version3 (input) [->output3 := ignore];
        output  := consensus (output1, output2, output3,)
                 [no_consensus -> output = ignore;
                      SIGNAL (no_consensus)];
END
```

The first example shows a multi-version driver without exceptions while the second integrates exception handling.

8.3.2 Recovery Blocks

Upon entry to an IFTC a recovery point is automatically established. This recovery point provides the basis for state restoration (as was previously demonstrated in

the case of atomic actions). Upon normal termination of an IFTC the recovery point is discarded (commit point). The "reset" command causes the recovery point to be restored. This state recovery mechanism plus the use of assertions and exceptions allows an easy implementation of recovery blocks on the IFTC model. Figure 8-6 shows how this can be done.

Although only two versions are shown (the primary and one alternate), this could clearly be extended to an arbitrary number of versions. One unsatisfactory aspect of this particular syntax for IFTC is that the acceptance test is repeated in each version. This is poor and unsafe programming practice. Either the syntax could be changed or a preprocessor could be added to translate from recovery block syntax into the syntax shown (see also [SANT83]).

FIGURE 8-7 RECOVERY BLOCKS IN IDEALIZED FAULT-TOLERANT COMPONENTS (IFTC)

```
    PO [-> RESET; P1[-> RESET; SIGNAL failure]]
```

where

```
    Pi is BEGIN Pi [NOT post -> SIGNAL failure] END
```

for example

```
    compute_accurate_position[-> RESET;
       compute_approx_position[ -> RESET; SIGNAL failure]]
```

where

```
    PROCEDURE compute_accurate_position;
    BEGIN
         body1 [ absolute (old_pos - new_pos) > threshold
                   -> SIGNAL failure]

    END

    PROCEDURE compute_approximate_position;
    BEGIN
         body2 [ absolute (old_pos - new_pos) > threshold
                   -> SIGNAL failure

    END
```

8.3.3 Exception Handlers

IFTC use a single level terminating model for exception handling as advocated in [LISK79] and [CHRIS82]. Single level means that exceptions are propagated on only one level in the recursive hierarchy of procedures. In a terminating model for exceptions, the raising of an exception causes the current computation to be terminated. The exception handler cannot resume execution of the computation which caused the exception. This simplifies the semantics of exception handling. The use of exception handling separates normal flow of control of the base algorithm from the abnormal flow of control. It is felt that this separation is necessary to control the complexity of introducing fault-tolerant techniques into the software development process. While it may be possible in some cases to use the normal linguistic constructs of the language to deal with exceptional cases, it would be both inefficient and unasethetic (2). Linguistically, separate exception handling allows the normal algorithm to be specified in a straightforward fashion without cluttering it with the details of abnormal behavior.

Exceptions, properly implemented, allow for the handling of both anticipated and unanticipated faults. Implemented within the framework of a hierarchy of abstractions, exceptions allow faults to be made at the level where the error was detected, or propagated to the next higher level where the fault can be masked or propagated further up the hierarchy recursively. The exception handling provided in IFTC satisfies these objectives.

8.3.3.1 Robust Data Structures

Most modern languages provide the necessary mechanism for implementing the structural redundancy used in robust data structures. [TAYL80] However, it is useful to run audit programs outside the mainstream of the normal computations to periodically check the data structure for consistency. These audit programs can be run on special processors or they can be run in the slack time of a processor. Of course, they can be run as a diagnostic aid for damage assessment when the normal processing itself discovers an inconsistency in the data structure.

The audit programs for robust data structures can be run using a watchdog process employing the mechanism suggested earlier. An alternative is to separately schedule the audit programs and let them operate more or less independently.

---

(2) For example, testing for 0 values before performing division.

However, there would probably not be much point in auditing a data structure during a critical section. Also, when an audit program has discovered an inconsistency and has invoked error recovery procedures, normal processing must be denied access to the data structures under repair. The use of critical sections (with priority given to error recovery procedures) would handle this case.

8.3.3.2 Watchdog Processes

The general problem of implementing watchdog processes was discussed earlier. Watchdog processes represent a class of fault-tolerant software in which some abstract aspect of the computation is monitored for consistency by an external, possibly concurrent, process (observer). The audit programs for robust data structures, as well as time-out mechanisms, are special cases of watchdog processes. Another class of watchdog process which has received attention in literature is control flow monitoring. [YAUL80] and [AYAC79]

8.3.3.3 Run-Time Assertions

Run-time assertions represent a class of techniques in which various assertions are made regarding the validity of the computation. The use of types in programming languages is an example of one important subclass. Range checking for arrays is another example. Yet another is the specification of abstract data types [SHAW80]. Run time assertions can be used to check the input data validity (PRECONDITIONS), the consistency of certain international state variables, or the output validity (POSTCONDITIONS). The latter is used in recovery blocks to decide if an alternate procedure should be executed because of the supposed failure of the current computation. Validity checks for timing can also be made. As discussed, these validity checks are useful only for error detection; however, they can be coupled with the exception handling mechanism in IFTC to provide error recovery as well. Examples of their use has already been given above.

8.3.4 Hybrid Multi-Version and Recovery Blocks

8.3.4.1 Tandem

This hybrid technique is shown in Figure 8-8.

FIGURE 8-8 TANDEM TECHNIQUE AND IDEALIZED FAULT-TOLERANT COMPONENTS (IFTC)

```
PROCEDURE tandem (input:  IN input_t;
                  output: OUT output_t) [no_match] IS
          output1, output2:  output_t;
BEGIN
          output1:= version1 (input);
          output2:= version2 (input);
          output := match (output1, output2)
                    [no_match -> RESET
                    output1:= version3 (input);
                    output2:= version4 (input);
                    output:= match (output1, output2)
                              [no_match -> RESET;
                              SIGNAL (no_match)]];
END
```

In this case only one pair of alternates can be attempted. Assuming the language has a mechanism to specify a function name as a variable, the example could be more compact and easier to generalize.

8.3.4.2 Consensus Recovery Blocks

This technique is shown in Figure 8-9.

FIGURE 8-9: CONSENSUS RECOVERY BLOCKS

```
PROCEDURE Consensus_RB (input:  IN input_t;
          output:  OUT output_t) [no_consensus] IS
          output1,output2, output3 :  output_t;
BEGIN
       output1 := version1 (input) [->output1 := ignore];
       output2 := version2 (input) [->output2 := ignore];
       output3 := version3 (input) [->output3 := ignore];
       output  := consensus (output1, output2, output3)
                  [no_consensus -> output := output1;
                  [NOT post (output) -> output := output2;
                  [NOT post (output) -> output := output3
                  [NOT post (output) -> SIGNAL (no_consensus]
END
```

Here there is no use of state restoration.

8.4 Software Fault-Tolerance and Communicating Processes

The concept of IFTC works well if the program can be organized as a collection of hierarchically structured modules of the "right" size. The "right" size is a tradeoff between the cost of encapsulation and responsiveness to detected errors. Not all systems can be organized to use IFTC effectively. For example, a system with very little modular structure and/or large modules would have insufficient recovery points for timely error recovery. In this case, it may be necessary to insert recovery points manually. Also systems built from a collection of communicating processes may find it difficult or impossible to achieve fault-tolerance using IFTC. This is not unusual since processes provide a level of abstraction for building modular systems which is quite different from that provided by procedures. In this section, the problems of achieving fault-tolerance in a system of communicating processes will be discussed.

Most discussions of software fault-tolerance ignore the interaction of communicating processes. This is partly due to the limited use of communicating processes for most application software and partly due to the complications which are introduced when considering communicating processes in error recovery. Before discussing the effects which communicating processes have on error recovery, it is instructive to examine the typical uses to which communicating processes are put.

8.4.1 Communicating Processes

Although communicating processes have been the subject of much study within the computer science community for the past 20 years, their use in most application systems has been limited. In fact the programming languages in common use do not support the concept of communicating processes. The use of communicating processes introduces a number of complexities including proper synchronization, mutually exclusive access to shared resources, freedom from deadlock and liveness. However, the (perceived) need for communicating processes is increasing as is evidenced by their support in many of the more recent programming languages (Ada in particular). However, the nature of processes and the interprocess communications is still the subject of debate.

Several typical patterns of usage for communicating processes can be identified:

- Processes can provide for a modular structure very similar to that provided by subroutines. Thus, instead of making a subroutine call to provide some function, one process will communicate the request to the serving process. When the function has been com-

pleted the serving process communicates the results back to the requesting process. If the serving process retains no history of requests (between requests), then the usage is very much like a procedure process remains suspended while the server is active. In order to point out these similarities, several implementations of interprocess communications appear syntactically like procedure calls (for example, the rendezvous of Ada or Hoare's monitor calls). There are several reasons for choosing a process implementation over a procedure call even though the usage is quite similar. The first might be to provide a level of encapsulation around the server. This might be desirable since most systems do provide protection for processes but not for procedures. Another reason might be to provide a common service to several different processes;

- Competition for shared resources provide a second use for communication processes. Thus, one process could serve as the resource manager and all other processes which require use of that resource communicate their needs to the resource manager. (3) The requesting processes usually have no direct relationship to each other. This pattern of usage is common in operating system and data base system;

- The most general use of communicating processes is that in which several processes are cooperating on a more or less equal basis in the solution of some problem. The impetus for multiple processes here is to achieve better performance through the use of parallelism. The semantics of interprocess communications in this case is very much application dependent and can, in general, be very complicated. Some applications have a producer/consumer relationship between processes. Such processes can be chained together to form pipelines as in the case of UNIX'S pipes. In some applications all processes run the same code but on different portions of the data space (these applications lend themselves to simd architectures). Here communications are usually regular between "neighboring" processes. However, the results of one process can eventually propagate to all others. Still other applications require processes with different functionality to cooperate on the solution) these processes lend themselves to mimd architectures). The organization of these processes can be centralized around a master process or distributed.

---

(3) The resource manager need not be a separate process. Instead it could be a shared data structure accessed in a disciplined manner by the requesting processes.

8.4.2 Fault-Tolerance Applied to Communicating Processes

The difficulties of providing error recovery for communicating processes are well known. [RAND75] and [RUSS80] These difficulties occur because recovery may involve several processes. If an error is detected within one process, then all processes which have had (recent) communications with the faulty process are also suspect. Determining what constitutes recent communications is equivalent to establishing an encapsulation around a set of communicating processes for the purposes of fault-tolerance. While encapsulation for a single process is greatly facilitated by the use of procedures to define the modular structure of a program, similar linguistic mechanisms for inter-process encapsulation do not exist. Defining the boundaries of encapsulation for communicating processes must either be done explicitly in the program text of processes involved or must be derived from the history of communications which have occurred between processes.

The pattern of usage of communicating processes will effect the manner in which error recovery can be carried out. In the first usage pattern, error recovery is similar to that for procedure calls and can be implemented using IFTC. Thus, each return communication from a serving process can indicate normal termination or one of several exception returns. Although the implementation would be different, the semantics would be identical to that for procedure calls.

The second usage pattern in which a resource manager allocates a shared resource among several processes has been investigated in light of fault-tolerance. [SHIR78] Because of the independent nature of the requesting processes, failure in one process should not effect the activity of other processes. In this case recovery would take the form of compensatory calls to the resource manager to release those resources allocated by the failing process for the computation being abandoned. In this case it becomes necessary to keep track of what resources need to be released.

The third usage pattern, that of cooperating processes, is the most difficult for which to provide error recovery. One strategy is perform recovery on all processes which had communications with the faulty process since the last recovery point of the faulty process. This in turn may cause further propagation of error recovery. In the worst case all processes may have to be backed-up to their starting points. (These discussions are oriented towards backward recovery because of the generality and difficulty involved. However, forward error recovery also involves propagation of error recovery requests. In other words the only proper encapsulation involves all work done by all the processes. This uncontrolled propagation of error recovery has been termed

the domino effect. Clearly, such an effect could be disasterous, so several suggestions have been made which define more appropriate levels of encapsulation. One proposal is to encapsulate a group of processes and their communications into a "conversation". (4) This conversation becomes the basis for error recovery if an error is detected. One of the problems with this approach is that it requires the prior identification of conversations. Processes involved in a conversation must all pass their acceptance test and exit the conversation together. This can significantly reduce the amount of parallelism available in a system. This has the undesirable property of penalizing the non-faulty performance of the system. Also reasonable implementations of the conversation construct seem difficult.

Other researchers have suggested restrictions on the nature of inter-process communications in order to facilitate the definition of a suitable encapsulation for error recovery. One possibility is to limit communications to a producer/consumer relationship. [RUSS80] Another possibility is to define a recovery point every time inter-processes communications [HOSS 1983] However, more global levels of encapsulation still would need to be defined in order to deal with a transaction which spans several interprocess communications. Other researchers have noted that the relatively simple and repetitive nature of real-time tasks allows encapsulation to be defined around the processing which occurs within a frame, or time-slot. [ANDE83] The most appropriate determination of an encapsulation for cooperating processes is a matter for further investigation.

8.4.3 Multi-Processors

The issues of multi-processing and multi-computing are largely implementation issues and, as such, are orthogonal to the issues of multi-processing discussed above. However, care must be taken in implementing fault-tolerance in a multi-processor environment in order to avoid the effects of partial results possible when failures in processors and communications can occur. Some of these effects are discussed earlier. Inter-process communications in a multi-processor environment should use a protocol engineered for fault-tolerant applications. Work in this area is currently underway at UCLA. One of the requirements of a fault-tolerant protocol is to reject spurious communications and communications which arrive too early (due to a fault) or too late. Some sort of timing mechanism will be necessary. This timing mechanism will be necessary. This timing mechanism will be easier to provide if control is centralized.

---

(4) The concept of conversation is similar to the idea of a transaction.

In a transaction oriented system it may be necessary to tag each communication with a unique transaction number so that the boundaries of encapsulation can be more easily determined. This is an area which will need further investigation.

8.4.4 Conclusions on Communicating Processes

The general problems of providing fault-tolerance for communicating processes is very difficult and requires further investigation. However, it may be possible to restrict the usage of communicating processes to simplify error recovery. Whether or not these restrictions prove practical will depend on experience building systems under such restrictions. As of yet it is too early to decide which patterns of usage will provide appropriate mechanisms for implementing fault-tolerant systems using communicating processes. However, several scenarios can be suggested for incorporating fault-tolerance into communicating processes:

- Don't use communicating processes. This (trivial) solution may be suitable in many problems;

- Check all results before communications takes place. This checking can be performed by an acceptance test or a vote to insure correctness. The purpose of this test is to prevent the propagation of errors across process boundaries. This usage could utilize IFTC to provide fault-tolerance if communications only took place at the highest level of nesting in the hierarchy. Basically, this scheme assumes that each process appears fault free to its environment;

- Restrict process communications so that serving processes act like procedure calls. With the addition of exception returns from these process "calls", IFTC could be utilized for fault-tolerance;

- For processes which are in competition for resources, IFTC can be used with the following additions. All requests to acquire resources are noted. If back-up is necessary, the appropriate inverse requests are made to release these resources are available. For resources which are being released, defer the actual release until after commitment (unless the resource was acquired in the current computation). Keeping track of resource requests should be the responsibility of the underlying system in order to assure its correct implementations;

- Encapsulate a set of real-time processes to the communications which occurs during a frame. [ANDE83] If necessary discard all processing done during the current frame (skip frame);

- Restrict communications to producer/consumer relationships. Allow process to define appropriate recovery points for the purposes of encapsulation. Under appropriate conditions discussed in [RUSS80] proper encapsulation can be determined dynamically from the history of communications;

- Use a master coordinator for each conversation. The master receives the initial request, assigns a unique request ID, and dispatches the request (with its ID) to the appropriate processes. The results from each process are sent back to the coordinator. If any process or the coordinator detects an error, the coordinator transmits a failure message for that request ID to all associated process. It can then retry the request with different processes or take other appropriate action. If the request was satisfied correctly, then all associated processes are notified so that appropriate clean-up actions can take place. Because of the unique labeling of each request, a process could be working on several requests simultaneously. This scenario is speculative and would need further investigation.

The integration of fault-tolerance software into a set of communicating processes is a challenging task. Because of its complexity, this integration will most likely entail restrictions on the nature of interprocess communications. In order for this area to mature, further practical experience and research investigation is required.

8.5 Fault-Tolerant Software Primitives

From the above discussion several primitives required for the support fault-tolerant software can be identified. The support of these primitives have certain implications on the language and its supporting run-time environment.

8.5.1 Encapsulation

The importance of encapsulation was discussed earlier. It should be pointed out that the matter of encapsulation is one of degree and not of kind. All systems support some form of encapsulation and probably no method of encapsulation is fault-proof. Some forms of encapsulation can be supported by the compiler and therefore do not involve the systems architecture. However, the architecture should support computational abstraction at the level of reconfigurability. This means the encapsulation method used by the system should be appropriate for the component chosen for atomic actions. If procedures are chosen as the unit of software redundancy then the architecture should strongly support the abstraction of a procedure. In particular, this

means that the procedure must be encapsulated. If the unit of reconfiguration is a process, then this must be supported (encapsulated). For backup software, it may be necessary to support domains within the architecture. For instance the underlying system should make sure there is enough resources to support the critical domains, in the event of loss of hardware resources.

It is critical that the architecture support a strong encapsulation at the level of reconfigurable redundancy. Not to do so will make it very difficult to use fault-tolerant software techniques effectively. Without proper encapsulation, certain classes of faults will propagate outside the established boundaries of reconfigurability. These errors will then have to be handled at the outer levels and probably at greater cost. If the levels of reconfigurability are not properly nested, or if errors are not properly handled in the outer levels, then system failure will occur.

The other important aspect of encapsulation is the ability to detect violations of the intended design structure. For instance if a computation tries to access objects outside its defined domain, the architecture should detect this error and allow error recovery to proceed. Tagged memory, small protection domains and object oriented architectures are all ways to achieve increased encapsulation. The linguistic primitives used to achieve encapsulation are strong typing, support for abstract data types (private data types), well-defined visibility rules, process synchronization, monitors, etc.

8.5.2 Error Detection

Languages which require that redundant information be specified assist in the detection of errors. Many of primitives for encapsulation include redundant information which can be used in the detection of errors. Examples include strong typing and abstract data types. Subrange checks and array bound check also allow certain classes of errors to be detected. In addition, the run-time environment should check for the use of uninitialized variables and prevent the use of dangling reference (i.e. pointers to deallocated data objects).

8.5.3 Timing Primitives

The system should provide access to a high resolution clock for time-out purposes. It may be necessary to provide for several of these for different levels in the hierarchy. Also, for the specification of timing for scheduling alternate computations, it would be valuable for the language support system to assist in bounding the time for execution of a computation. [WEI81]

8.5.4 Recovery Point Primitives

Restoring a consistent state after the detection of an error can be very difficult. Many times the only sensible strategy is to roll the computation back to a previous (hopefully consistent) state and start from there. In the extreme, a cold start can be performed. However, less drastic rollbacks are desirable. The following primitives are needed.

8.5.4.1 State-Save Primitive

In order to establish a recovery point, the current state must be saved occasionally. Saving can be accomplished transparently to the application software by periodic checkpointing of the current state. One way to accomplish this checkpointing is to snapshot the entire state of the system; however, this is undesirable for two reasons. First, it would take considerable time to checkpoint the entire system. Second, and more importantly, there is no guarantee that the system state saved in the randomly determined manner will be consistent (i.e., is error-free). Snapshot checkpointing is good for detectable hardware failures. Even then it may be necessary to keep a record of the input values received since the last checkpoint to recovery the current state.

A more sensible state-saving scheme is to consider the semantics of the program when performing the checkpointing. The state will be saved only when the semantics dictate the system is in a consistent state. The application system can establish these recovery points through the use of explicit calls to the state-save primitive. Alternatively, the modular structure of the system can be used to implicitly determine the recovery points. Thus, the entrance to an IFTC establishes a recovery point. The assumption is that an IFTC is entered in a consistent state. This concept can be applied recursively throughout the hierarchy of a modular system.

8.5.4.2 Restore state

When an error is discovered it is necessary to restore the computation to a previous state using a "restore state" primitive. The "reset" command for IFTC accomplishes this function.

8.5.4.3 Commit state

At some point it becomes necessary to commit to the current state and discard the old saved state information. Commitment is necessary not only to keep the saved state

information from growing indefinitely, but also to reflect a commitment point in the computation that is dictated by the semantics of the program. Commitment can be accomplished either explicitly or implicitly. In IFTC in which implicit recovery points are made at the entrance to each module, implicit commit points are made on the (normal) exit of each module.

8.5.5 Exceptions

The architecture should use the same exception propagation technique as the application system. At least the architecture should propagate its unmaskable errors to the application in a manner consistent with the exception handling mechanism used in the application program and the architecture should allow the application to handle exceptions which might otherwise be handled by the system itself. Examples include memory faults, illegal instructions, etc. (exceptions which in most time-sharing systems would cause an abort of the offending program). Unless the underlying system is itself in an inconsistent state, the currently executing application should be allowed to attempt its own recovery.

8.5.6 Computation Control

For watchdog processes, fault-tolerant software will require the ability to abort a computation. In such cases it will be necessary to restore the system to a consistent state. Some of the implications of this were discussed above.

8.6 Supporting Idealized Fault-Tolerant Components (IFTC) in Ada and C

None of the existing popular programming languages were designed to support software fault-tolerant technology. However, with proper support from the underlying architecture, most languages can be used in implementing IFTC. In some cases it will be necessary to restrict the use of the languages where some of the languages features might compromise the reliability of the system. The amount of support required from the underlying architecture and the number of restrictions is a measure of the suitability of the language for implementing IFTC. This discussion concentrates on the programming languages Ada and C. Ada is a modern programming language oriented towards real-time embedded systems and, as such, would seem ideally suited for the environment under study. However, there has been little experience with the language and several noted computer scientists have questioned the reliability of the language. [HOAR81] and

[DIJK78] Thus, the study also investigates the suitability of a more mature systems implementation language - C.

8.6.1 Ada

The Ada programming language project has undergone an interesting development history, and is just becoming available for use in realistic software projects. As might be expected with any language of this size, it has been criticized. [HOAR81], [DIJK78], [MAHJ81], [SILB81], [LIND82], [JESS82], [BOUT80], [JONE80], [CLAR80], [SANT83], and [KNIG83] However, we will only concentrate on those features which effect the software fault-tolerance capabilities of Ada in a distributed real time environment. The major problems in the Ada language definition which complicate its use for implementing IFTC are in the areas of exception handling, specification and verification of timing constraints, and the rendezvous. In the following sections, we will examine the ability of Ada to support the fault-tolerant software primitives defined previously.

8.6.1.1 Encapsulation

Of the available programming languages for use in real-time distributed systems, Ada probably has the most comprehensive set of encapsulation mechanisms. Ada is a strongly typed language and allows for user-defined data types. It has the usual control encapsulations (procedures, functions, and tasks) as well as abstract data type encapsulations (packages). Ada's visibility rules are enforced across separate compilations.

Ada does allow some relaxing of its encapsulation rules. The ability to turn off some of Ada's strong typing must be carefully controlled if reliability is to be maintained. [Note that the relaxation of strong typing is usually done in the name of efficiency as, for example, when array bound checks are turned off. Using these run-time checks only for testing and turning them off during production is like wearing a life vest during training runs, but abandoning them for the actual mission. Errors in the real mission are much more critical. If efficiency is a concern, then provide hardware support for these run-time checks.]

In addition, the visibility rules will allow several tasks to access shared variables simultaneously. If access to these shared variables must be mutually exclusive, then the access procedures must be properly encoded. While such procedures can be written in Ada, they are not required. In both of these cases, a properly disciplined programmer can make use of Ada's encapsulation facilities in order to limit the possibility for system failures. However, since it is

possible to violate these encapsulation facilities, it would be advisable to use a preprocessor to enforce their use.

8.6.1.2 Error detection

Because of Ada's strong typing, the language itself offers many opportunities for error detection.

8.6.1.3. Timing Primitives

The semantics of Ada prevent two processes engaged in a rendezvous from executing simultaneously. Because of this the language provides no way for the calling process to terminate a rendezvous once it has begun (5). If a rendezvous has started and the machine which holds the serving process fails, then the calling process will wait in the rendezvous forever. Several researchers have recognized this problem and proposed a solution [KNIG83]. However, the problems of software failures have not been addressed. Some of the considerations concerning critical sections which must be taken into account were discussed earlier in section 8.2.1.

While Ada does allow some specification of timing constraints to be made, they are not generalized. For example, there is no way to time-out an on-going rendezvous. Also, it is not possible to time out an intra-process procedure call. Thus, timing violations within a process could not be detected. What is needed is a mechanism which is more general than is provided by the Ada standard.

8.6.1.4 Recovery Point Primitives

Ada contains no recovery point primitives.

8.6.1.5 Exception Handling

Many language researchers have argued for a single level terminating model for exception handling [LISK79] and [CRIS82a]. Ada does not support a single level exception discipline. Also, the state of the machine after an exception is undefined with regard to the values of the parameters (implementation dependent). Thus, in the case of IN/OUT parameters, the Ada run time support system can choose either call by reference or call by value with copy out of the updated value. Thus, the state of an IN/OUT

---

(5) There is a time out associated with a rendezvous, but it only checks that the rendezvous, starts within the specified time. Nothing is implied about finishing.

variable in the calling procedure when an exception occurs depends on the implementation chosen. In addition, Ada does not provide for atomic actions needed for IFTC. To overcome these problems, several researchers have proposed a restricted version of Ada that is capable of implementing IFTC. [SANT83] Called R_Ada, this version is implemented as a preprocessor to standard Ada along with the help of architecture a dependent Ada package providing for recovery points. The efficiency of implementing this package depends on the support of the underlying systems architecture.

8.6.2 The C Programming Language

The C programming language is tiny in comparison to standard Ada and will therefore require much more support from the underlying architecture. For example, C does not provide support for concurrent processing or exception handling. In a way this lack of support is beneficial, since we are at liberty to provide these features to closely support IFTC. Lee has implemented an exception handling package for the C language [LEE83]. Concurrency can be provided in any of a number of methods (such as Unix's fork and exec). Randell has used a network of UNIX system to explore issues in distributed fault-tolerant systems based on the UNIX operating system and the C language. [RAND83] This work has provided for IFTC within the UNIX UNITED project at Newcastle-upon-Tyne by providing support for atomic actions within each machine.

Another distributed UNIX/C project, the LOCUS project at UCLA, is focusing on the problems of multi-version consensus. The issue of late and early results from versions is being addressed. [MAKA83] Current work in fault-tolerant communications protocols is also underway. In terms of support for fault-tolerant software primitives, the C programming language provides weak support for encapsulation and error detection and no support for the other required primitives. However, this weakness does provide an opportunity to extend the language in a way to incorporate fault-tolerant software principles. While this work is still experimental, it does indicate the potential suitability for using mature systems (UNIX) and languages (C) to implement IFTC.

8.7 Unresolved Issues and Conclusions

Although the benefit of fault-tolerant software technology is not yet quantified, it is believed that these techniques can provide useful increases in reliability. Most of the major software fault-tolerant techniques can be integrated into a structuring concept called the (Idealized Fault-Tolerant Components) (IFTC). The major exotic hardware support for IFTC appears to be in the form of efficient

state-saving in the case of rollback to a previous recovery point.

The conceptualization of an IFTC has been shown to apply to the full spectrum of software fault-tolerant techniques and methods. In particular, it seems to provide a unifying framework from which the study and development of fault-tolerant methods can be accomplished. Utilizing this framework to examine the primitives required higher order languages in general we have exposed and discussed several common primitives and several key recovery primitives. Extending this work to a specific examination of the Ada and C languages reveals that Ada supports many of these principles, although some only partially. Full support of these fault-tolerant principles will require the abandonment of certain Ada features (a restricted use of Ada), as well as the development of additional system provided primitives (such as state recovery and generalized timing). The impact of these changes on the extensive run-time support provided by the Ada language is unclear. C, on the other hand, provides little, if any, support for these primitives. Additional system provided primitives will be required. However, the incorporation of these primitives into the run-time environment provided by the C language may be easier than for the incorporation into the Ada run-time environment. The feasibility and practicality of modifying the Ada environment is unknown.

The major unresolved issue is the methodology to be used to manage processes in the case of reconfiguration. In some instances it may be appropriate for the underlying architecture to restart processes which were on a failed processor. However, in the case of a major software failure (i.e., the system enters a region of the input space which consistently causes a version to fail), reconfiguration will have to take place under the guidance of the application program itself. Also, in the case where a significant portion of the computation resources are lost, it may be necessary to reconfigure the system to a minimal essential set of functions (related to the concepts of hardened kernel or back-up software). How such a set is specified and guaranteed to be operational is unresolved. It is imagined that this last problem can be solved by defining several domains and using a domain manager to control the execution of domains depending on resources. In order to be able to perform its task, this domain manager will require information concerning the availability of the hardware resources and some history of the failures of the software. It will also require the ability to abort processes and restart them (this implies some rebinding of the names of these processes will be necessary). It is recommended that further work be done in this area.

9.0 Effectiveness Assessment Methods

Methods for evaluating the effectiveness of software in highly reliable systems primarily consider reliability of the software [RAMA82, GEPH78, GOEL82] and in some cases performance (execution time). [GRNA81, SONE81]

Many models have been proposed for software reliability. The majority of these models were developed for existing fault-intolerant software. (Fault-intolerant software requires that software contain no faults of any type and perform precisely as specified. In other words, software faults are not tolerated and can cause system failure due to loss of a software implemented function.)

Seven studies [SUKE76, RAMA82, GEPH78, GOEL82, DALE82, JAME82, and SCHI78] have reviewed the many software reliability models developed for existing fault-intolerant software. The results of these studies and the specifics of each of those models will not be repeated in this report. These reliability models of fault-intolerant software are limited to attempting to estimate the reliability of single modules. However, most reliability models of fault-tolerant software assume the reliability of a single module as a given parameter and instead concentrate on developing the reliability of the fault-tolerant structure itself.

9.1 Synopsis of Models for Fault-Tolerant Software Reliability

This report presents information on seven identified models specifically developed for fault-tolerant software reliability estimation. The description includes assumptions inherent in the model and an analysis and evaluation on its applicability.

Table 9.1 identifies the fault-tolerant software models that were uncovered in this study. It identifies the last name of the developer and the fault-tolerant software methods to which the model(s) are applicable.

9.1.1 Granrov, Arlat, and Avizienis Models

A set of models for evaluation of the two primary software fault-tolerance strategies, recovery blocks and N-Version programming, was developed by Grnarov, Arlat, and Avizienis. [GRNA80, GRNA81] The set consists of a general model and specific models for the two strategies. The models generate estimates for processing times and reliability.

TABLE 9.1 COMPARISON OF FAULT-TOLERANT SOFTWARE RELIABILITY MODELS

Fault-Tolerant Software Techniques Modeled / Modeler / Type	Grnarov Avizeinis: Stochastic	Migneault: Probabilistic	Scott: Probabilistic Input Domain	Sonerlu: Time Domain	Wel: Probabilistic Structoral	Mellar Smith: Probabilistic	Bhargava: Probabilistic
Recovery Block (RB)	X		X	X	X	X	X
N-Version Parallel (NVP)	X	X	X	X	X		
N-Version Sequential (NVS)	X			X	X		
Tandem				X	X		
Consensus RB			X		X		

The general approach is based on queueing theory and examines the processing of a single "segment". In the recovery block scheme, a segment consists of a primary alternate of the function to be performed, a set of supplementary alternates and an acceptance test. In the N-Version programming technique, a segment is a set of $N \geq 2$ independently designed programs (versions) that perform the same function.

9.1.1.1 Assumptions

Assumption 1. The execution time of the segments is exponentially distributed.

Assumption 2. The "bad" versions are recovered by updating them with data provided by identified "good" versions and normal processing of all N versions can be continued.

Assumption 3. A recovery block configuration using N-1 alternates and the acceptance test can be compared with an N-Version scheme using N-Versions.

Assumption 4. The failure rate of a software module is assumed to be proportional to its execution time.

Assumption 5. The recovery block scheme has a "safe down and repair" state.

Assumption 6. In the recovery block scheme, the coverage provided by the acceptance test is a function of the ratio of the acceptance test execution time and the segment execution time.

Assumption 7. In the recovery block scheme the density of faults in the acceptance test is the same as in the alternates.

Assumption 8. The recovery block scheme will fail only if the acceptance test fails.

Assumption 9. In the model for sequential N-Version scheme, agreement of two versions out of N is sufficient to determine a correct result.

9.1.1.2 Analysis/Evaluation

The Grnarov models exhibit some innovative thinking in terms of approaches to modeling software fault-tolerance strategies. For example, the models include factors to account for correlated faults. However, the models have two major problem areas, assumptions and data.

The exponential distribution assumption (assumption 1) implies that a segment will have many execution times that are relatively short and a few execution times that are relatively long. This assumption may be reasonable for the recovery block scheme since the alternates are processed sequentially and the number of alternates processed depends on the comparison of their results with the acceptance test. While the exponential distribution would allow a processing time to be arbitrarily small, the smallest actual processing time would be the sum of processing times for the primary alternate and the acceptance test with a correct result. If this sum is small relative to the largest possible processing time (such as, if only one or two alternates exist), then the exponential distribution may be appropriate; otherwise, a different assumption may be required.

The processing time for one segment implemented with the N-Version scheme, however, should be fairly constant since all versions will be executed. It is doubtful that the exponential distribution is a realistic assumption for processing times in the general N-Version programming concept. The assumption may not be unreasonable for the sequential N-Version scheme described in [GRNA80]. In this scheme, as each successive version is executed, its results are compared with the results of the preceding versions and a 2-out-of-N decision is used (i.e., as soon as two results agree, that answer is assumed to be correct). Hence, the number of versions executed will be a random variable depending on the faults in the versions and the input data.

Regarding the second assumption, it certainly may be possible to recover those failed versions in which the failure was caused by inappropriate processing of a particular, limited class of input data. However, if the fundamental algorithm or logic in a version is flawed, correcting the data for the next iteration of the segment will not correct the fault in the flawed version. Furthermore, if one of the versions maintains unique local state information, then it may not be correctable on the basis of data from the other versions.

Implicit in the third assumption is the idea that having N software modules in each scheme somehow makes them "comparable". The authors may have assumed that this same number of modules implied equivalent costs or performance. However, the complexity of the modules may be different. Some of the recovery block supplementary alternates may implement simpler, less accurate, and faster versions of the algorithm used in the primary alternate. In the N-Version scheme, each version is independently designed. The versions may exhibit a range of complexity and costs. Also, the overhead required to perform comparison among the N-Versions is not addressed. The idea that having N modules in each scheme makes the schemes comparable is not justified.

Assumption 4 in general probably is not true since it does not take into account the complexity of the module, the ability of the personnel who designed and programmed the module, or the testing process used in development of the module. Since the Grnarov Model is concerned with specific schemes oriented towards achieving high reliability, it is reasonable to consider modules that implement algorithms of similar complexity, and which are developed in a similar, high-quality environment. In this case, the assumption may be valid for population of modules but invalid for comparing two particular modules. Therefore, this assumption does not provide a basis for comparing recovery block modules with N-Version modules.

The validity of assumption 5 could depend on the particular implementation of the recovery block scheme. Removing this state would decrease the reliability results in the referenced papers.

Assumption 6 asserts that a longer acceptance test will provide higher coverage. While there is some intuitive appeal to this notion, no supporting rationale is provided in the references.

The definition of minimal coverage is vague. The numerical examples in [GRNA80] and [GRNA81] use 0.90 as the smallest value of coverage, but no reason for selecting this value is provided. Perhaps it could be argued that the acceptance test should be at least this good to make the recovery block scheme feasible.

Regarding assumption 7, if the acceptance test is constructed as an abstract implementation of the function performed by the alternates, then this assumption may be true. However, a different approach could cause the assumption to be invalid.

Assumption 8 will be true if there is some global recovery method to handle the event that the acceptance test does not fail, but all alternates fail, or an alternate has an infinite loop. If no such method is used, then the model would need to be modified.

The last assumption represents only one of several possible ways to compare the results of the multiple versions. For example, another approach would be to discard any results that are highly different from the others and then average the remaining results.

The choice of the comparison method may affect the reality of the N-Version approach. Grnarov's Model for the sequential N-Version scheme is limited to the 2-out-of-N comparison method.

There are enough difficulties with the assumptions in the models to seriously question the value of using the models to compare the recovery block and N-Version programming schemes. The model for each scheme may be useful for investigating the effects of different values for the various input parameters, but justification for comparing schemes is not present in the references.

Two aspects of the input data are of concern. From a global perspective, no approach for estimating some of the data items exists. Examples include average repair rate, probabilities of correlated faults, and success probabilities of individual modules. The second concern relates to the numerical results presented in [GRNA80] and [GRNA81]. Many of the inputs required to perform the computations are not given in the references. In addition, the realism of some of the numbers (such as minimal factor for the recovery block acceptance test) is questionable. Accordingly, the results in the references should be considered preliminary and interpreted with caution.

In general, two additional factors should be added to the Grnarov Model, a factor for the probability of a failure in the recovery block model and a factor for the the probability that the "bad" versions are correctly updated in the N-Version parallel case.

The Grnarov Models have not been validated with empirical data.

9.1.2 Migneault Cost Model

The cost model presented by Migneault in [MIGN82] uses computations of system unreliability caused by software faults to compare the cost of different levels of software fault-tolerance. Migneault points out that the equations given in [MIGN82] represent a "first attempt" at developing a system level of view of the relationships among cost, redundancy, and reliability. This review discusses the key assumptions in the model that must be considered to interpret the equations and results.

9.1.2.1 Assumptions

Assumption 1. Given that k faults have been removed from a properly developed module, the subsequent failure rate of a module is:

$$\lambda_k = 3600me^{-(a+bk)}$$

where

m = number of module executions per second

a,b = model parameters.

Assumption 2. The cost to develop a module to the point of acceptance is constant regardless of whether the module is developed for a single-string logic application or for a fault-tolerant (e.g., N-Version) application.

Assumption 3. For an N-Version programming scheme, errors in one program module-voter module (PV) pair are independent of errors in another PV pair.

9.1.2.2 Analysis/Evaluation

The model and results in [MIGN82] are useful from the perspective of prompting thoughtful consideration of the relationships among software costs, reliability, and redundancy. However, sufficient questions about assumptions used in the model exist to indicate that the results should be considered preliminary and approximate. Variations in the assumptions and parameter values could change the results.

Several concerns exist about the validity of assumption 1. The idea that removing more faults (i.e., a higher value of k) results in a lower failure rate is intuitively appealing. The observed failure rate of the module depends on the existing faults in the code and the input data. If the input data do not exercise any faulty portions of the module, then no error will be observed. The sets of input data exercised in the identification of k faults will encounter the module faults at some frequency. Those k faults are then removed. However, there may exist n additional faults that were not identified. If subsequent input data are similar to the data used to identify the k faults, then the observed failure rate probably will be lower. However, if the subsequent input data are substantially dissimilar to the original data, then the subsequent input data may encounter the remaining n faults with a high frequency and the observed failure rate could be higher.

The equation for λ_K assumes a fixed level of quality in the development of a non-debugged module. In reality, a better (and possibly more costly) development process would result in a lower failure rate.

Regarding the second assumption, the effort devoted to development of a module may be a function of the intended application of the module (single string versus redundant logic). It may be more meaningful to compare costs and reliability in which the initial development effort reflects the nature of the intended application.

The third assumption also presents some problems. All PV pairs are developed from the same specification. Therefore, they could contain common logic problems associated with errors or uncertainty in the specification. Migneault recognizes the correlated faults among redundant modules may be a problem, and that the assumption of independence of errors is made to develop a comparison.

The model should be refined and extended to include recovery blocks. This model has not been validated.

9.1.3 Scott Reliability Models for N-Version Programming, Recovery Block, and Consensus Recovery Block

These data domain models in which the reliability estimate is time independent were developed by R. Keith Scott while at North Carolina State University [SCOT83a and 83b] in his PhD dissertation [under the direction of Drs. James W. Gault and David F. McAllister].

"These models will give a user the ability to estimate reliability improvements provided by each method as a function of the reliability of the software components that comprise the system." [SCOT83a]

Scott developed separate models for the different fault-tolerant techniques. Three of these are discussed in this report: the N-Version programming, recovery block and consensus recovery block model. Furthermore, Scott's models are the only models found which have been at least partially validated through experimentation.

These models are based on computation of probabilities related to various error types. Table 9.2 shows the four recovery block error types, Table 9.3 shows the three N-Version programming error types. Figure 3-5 in Chapter 3 depicts the consensus recovery block model. Table 9.4 defines the event outcomes and their associated probabilities for the recovery block.

TABLE 9.2 SCOTT'S RECOVERY BLOCK MODEL ERROR TYPES

Type	Definition
1-Incorrect but accepted	Program version produces an incorrect result and the acceptance test accepts incorrect result as being correct.
2-Correct but not accepted	Final version produces correct result and the acceptance test rejects the correct results.
3-Unsuccessful State Recovery	Recovery program cannot successfully recover the input state of the previous alternate in preparation for executing another version or cannot successfully invoke the next version.
4-Incorrect and not accepted	Final version produces incorrect results and the acceptance test judges that the results are incorrect. (There are no more alternate versions to execute and the fault-tolerant recovery block has failed).

TABLE 9.3 SCOTT'S N-VERSION PROGRAMMING MODEL ERROR TYPES

Type	Definition
1	A Type 1 error occurs when all outputs, comparisons disagree.
2	A Type 2 error occurs when an incorrect output occurs more than once. (1)
3	A Type 3 error occurs when there is an error in the voting procedure.

The probability of a system software error is then a sum of the three probabilities of the Type 1, 2, and 3 errors.

(1) Correct outputs can still be generated if a Type 2 error occurs (e.g., in a majority vote of 5 inputs, if 2 of the inputs agree and are incorrect and if the other 3 agree and are correct, then resulting output will be correct). However, Scott's Type 2 errors generally refer to output errors caused by conditions where a majority of inputs are incorrect, causing an incorrect output value to be computed.

TABLE 9.4 SCOTT'S RECOVERY BLOCK MODEL PROBABILITIES

$P(C_i)$	= Probability of version i producing a correct output for a correct input
$P(I_i) = 1-P(C_i)$	= Probability of version i producing an incorrect output (for a correct input)
$P(C_R)$	= Probability of the recovery program executing correctly (input state of the previous recovered and the next version invoked).
$P(I_R) = 1-P(C_R)$	= Probability of the recovery program executing incorrectly
$P(A_I)$	= Probability of accepting an incorrect result
$P(R_I) = 1 - P(A_I)$	= Probability of rejecting an incorrect result
$P(R_C)$	= Probability of rejecting a correct result
$P(A_C) = 1 - P(R_C)$	= Probability of accepting a correct result
$P_{RB}\ (E_{k,n})$	= Probability of type k error given n alternate versions

9.1.3.1 Scott's Recovery Block Model

9.1.3.1.1 Assumptions

Scott initially presented his model for the recovery block method under the general assumption that the versions are statistically independent. [SCOT83a]

Assumption 1. The versions are developed independently by different designers and programmers,

Assumption 2. The probability of a correlated error among the versions is negligible.

Scott follows this initial model with a dependent form of the model in which the previous assumption of statistical independence is relaxed.

Assumption 3. The alternate program versions are dependent, but no dependence exists between the versions and acceptance test. [SCOT83a]

9.1.3.1.2 Analysis/Evaluation

The independent version model, before any simplifying assumptions are made, appears relatively simple and easy to use. The difficulty, as with all models, lies in determining accurate values of the inputs. Scott attempted to estimate the reliability of the individual versions of sixteen acceptable programs, written by sixty-five students, using Bernoulli trials [SCOT83a and 84] on fifty input test cases drawn from a set of 100 test cases. A number of problems were encountered in selecting the fifty reliability estimation test cases and the 50 model verification test cases due to biased estimation. The problem was solved by assigning difficulty factors to each input test case and then putting equally difficult test cases in each of the two pools. This approach resulted in an estimate of the reliability and variance for each of the sixteen programs. [SCOT83a and 84]

Validation of the software reliability models for the recovery block method was based on the initial assumption that the acceptance test program contained no errors and therefore, $P(A_C)$ and $P(R_I)$ equal 1.0. In all cases Scott assumed perfect recovery could be achieved between detection of the fault and execution of the next version, i.e., $P(C_R) = 1.0$.

(There is a statement in Scott's dissertation that a recent study reports about one half of software errors discovered in supposedly independent versions were common errors in all versions.)

Scott relaxed the assumption of perfect acceptance tests and computed the predicted reliability for a 3-version recovery block using various values (0.99, 0.95, 0.90, and 0.75) of acceptance test reliability. The results of this test force the rejection of the null hypothesis in favor of the alternate hypothesis - "the model could not predict system reliability". [SCOT83a]

The null hypothesis that "the dependent model can predict reliability" of the recovery block was accepted using the assumption of perfect recovery. [SCOT83a]

Finally, Scott demonstrated that there is a reliability improvement of a recovery block system over single version

software even with acceptance test reliability as low as 0.75 at a 95 percent confidence level as shown in Table 9.5.

TABLE 9.5 SUMMARY OF THE RELIABILITY IMPROVEMENT OF A RECOVERY BLOCK SYSTEM OVER A SINGLE VERSION

NULL HYPOTHESIS		REJECTION	
		.95	.99
NO RELIABILITY IMPROVEMENT OVER A SINGLE VERSION	ACCEPTANCE TEST = 1.00	YES	YES
	ACCEPTANCE TEST = 0.99	YES	YES
	ACCEPTANCE TEST = 0.95	YES	YES
	ACCEPTANCE TEST = 0.90	YES	YES
	ACCEPTANCE TEST = 0.75	YES	NO

9.1.3.2 SCOTT'S N-Version Programming Method

9.1.3.2.1 Assumptions

Assumption 1. The probability of a Type 2 error is 0.

Assumption 2. The probability of a Type 3 error is 0.

Assumption 3. There exists one and only one solution for a given set of inputs.

9.1.3.2.2 Analysis/Evaluation

Regarding Scott's assumptions, assumption 1 is a reasonable assumption if the versions have independent errors. However, the assumption that there are no common faults among the N-Versions is theoretically correct but, in fact, is a bad assumption, as previously demonstrated by Scott in his software model and tests for the recovery block.

Scott's assumption that there exists one and only one solution for a given set of inputs is based upon the comparison of outputs from each version being exact. This is a very bad assumption, since in the real world the comparison model does not normally require an exact vote. Instead, the outputs of each version are often checked against boundary variables and all version inputs which fall within the legal

range can then be mathematically manipulated to arrive at the voted or correct output. For example, in the case of three outputs, the algorithm could average the three inputs or could take the middle value as the correct output. In the case of two versions in which the inputs fall in the legal limits, the comparison algorithm often takes an arithmetic average to compute the output.

Based upon the assumptions made by Scott, the system reliability model is the probability of two versions reaching similar, correct conclusions upon execution. This results in his reliability model for N-Version programming being the same model as that for two-of-n redundant hardware system.

Scott repeated the experiments in another attempt to validate the models with the Type 2 errors no longer assumed to be 0. Table 9.6 summarizes the results of his experiments.

TABLE 9.6 SUMMARY OF N-VERSION PROGRAMMING FINDINGS

NULL HYPOTHESIS			REJECTION	
			.95	.99
NO RELIABILITY IMPROVEMENT OVER SINGLE VERSION			NO	NO
CAN PREDICT TYPE 2 ERRORS			NO	NO
MODEL CAN PREDICT RELIABILITY	INDEPENDENT	NO TYPE 2 ERRORS	YES	YES
		TYPE 2 ERRORS	YES	YES
	DEPENDENT	NO TYPE 2 ERRORS	YES	YES
		TYPE 2 ERRORS	YES	YES
TYPE 2 ERRORS HAVE NO EFFECT ON RELIABILITY PREDICTION			YES	YES

"A number of observations can be made. First, the occurrence of Type 2 errors could be predicted accurately, but their occurrence did not affect the success of reliability prediction. The reason for this is that each test case had the possibility of multiple correct answers. The definition of "success" in an N-Version programming system is the agreement on a (presumably) correct output. The programs could provide a number of different correct solutions to each input test case, but an N-Version program-

ming system would not recognize an output as correct unless it occurred more than once. Therefore, the predicted reliability, which assumes unique correct outputs, was consistently greater than the estimated reliability. This leads to the final observation: an N-Version programming system is useless in situations where multiple correct outputs can occur. In fact, as in the case of our experiment, the reliability of an N-Version programming system may be less than any of the component programs." [SCOT83a]

9.1.3.3 Scott's Consensus Recovery Block Method

9.1.3.3.1 Assumptions

Scott developed two models, the first based upon the assumption of version independence, and the second model based upon the assumption that there was dependence among the versions.

Assumption 1. The N-versions of a program task can be developed such that no common software faults exist among the versions. (Independent Model). In the case of the dependent model, Scott relaxes this assumption.

Assumption 2. The probability of a Type 5 error is 0.

9.1.3.3.2 Analysis/Evaluation

These models are based upon computation of probabilities related to the four error types previously depicted in Table 9.2 for recovery blocks plus a Type 5 error. The N program versions were executed and their outputs submitted to a voting procedure. If two or more of the versions agree on one output, that output is designated as correct. The Type 5 error occurs if the consensus output is incorrect.

Scott conducted an experimental verification of the consensus recovery block models.

In his experimental test of both the independent and dependent models for the consensus recovery blocks, Scott tested the second assumption. Scott found:

"For a consensus recovery block composed of dependent components, it is not possible to assume that the probability of Type 5 error, agreement on an incorrect output, is zero. This positive parameter must be estimated...complexity of the model and the difficulties in estimating the dependent, conditional probabilities may render the dependent model useless in practical situations." [SCOT83a]

From his tests, Scott also concluded that the independent models cannot accurately predict a consensus recovery block reliability, whereas the dependent model can predict the consensus recovery block reliability.

Scott also conducted an experiment to determine if the consensus recovery block gave a reliability improvement over a single version. This experiment was conducted based upon probability of a successful acceptance test ranging from 0.75 to 0.99. Table 9.7 contains results of this experiment. Note that for an acceptance test probability equal to 0.9, the null hypothesis can be rejected at the 95 percent confidence level but cannot be rejected at the 99 percent confidence level. At an acceptance test probability equal to 0.75 the null hypothesis cannot be rejected at either the 95 or the 99 percent confidence level. This indicates that the reliability of the consensus recovery block system may, in fact, be less than that of the single version in systems with a poor acceptance test.

TABLE 9.7 SUMMARY OF THE RELIABILITY IMPROVEMENT OF A CONSENSUS RECOVERY BLOCK SYSTEM OVER A SINGLE VERSION

NULL HYPOTHESIS		REJECTION	
		.95	.99
NO RELIABILITY IMPROVEMENT OVER A SINGLE VERSION	ACCEPTANCE TEST = 1.00	YES	YES
	ACCEPTANCE TEST = 0.99	YES	YES
	ACCEPTANCE TEST = 0.95	YES	YES
	ACCEPTANCE TEST = 0.90	YES	NO
	ACCEPTANCE TEST = 0.75	NO	NO

9.1.4 Soneriu Discrete State Continuous Time Markov Model

Soneriu's model is a discrete state continuous time Markov model. The model considers N-Version programming and recovery blocks. In addition, Soneriu introduces a hybrid method, the Tandem Method. The Tandem Method is shown in Figure 3-4 in Chapter 3. The model has been developed for a fault-tolerant machine consisting of a set of M subsystems with each subsystem made up of N modules. A subsystem provides one or more fault-tolerant operations. In order for a fault-tolerant machine to survive in the presence of faults, subsystems must remain operational.

"A completely fault-tolerant machine must maintain all its subsystems operational at the specified rate for any faults. In a partially fault-tolerant machine with non-degrading services, all subsystems must remain operational at the specified rate for certain faults, while in a partially fault-tolerant machine with degrading services, some subsystems may decrease their rate of execution or even become unoperational in the presence of faults." [SONE81]

The goal of the Soneriu method is to estimate the reliability of a partially fault-tolerant machine as compared to that of a completely (ideal) fault-tolerant machine.

9.1.4.1 Assumptions

Assumption 1. The first assumption that Soneriu makes is that the software modules are a collection of the independent modules.

Assumption 2. Each Module is either in an "up" state or a "down" state. The system reliability is then based upon the number of members in the up state.

Assumption 3. The modules have exponentially distributed failures.

Assumption 4. The module recovery and repair times are exponentially distributed.

9.1.4.2 Analysis/Evaluation

Soneriu's Model, in its present state, cannot be directly applied unless one can provide realistic estimates based upon another model or historical data for the input parameters.

These constraints would specifically require that the user be able to estimate the coverage provided by the actual recovery algorithm for the recovery blocks, (i.e., coverage

provided by the acceptance test) and for N-Version programming, (i.e., coverage provided by the comparison algorithm). In addition, the user must specify the repair rate of each of the modules, the number of modules, and the time of interest.

Hypothetical values could be assumed and used in the Soneriu model in a manner similar to that used by Scott. This approach would be useful to gain initial experience in using the model. It must be supplemented by estimated values at some subsequent point.

9.1.5 Wei Reliability Model for Fault-Tolerant Software System

Wei's model [WEI81] provides a basis for decomposing processes into segments which can represent individual software functions such as algorithm version, fault-detection, etc., and computing the resulting process reliability based upon the reliability of the segments and the probability of their execution. Wei's model can be applied to recovery blocks or N-Version programming (two out of three majority voting) as well as consensus recovery blocks [SCOT83a,84] and the Tandem Method [SONE81].

Wei assumes a real-time system consisting of M(>0) periodic processes with request periods t_1, t_2, ..., t_M where $t_1 < t_2 < \ldots < t_M$ and t_{i+1} is a multiple of t_1 for $i = 1, 2, 3, \ldots, M-1$. The failure probability for process i is f_i. The probability of total system failure can then be approximated by:

$$F = \sum_{i=1}^{M} n_i * f_i \quad \text{where} \quad n_i = \frac{t_M}{t_i} .$$

Wei then looks at the segments that compose the processes in an attempt to express the overall probability of failure F as a function of the individual failure probabilities for the segments of process i.

To do this, he introduces e_{ij}, the conditional probability that segment j will be executed during the execution of process i and examines the probability of failures along each of the paths (or sequence of segments) that comprise the process. He then shows that the probability of failure of process i can be approximated by :

$$f_i = \sum_{j=1}^{N_i} e_{ij} * f_{ij} \quad \text{where } N_i \text{ is the number of segments}$$

and then

$$F = \sum_{i=1}^{M} n_i \quad \sum_{j=1}^{N} e_{ij} * f_{ij}.$$

In other words, Wei shows that the probability of failure of any process can be approximated by the linear sums of the failures of the individual segments of the process over the conditional probabilities that the segments will be executed.

Wei then uses this model to examine reliability of recovery blocks and majority voting.

9.1.5.1 Assumptions

Assumption 1. A real-time software system could be expressed as a set of periodic processes and each process's request period is a multiple of the next smallest request.

Assumption 2. Each process in a real-time system can be further decomposed into a set of segments in a structured way. Three basic structures are used extensively for the refinement, i.e., sequence, if-then-else, and while loop. Wei assumes that process i contains N_i segments of which each has failure probability f_{ij} where $j = 1,2,\ldots,N_i$.

Assumption 3. There are two alternates in the recovery block and that in a majority vote the probability of failure of the voting component is 0.

9.1.5.2 Analysis/Evaluation

Wei's model appears to be useful in the design of software. Because Wei's model captures the conditional probabilities that each segment will be executed given the fact that the preceding segment has been executed, the model is very representative of the actual process of software execution.

Also, because the model examines the paths of segments internal to processes, the critical paths can be identified, (quantatively paths with large $n_i * e_{ij}$) and segments with "high-execution probability residing in a frequently executed process" can be designed with greater fault-tolerance.

One extension to the model would be to develop a method to accurately estimate the failure probability of each segment in a path. This would include not only the primary and alternate version, but also the acceptance test in the case of the recovery block.

In its present status, the model is useful in evaluating tradeoffs between recovery blocks and majority voting concepts. The model should be used in an experiment in order to further verify its assumptions. Since many of the higher order terms were discarded in the linear approximation, the experiment would test this use of the linear approximation.

9.1.6 Melliar-Smith Probabilistic Model of the Reliability of Recovery Blocks

Peter Michael Melliar-Smith developed a probabilistic analysis of nested recovery blocks. [MELL83] The model is only applicable to recovery blocks and includes some analysis of the effect of correlated errors. The model estimates the probability of obtaining an erroneous result or probability of an error return using nested recovery blocks.

9.1.6.1 Assumptions

Assumption. This model assumes complete independence between the alternates and the acceptance test, and between the various alternates.

9.1.6.2 Analysis/Evaluation

Melliar-Smith's model for the recovery block is similar to that developed by Scott for the recovery block.

The user of the model must furnish the probability of error associated with each version, as well as the probability of the acceptance test not detecting any errors that have been submitted to it. The classic problem with these types of models is that the developer exercises the model using hypothetical data, just as Scott has done.

Regarding the assumption, Melliar-Smith states: "It would be very desirable to be able to include the correlations between those programs and the model, but at present we do not know how to measure, to express, or to analyze the correlations between programs. We will consider the problem further, but a conceptual breakthrough is required and there can be no expectation of a quick solution." [MELL83]

Melliar-Smith extended the model to include the effect of correlated faults due to design error. This extension formed the basis for his conclusions. These are:

Recovery blocks should substantially improve the reliability of new or little tested programs, because many of the coding faults will be random and will be caught by the acceptance test. However, in well tested operational programs, such as flight-critical functions, recovery blocks

will show less improvement in reliability because, Melliar Smith hypothesizes, the program faults will tend to be correlated between the validation procedures and the acceptance test. Most random faults will normally have been detected by the validation procedures and during operation. Because the acceptance test will normally be correlated with the validation process, program faults will tend to escape detection.

"If recovery blocks are to substantially improve reliability, it must be because almost all of the residual faults left in the operational program are faults that are highly correlated to the validation procedures but uncorrelated, or only partially correlated to the acceptance tests and alternatives. Since the validation procedures and the acceptance tests are already highly correlated, it will be hard to argue that such faults predominate.

"...this analysis shows that recovery blocks do not, by themselves, provide a reliability improvement sufficient to meet the stringent reliability requirements of flight-control applications. However, once an initial version of the programs (of adequate reliability) has been obtained, recovery blocks can be used to allow some modification and enhancement of programs without loss of reliability." [MELL83]

9.1.7 Bhargava Software Reliability Model for Recovery Blocks

Bhargava developed a probabilistic model for recovery blocks. [BHAR81] The model is based upon computing the probability of reliability in four types of software: A primary module with an acceptance test; a primary module decomposed into a number m submodules, each with its own acceptance test; a primary module with a number n-1 of alternatives; and, a primary module composed of m submodules, each with its own acceptance test and alternative. He then computes the probability of failure for each of these types.

Bhargava's basic purpose is to look at the tradeoffs involved in implementing a recovery block system and specifically to examine the effects of increasing the number of alternates, the testing granularity and module structure relative to achieving a specific reliability within a constrained cost for acceptance testing. [BHAR81]

9.1.7.1 Assumptions

Assumption 1. All acceptance tests are perfect.

Assumption 2. The probability of failure of the primary and alternates are independent.

Assumption 3. "The cost of recovering states after a failure is small compared to the cost of executing the module." [BHAR81]

9.1.7.2 Analysis/Evaluation

This probabilistic model requires extensive rework in order to incorporate dependency and realistic assumptions.

Bhargava states that assumption 1 has been made for simplicity. Clearly, this assumption severely restricts the usefulness of this model.

The second assumption is also suspect. As Melliar-Smith hypothesized in his analysis of the reliability of recovery blocks, in highly tested flight-critical operational software, there may well be a correlation between the faults in the programs and the alternatives (as well as the acceptance tests). [MELL83] To assume in a probabilistic model that these are independent is a major concern. Bhargava makes this assumption because "it is very difficult to obtain dependencies between failures of primary and alternates. Of course this assumption can be generally true for independently designed modules".

Bhargava is currently working on generalizing his analysis by relaxing the assumption of perfect acceptance tests, and introducing probabilities that the acceptance test will or will not operate properly. Bhargava gives no further information in this paper on the status of extensions.

9.2 Summary

This study has not identified a single well developed, fully validated model to quantitatively assess the effects of fault-tolerant software.

Of the models in Table 9.8, only those of Grnarov, Scott, Soneriu, and Wei are sufficiently developed to warrant further consideration.

Only the Grnarov and Scott models have been subjected to limited validation testing. Both of these model developers have structured their models to permit extension to more than a single fault-tolerant software technique. Both model developers assumed hypothetical data values for their models and derived numerical results, plotted curves of these results, and drew conclusions. Unfortunately, their conclusions contradicted each other.

TABLE 9.8 COMPARISON OF FAULT-TOLERANT SOFTWARE RELIABILITY MODELS

FTSW TECHNIQUES MODELLED	MODELLER/TYPE							
	Grnarov Avizienis Stochastic	Mignault Probabilistic	Scott Probabilistic Input Domain	Soneriu Markov Time Domain	Wei Probabilistic Structural	Melliar Smith Probabilistic	Bhargava Probabilistic
Recovery Block (RB)	x		x		x	x	x
N Version Parallel (NVP)	x	x	x	x	x		
N Version Sequential (NVS)	x			x	x		
Tandem				x	x		
Consensus RB			x		x		
COMPARISON CRITERIA							
PROBABILITY OF FAULT							
N Version	x		x	x	x	x	x
Correlated Fault	x					x	
Fault Detection Algorithm	x		x	x	x	x	
Damage Assessment	x		x	x	x	x	
Recovery	x		x	x	x	x	
Select Prior to Design	x	x	x	x		x	
Measure Reliability	x		?	x	x		
Ease of Learning	3	7	6	5	7	4	
Ease of Use	4	7	5	5	4	3	
Degree of Validation	2	0	2	0	0	0	
Coverage (C)							

Soneriu's model for fault-tolerance can be used to represent both of the major fault-tolerant software techniques. The specific extension to both the recovery block and N-Version programming has not been done.

Wei's model requires the user to decompose the technique into segments which represent each software function and to provide failure probabilities for each segment during one request period as well as the conditional probabilities related to the frequency of execution of the segment. This was not specifically done by Wei for either of the major fault-tolerant software techniques.

While it has not been possible to select a single model that has been fully validated, the models can be useful in estimating the reliability of the fault-tolerant software techniques in contrast to the reliability of fault-intolerant software, and they all predict that fault-tolerant software can improve the reliability of the resulting systems, if properly applied.

While both Scott and Grnarov used hypothetical data, their results for both recovery blocks and N-Version programming indicate that either technique is more reliable than the fault-intolerant technique if certain conditions are satisfied:

- For the recovery block, the acceptance test must be more reliable than the primary version, and the probability of correlated faults between the versions and the acceptance test must be small;

- In the case of N-Version programming, the probability of correlated faults in the versions must be quite small. N-Version programming "success as a method for run-time tolerance of software faults depends on whether the residual design faults in each version are distinguishable". [AVIZ82] Another system level reliability consideration which should be kept in mind in N-Version programming is the impact of the reliability of the N processors executing the N-Versions on the overall system reliability.

It is not possible to state that one of the two primary fault-tolerant software techniques is more reliable than the other. Based upon the assumptions and hypothetical data used by Grnarov, his models indicate that N-Version programming is more reliable than recovery blocks. Scott's hypothetical data assumptions and models indicate that the recovery block is more reliable than N-Version programming.

Clearly, the state of the art in modelling fault-tolerant software is still in its infancy and much additional work needs to go on before any definitive conclusions can be reached. The most that one can say today is that a start

has been made and that work is required to develop and refine the analyses already conducted and to extend the results to the real world. Furthermore, much empirical data is required to validate and verify the model results. Without this empirical data and verification, the models will continue to suffer.

10.0 Conclusions and Recommendations

This chapter presents the conclusions and recommendations of this study.

10.1 Conclusions

10.1.1 Current State of the Art

Fault-tolerant software is just emerging from the research environment into the real world. Practical experience has been limited and results have been mixed; on the whole these results have been more positive than negative. For example:

- Fault-tolerant software can improve the reliability of the programs;

  The cost, performance impact, overheads, applications most susceptible to improvements and the quantification of the resulting improved reliability, as well as other important factors are not known at this time;

  The quantitative data to support definitive conclusions in these areas has not been developed;

- There has been significant conceptual development of the ideas and techniques of fault-tolerant software;

  Techniques considered include multi-version, recovery blocks, exception handlers, and variants of these;

  All techniques appear to follow the same general model or strategy - error detection, damage assessment, error recovery, fault treatment;

  Many of fault-tolerant software techniques have developed from programming practices that have been used successfully for a number of years - dissimilar software, exception handlers, assertions, etc.;

- No "best" fault-tolerant software technique exists. No validated model or other quantitative method exists to predict the "best" fault-tolerant software technique for a given application;

- There has been limited practical experience with fault-tolerant software and the results, while generally favorable, have been mixed;

Most of the practical implementations uncovered in this study were for applications that were critical to the performance of a particular function;

The NASA Shuttle is the only example of a complete operational multi-version system uncovered;

Most of the applications were imposed by regulatory bodies as part of a certification process rather than originated by the implementors. Applications were typically in failure critical areas such as commercial aircraft, nuclear reactors, and transit systems;

Because the functions were so critical, the software was generally tested as thoroughly as if there had been no fault-tolerance. As a result, the software was well tested when put into service and the incidence of error caught by the fault-tolerant software has been small;

No empirical data was collected in an organized way during most implementations. (The only exception is the work currently underway by University of Newcastle-upon-Tyne for the British Navy.) The very limited empirical data that does exist was generally collected by universities and other research institutions, and is, therefore, somewhat skewed toward an academic environment;

- Opinions have been expressed which suggest that use of fault-tolerant software will expedite the software development process by reducing the requirements for testing. (In this way the costs of the redundancy can be recouped). This study found no "hard data" to support this claim one way or the other, although it appears to be reasonable;

- Research into fault-tolerant software is entering the "second generation" at such institutions as UCLA and Newcastle-upon-Tyne;

The "first generation" research looked to the feasibility of the concepts;

The second generation seeks to expand the concepts into new areas such as telecommunications, operating systems and distributed systems. It also is oriented to producing the empirical data needed to support the analysis of the processes;

- Models are not sufficiently developed to provide accurate predictions on the improvements in reliability, costs, overheads, etc., resulting from the introduction of fault-tolerant software into an application;

Models cannot be used to quantitatively predict the effects of alternatives in the implementation of fault-tolerant software except in the broadest sense;

Well-conceived and controlled experiments must be conducted to acquire useable data to validate these (or any other) models or identify deficiencies of these models;

10.1.2 Implications on Hardware/Operating Systems/Languages

- Modern computer design principles and practices are generally consistent with the requirements of fault-tolerant software;

  Encapsulation of data, programs and other computational entities facilitates the introduction of fault-tolerant software;

  Other concepts such as hierarchical decomposition of hardware and software, kernel and ring based designs, context management and name based memory management (associative processing) also make the use of fault-tolerant software easier;

- Trends of reduced costs for computer hardware will generate increased use of hardware redundancy and thus also accelerate the introduction of fault-tolerant software;

  Specialized hardware such as processors, multiple register sets, controllers, memory caches and other logical units, needed to efficiently support various fault-tolerant software functions will become more economical;

- The concept of Idealized Fault-Tolerant Components (IFTC) offers promise in that it may give the designer control over the amount of error detection and the kind of error recovery techniques most appropriate to a specific application;

- Ada, as currently defined, provides significant support for highly fault-tolerant software. However, Ada also has certain major shortcomings in this regard, and changes and enhancements are required before it can be an effective language support tool. Ada provides significant encapsulation capabilities including control encapsulations over procedures, functions, and tasks, as well as strong data typing (packages);

  Ada rendezvous specifications provide no way to terminate a rendezvous once it has begun;

Ada timing constraint specifications are not generalized and need to be extended;

Ada does not support a "single level terminating model for exception handling";

- C does not provide many of the constructs for fault-tolerant supports. Basic facilities such as concurrent processing and exception handling must be implemented external to C.

10.2 Recommendations

This section presents the recommendations that have been developed in the course of this study. Recommendations are given in two areas: (1) research issues requiring immediate attention if fault-tolerant software is to be developed conceptually to its full potential; and (2) developmental issues which are deemed important because they will provide the base required to practically implement systems using these techniques.

10.2.1 Research Issues

It is apparent that much additional work is needed before fault-tolerant software can be considered a truly mature technology. The purpose of this section is to outline research issues that were identified in the course of the study as requiring additional research before the state of the art of fault-tolerant software can be significantly advanced. These issues are shown below:

- The first issue that must be addressed is the question of the relative independence of alternate versions of software and the correlation of errors. Fault-tolerant software is based upon the assumption that software errors will be random and that redundancy in the form of multiple versions or alternates will improve the overall reliability of the software because the simultaneous occurrence of faults occurring in all or most of the versions will be small. If it is shown that this assumption is not valid, or valid only in selected application areas, then this will have a major effect on the introduction and utility of fault-tolerant software in these environments;

- The second issue is the ability of certain specific modern structures to support fault-tolerant software. This study briefly reviewed the capabilities of Ada and the Advanced Information Processing System (AIPS) to accommodate fault-tolerant software. Additional study and analysis is required to determined more

specifically the changes, enhancements, costs and impacts of modifying Ada and AIPS to accommodate fault-tolerant software in a more efficient manner;

- The third issue requiring immediate attention is the application of fault-tolerant software to problems involving a high degree of concurrency between processes and computers. Research into fault-tolerant software in a distributed highly concurrent environment must address such problems as eliminating service requests from software that has been cancelled, release of resources required by concurrent processes, etc.;

- The fourth major research issue is the need for a validated model or other analytical technique to predict the impact of introducing fault-tolerant software into software design. As stated earlier in this paper, the models which have been developed have problems and require further validation using empirical data. It may be that the only modeling approach which is possible is so highly applications dependent that the best generalization that can be achieved a set of guidelines or standards which must be included. In any case, good and accurate quantitative assessment tools are needed to advance the technology;

- The fifth research issue is the need for a methodology to be used to manage processes in the case of reconfiguration. In the case of a major software failure (e.g., the system enters a region of the input space which consistently causes a version to fail), reconfiguration will have to take place under the guidance of the application program itself. Also, in the case where a significant portion of the computational resources are lost, it may be necessary to reconfigure the system to a minimum essential set of functions.

  How such a set is specified and guaranteed to be operational is unresolved. It is imagined that this last problem can be solved by defining several domains and using a domain manager to control the execution of domains depending upon resources. In order to be able to perform its task, this domain manager will require information concerning the availability of hardware resources and some history of the failures of the software. It will also need the ability to abort processes and restart them. (This implies some rebinding of the names of these processes will be necessary);

- The sixth major research issue is the applicability of fault-tolerance in the design and implementation of operating systems. While the introduction of fault-tolerant software technology at the application level has been demonstrated, the use of these techniques in operating systems is not clear;

- The last major research issue is the need to develop a comprehensive engineering methodology for applying fault-tolerance to systems architecture. [Design principles and procedures need to be evolved in the areas of specifications development, systems functional design, systems detailed design and implementation techniques.] The IFTC provides a beginning for this development. This concept should be expanded and unfolded to provide the tools necessary for real world application of fault-tolerance.

10.2.2 Development Issues

There is a great lack of empirical data with which to make decisions concerning the impacts and costs of fault-tolerant software and upon which to base predictions on the effectiveness of the various proposed techniques.

- The first development issue is the immediate need for an engineering test bed for which to capture the empirical data on the use, efficacy, practicality and costs of fault-tolerant software;

- The second development issue is the introduction of fault-tolerant software into the Advanced Information Processing System (AIPS);

  To provide empirical data, a test facility must be developed; the facility should provide the tools (language notations and mechanisms) needed to conveniently and efficiently implement and test various fault-tolerant software techniques; the facility should also provide the data capture tools and strategies needed to introduce controlled redundancy into software and to measure its impact; finally, it should provide the software development mechanisms needed to test the use of fault-tolerant software in realistic environments.

AIPS, as a new highly reliable system, is a good candidate for an engineering and development tested facility. AIPS, however, is several years in the future, and the empirical data collection needed can not wait. Therefore, some interim test facility should be constructed and an active experimentation program developed to use the facility.

BIBLIOGRAPHY

ANDE76
Anderson, T. and R. Kerr. Recovery Blocks in Action: A System Supporting High Reliability. Proceedings of 2nd International Conference on Software Engineering. October 1976. pp 447-457.

ANDE81
Anderson, T. and P. A. Lee. Fault-Tolerance Principles and Practice. Prentice-Hall International. 1981.

ANDE81a
Anderson, T. and John Knight. Practical Software Fault-Tolerance in Real-Time Systems. ICASE Report 81-10. NASA Langley Research Center. May 1981. (a)

ANDE83
Anderson, T. and John Knight. A Framework for Software Fault Tolerance in Real-Time Systems. IEEE Transactions on Software Engineering, Vol SE-9(3). May 1983. pp 355-364.

ANDE84
Anderson, T. Personnel communication provided by T. Anderson at NASA Langley. University of Newcastle upon Tyne. 1984.

ANDR79
Andrews, D. M. Using Executable Assertions for Testing and Fault-Tolerance. Digest of Papers of FTCS 9: The 9th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1979. pp 102-105.

AVIZ77
Avizienis, A. and L. Chen. On the Implementation of N-version Programming for Software Fault-Tolerance During Execution. Proceedings of COMPSAC 77. November 1977. pp 149-155.

AVIZ82
Avizienis, A. and John Kelly. Fault-Tolerant Multi-version Software: Experimental Results of a Design Diversity Approach. UCLA Computer Science Dept, Quarterly. Spring, 1982.

AVIZ83
Avizienis, A. Personal communication between Battelle Columbus Laboratories and A. Avizienis. UCLA. Los Angeles, California. November 1983.

AYAC79
Ayache, J. M., P. Azema, and M. Diaz. Observer: A Concept for On-Line Detection of Control Errors in Concurrent Systems. Digest of Papers of FTCS 9: The 9th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1979. pp 79-86.

BACK78
Backus, J. Can Programming be Liberated from the von Neumann Style?. Communications of the ACM. August 1978. pp 613-641.

BEUS69
Beuscher, H. J., G. E. Gessler, D. W. Huffman, P. J. Kennedy, and E. Nussbaum. Administration and Maintenance Plan. Bell System Technical Journal, Vol 48. October 1969.

BHAR81
Bhargava, B. Software Reliability in Real-Time Systems. Proceedings of National Computer Conference 1981.

BLAC80
Black, J. P., D. J. Taylor, and D. E. Morgan. An Introduction to Robust Data Structures. Digest of Papers of FTCS 10: The 10th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1980.

BLAC81
Black, J. P., D. J. Taylor, and D. E. Morgan. A Compendium of Robust Data Structures. Digest of Papers of FTCS 11: The 11th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1981.

BOI81
Boi, P. M., et. al. Exception Handling and Error Recovery Techniques in Modular Systems--and Application to the ISAURE System. Digest of Papers of FTCS 11: The 11th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1981.

BOUT80
Boute, R. Simplifying Ada by Removing Limitations. SIGPLAN, Vol 15(2). February 1980. pp 17-28.

CAMP79
Campbell, R. H., K. H. Horton, and G. G. Belford. Simultions of a Fault-Tolerant Deadline Mechanism. Digest of Papers of FTCS 9: The 9th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1979.

CART83
Carter, W. C. Architectural Considerations for Detecting Run Time Errors in Programs. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1983. pp 249-256.

CHEN78a
Chen, L. and A. Avizienis. N-Version Programming: A Fault-Tolerant Approach to the Reliability of Software Operation. Digest of Papers of FTCS 8: The 8th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1978. pp 3-9. (a)

CHEN78b
Chen, L. Improving Software Reliability by N-version Programming. Technical Report, UCLA Computer Science Dept. September 1978. (b)

CLAR80
Clarke, L., Jack Wileden, and A. Wolf. Nesting in Ada Programs is for the Birds. SIGPLAN, Vol 15(11). November 1980. pp 139-145.

CONN72
Connet, J.R., E.J. Pasterrak and B.D. Wagner. Software Defenses in Real-Time Control Systems. Digest of Papers of FTCS 2: The 2nd Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1972.

CRIS82a
Cristian, F. Exception Handling and Software Fault-Tolerance. IEEE Transactions on Computers, Vol C-31(6). June 1982. pp 531-540.

CRIS82b
Cristian, F. Robust Data Types. Acta Informatics, Vol 17. 1982. pp 365-397.

CRIS83
Cristian, F. Correct and Robust Programs. Technical Report, University of Newcastle upon Tyne, Computing Laboratory. February 1983.

CSDL83
Charles Stark Draper Laboratory, AIPS System Requirements. CSDL Report # AIPS-83-50, August 30, 1983.

DALE82
Dale, C. J. and L. N. Harris. Software Reliability Evaluation Methods. British Aerospace Report No. ST 26750. September 1982.

DE81
DE, B.B. and H.B. Krakan, Fault-tolerance in a Multi-Processor, Digital Switching System. IEEE Transaction on Reliability, Vol R-30, No. 3, August 1981.

DIJK76
Dijkstra, Edger. A Discipline of Programming. Prentice-Hall. 1976.

DIJK78
Dijkstra, Edger. On the GREEN Language Submitted to the DOD. SIGPLAN Vol 13(10). October 1978. pp 16-21.

GEPH78
Gephart, L. S., C. M. Greenwald, M. M. Hoffman, and D. H. Osterfeld. Software Reliability: Determination and Prediction. Technical Report AFFDL-TR-78-77. June 1978.

GILB84
Gilbert, Ray and Norm Ichiyen. Personal communication between Battelle Columbus Laboratories and Gilbert and Ichiyen. Atomic Energy of Canada Limited. January 1984.

GILO83
Giloi, W. K. and P. Behr. Hierarchical Function Distribution - A Design Principle for Advanced Multicomputer Architecture. Proceedings of the 10th Annual International Symposium on Computer Architecture. IEEE Computer Society and ACM. 1983. pp 318-325.

GOEL82
Goel, A. L. Software Reliability Modeling and Estimation Techniques. RADC-TR-82-263. October 1982.

GOLD80
Goldberg, Jack. SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control. IFIPS World Computer Congress. IFIPS. 1980.

GOLD84
Goldberg, Jack, et al. Development and Analysis of the Software Implemented Fault-Tolerance (SIFT). Technical Report, NASA Contractor Report 172146, Contract NAS1-15428. SRI International, Menlo Park, California. February 1984.

GREV83
Greve, W. E., and R. J. Schroder. A Distinct Software Implementation in a Vehicle Controller. Boeing Aerospace Company. IEEE Proceedings of 33d Vehicular Technology Conference. Toronto, Canada. May 25-27, 1983.

GRNA80
Grnarov, A., J. Arlatt, and A. Avizienis. Modeling of Software Fault-Tolerance Strategies. Proceedings of 1980 Pittsburgh Modeling and Simulation Conference. Pittsburgh, Pennsylvania. May 1980.

GRNA81
Grnarov, A., J. Arlatt, and A. Avizienis. Modeling and Performance Evaluation of Software Fault-Tolerance Strategies. Technical Report, UCLA Computer Science Dept. 1981.

HECH76
Hecht, H. Fault-Tolerant Software Software for Real-Time Systems. Computing Surveys, Vol 8(4). ACM. December 1976. pp 391-407.

HECH79
Hecht, H. Fault-Tolerant Software. IEEE Transactions on Reliability, Vol R-28(3). August 1979. pp 227-232.

HECH82
Hecht, H. and M. Hecht. Use of Fault Trees for the Design of Recovery Blocks. Digest of Papers of FTCS 12: The 12th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1982. pp 134-139.

HOAR81
Hoare, C. A. R. The Emperor's Old Clothes. CACM, Vol 24(2). February 1981. pp 75-83.

HORT78
Horton, K. H., R. H. Campbell, and G. G. Belford. Meeting Real-Time Deadlines. Proceedings of Computers, Electronics, and Control. Acta Press. 1978.

HOSS83
Hosseini, S., J. Kuhl and S. Readdy. An Integrated Approach to Error Recovery in Distributed Computing Systems. IEEE. 1983.

JAME82
James, L. E., J. E. Angus, J. B. Bowen, and J. McDaniel. Combined Harware-Software Reliability Models. Technical Report RADC-TR-82-88. April 1982.

JESS82
Jessop, W. Ada Packages and Distributed Systems. SIGPLAN, Vol 17(2). February 1982. pp 28-36.

JONE80
Jones, Douglas. Tasking and Parameters: A Problem Area in Ada. SIGPLAN, Vol 15(5). May 1980. pp 37-40.

KELL82
Kelly, John. Specification of Fault-Tolerant Multi-version Software: Experimental Studies of a Design Diversity Approach. PhD Dissertation, UCLA. 1982.

KELL83
Kelly, John and A. Avizienis. A Specification-Oriented Multi-version Software Experiment. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1983.

KNIG83
Knight, John and John Urquhart. Fault Tolerant Distributed Systems Using Ada. AIAA Computers in Aerospace IV Conference. October 1983. pp 37-44.

KNIG84
Knight, John. Personal communication between Battelle Columbus Laboratories and John Knight. University of Virginia. Charlottesville, Virginia. January 1984.

LEE78
Lee, P. A. A Reconsideration of the Recovery Block Scheme. Computer Journal, Vol 21(4). November 1978. pp 306-310.

LEE79
Lee, P. A., N. Ghant, and K. Heron. A Recovery Cache for the PDP-11. Digest of Papers of FTCS 9: The 9th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. June 1979.

LEE83
Lee, P. A. Structuring Software Systems for Fault Tolerance. AIAA Computers in Aerospace IV Conference. October 1983. pp 30-36.

LEVE83
Leveson, N. G. and T. J. Shimeall. Safety Assertions in Process Control Systems. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1983. pp 236-240.

LIND82
Linden, Peter van der. Ambiguity and Orthogonality in Ada. SIGPLAN, Vol 17(3). March 1982. pp 93-94.

LISK79
Liskov, Barbara and Alan Snyder. Exception Handling in CLU. IEEE Transactions on Software Engineering, Vol SE-5(6). November 1979. pp 546-558.

MAHJ81
Mahjoub, Ahmed. Some Comments on Ada as a Real-Time Programming Language. SIGPLAN, Vol 16(2). February 1981. pp 89-95.

MAKA82
Makam, S. V. Design of a Fault-Tolerant Computer System to Execute N-version Software. Technical Report, UCLA Computer Science Dept. December 1982.

MART82
Martin, D. J. Dissimilar Software in High Integrity Applications in Flight Controls. NATO AGARD Conference Proceedings No. 330 "Software for Avionics". October 6, 1982.

MART84a
Martin, David. Personal communication between Battelle Columbus Laboratories and David Martin. Marconi Avionics. Rochester, Kent, England. January 1984.

MART84b
Martin, Don. Personal communication between Battelle Columbus Laboratories and Don Martin. Boeing. Seattle, Washington. January 1984.

MATS83
Matsumoto, Y. and H. Nakamura. TREX/MCS: A Fault-Tolerant Multicomputer. Proceedings IFAC Safecomp 83. IFAC. 1983. pp 255-260.

MCAL83
McAllister, David. Personal communication between Battelle Columbus Laboratories and David McAllister. North Carolina State University, North Carolina. December 1983.

MCWH84
McWha, Jim and Peter O'Toole. Personal communication between Battelle Columbus Laboratories and Jim McWha and Peter O'Toole. Boeing. Seattle, Washington. January 1984.

MELL82
Melliar-Smith, P.M. and R. Schwartz. Formal Specifications and Mechanical Proof of SIFT: A Fault-Tolerant Flight Control System. IEEE Transactions on Computers, Vol C31(7). July 1982.

MELL83
Melliar-Smith, P. M. Development of Software Fault-Tolerance Techniques. NASA CR-172122. March 1983.

MIGN82
Migneault, G. E. The Cost of Software Fault-Tolerance. NASA Technical Memorandum 84546, NASA-Langley Research Center. September 1982.

MIGN83
Migneault, G.E. On Requirements for Software Fault-Tolerance for Flight Controls. Technical Report, NASA Langley Research Center. 1983.

MORR81
Morris, M. A. An Approach to the Design of Fault-Tolerant Software. M.S. Thesis. Cranfield Institute of Technology. September 1981.

NAMJ83
Namjoo, M. CEREBRUS-16: An Architecture for a General Purpose Watchdog Processor. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1983. pp 17-20.

NEUM80
Neumann, P. G., et al. A Probably Secure Operating System: The System, its Applications and Proofs. Technical Report, Computer Science Laboratory No. CSL-116. SRI International. Menlo Park, California. May 1980.

ORNS75
Ornstein, S.M., et al. Pluribus - A Reliable Multiprocessor. AFIPS National Computer Conference. AFIPS. 1975. pp 551-559.

PARN72
Parnas, D. On the Response of Detected Errors in Heirarchically Structured Systems. Technical Report, Carnegie Mellon University. 1972.

PATT82
Patterson, D. A. and C. H. Sequin. RISC 1: A Reduced Instruction Set VLSI Computer. Proceedings of the 8th Annual Symposium on Computer Architecture. ACM. 1982.

PEAS80
Pease, M. C., R. Shostak and L. Lamport. Reaching Agreement in the Presence of Faults. Journal of the ACM. Vol 27(2). April 1980. pp 228-234.

PRAT83
Pratt, T. W., J. Knight, and S. T. Gregory. On the Engineering of Crucial Software. NASA Grant No. AG-1-233, Report No. UVA/52808/AMCS83102. February 1983.

RAMA82
Ramamoorthy, C. V. and F. B. Bastani. Software Reliability - Status and Perspectives. IEEE Transactions on Software Engineering, Vol SE-8(4). July 1982.

RAND75
Randell, B. System Structure for Fault-Tolerance. IEEE Transactions on Software Engineering Vol SE-1(2). June 1975. pp 220-232.

RAND78a
Randell, B. Reliable Computing Systems. SpringerVerlay. W. Germany. 1978. (a)

RAND78b
Randell, B. Error Recovery in Distributed Computer Systems. Computer Bul. Ser. 2, No. 16. June 1978. (b)

RAND78c
Randell, B., P. A. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. Computer Surveys, Vol 10(2). June 1978. pp 123-165. (c)

RAND79
Randell, B. Software Fault-Tolerance. EURO IFIP. 1979.

RAND83a
Randell, B. Fault Tolerance and System Structure. 4th Jerusalem Conference on Information Technology (TB Given). Computing Laboratory, University of Newcastle upon Tyne. December 2, 1983. (a)

RAND83b
Randell, B. Recursively Structured Distributed Computing Systems. Technical Report. University of Newcastle upon Tyne. May 1983. (b)

RANE83
Raney, L. H. The Use of Fault-Tolerant Software for Flight Control Systems. NAECON. 1983.

RUSH83
Rushby, J. and B. Randell. A Distributed Secure System. IEEE Transactions on Computer. July 1983. pp 55-67.

RUSS80
Russell, D. L. State Restoration in Systems of Communicating Processes. IEEE Transactions on Software Engineering, Vol SE-6(2). March 1980. pp 183-194.

SANT83
Santo, M. Di, L. Nigro, and W. Russo. Programming Reliable and Robust Software in Ada. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1983. pp 196-203.

SCHI78
Schick, G. J. and R. W. Wolverton. An Analysis of Computing Software Reliability Models. IEEE Transactions on Software Engineering, Vol SE-4(2). March 1978.

SCOT83a
Scott, R. K. Data Domain Modeling of Fault-Tolerant Software Reliability. PhD Dissertation. North Carolina State University, Raleigh, North Carolina. 1983. (a)

SCOT83b
Scott, R. K., J. W. Gault, and D. F. McAllister. Modelling Fault-Tolerant Software Reliability. Proceedings of the 3d Symp. on Reliability in Distributed Software and Database Systems. 1983. (b)

SCOT84
Scott, R. K., J. W. Gault, D. F. McAllister, and J. Wiggs. Experimental Validation of Six Fault-Tolerant Reliability Models. Submitted to 14th Annual International Symposium on Fault-Tolerant Computing. 1984.

SHAW80
Shaw, Mary. The Impact of Abstraction Concerns on Modern Programming Languages. Proceedings of the IEEE, Vol 68(9). September 1980. pp 1119-1130.

SHRI78
Shrivastava, S.K. and J.P. Banatre. Reliable Resource Allocation Between Unreliable Processes. IEEE Transaction on Software Engineering. Vol SE-4, May 1978, pp. 230-241.

SHRI82
Shrivastava, S. K., and F. Panzieri. The Design of a Reliable Remote Procedure Call Mechanism. IEEE Transactions on Computers, Vol C-31(7). July 1982.

SILB81
Silberschatz, Abraham. On the Synchronization Mechanism of the Ada Language. SIGPLAN, Vol 16(2). February 1981. pp 96-103.

SMIT81
Smith, T. B. Generic Data Manipulative Primitives of Synchronous Fault-Tolerant Systems. Digest of Papers of FTCS 11: The 11th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1981.

SMYT84
Smyth, Richard. Personal communication between Battelle Columbus Laboratories and Richard Smith. Milco International. Huntington Beach, California. January 1984.

SONE81
Soneriu, M. D. A Methodology for the Design and Analysis of Fault-Tolerant Operating Systems. PhD Dissertation. Illinois Institute of Technology. Chicago, Illinois. 1981.

SUKE76
Sukert, A. N. A Software Reliability Modeling Study. In-house Technical Report. RADC-TR-76-247, AD A030-437. 1976.

TAYL80
Taylor, D. J., D. E. Morgan, and J. P. Black. Redundancy in Data Structures: Some Theoretical Results. IEEE Transactions on Software Engineering, Vol SE 6(6). November 1980. pp 595-602.

TAYL81
Taylore, R. Redundant Programming in Europe. ACm SIGSOFT SEN, Vol 6. No. 1, January 1981.

TROY84
Troy, Alan. Personal Communications Between Battelle Columbus Laboratories and alan Troy. Rockwell - Space Division. Downey, California. February 1984.

TYNE81
Tyner, P. iAPX 432 General Data Processor Architecture Manual. Technical Report, Intel Corporation. 1981.

VONL79
von Linde, Otto Berg. Computers Can Now Perform Vital Functions Safely. Railway Gazette International. November 1979.

WEGN83
Wegner, P. On the Unification of Data and Program Abstraction in Ada. Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages. ACM. January 1983.

WEI81
Wei, Anthony. Real-Time Programming with Fault-Tolerance. PhD Dissertation (N82-20897). University of Illinois. Urbana, Illinois. 1981.

WELC83
Welch, H. O. Distributed Recovery Block Performance in a Real-Time Control Loop. Proceedings of Real-Time Systems Symposium. December 6-8, 1983.

WILL82
Willett, R. J. Design of Recovery Strategies for a Fault-Tolerant No. 4 Electronic Switching System. Bell System Technical Journal, Vol 61(10). December 1982.

WILL83
Williams, J. F., L. J. Yount, and J. B. Flannigan. Advanced Autopilot-Flight Director System Computer Architecture for Boeing 737-300 Aircraft. 5th Digital Avionics Systems Conference. October 31 - November 3, 1983.

WILL84
Williams, John. Personal Communication between Battelle Columbus Laboratories and John Williams. Sperry Corporation. Phoenix, Arizona. January 1984.

WOOD80
Wood, W. G. Recovery Control of Communicating Processes in a Distributed System. Technical Report 158, Computing Laboratories, University of Newcastle upon Tyne. November 1980.

YAU80
Yau, S. S. and F. Chen. An Approach to Concurrent Control Flow Checking. IEEE Transactions on Software Engineering, Vol SE-6(2). March 1980. pp 126-137.

YOSH83
Yoshihara, K., Y. Koga, and T. Ishihara. A Robust Data Structure Scheme with Checking Loops. Digest of Papers of FTCS 13: The 13th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society. 1983. pp 241-248.

ZUBE84
Zuber, Pierre. Personal Communication Between Battelle Columbus Laboratories and pierre Zuber. Westing House Transportation Division, Pittsburgh, Pennsylvania.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-172385		

4. Title and Subtitle	5. Report Date
Study of Fault-Tolerant Software Technology	September 1984
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
T. Slivinski, C. Broglio, C. Wild, J. Goldberg, K. Levitt, E. Hitt, J. Webb	
	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
MANDEX, INC. 5201 Leesburg Pike Suite #207 Falls Church, Virginia 22041	NAS1-17412
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes
Langley technical monitor: Dr. David Eckhardt
Final Report

16. Abstract

This paper presents an overview of the current state of the art of fault-tolerant software and an analysis of quantitative techniques and models that have been developed to assess its impact. It examines research efforts as well as experience gained from commercial application of these techniques. The paper also addresses the computer architecture and design implications on hardware, operating systems and programming languages (including Ada) of using fault-tolerant software in real-time aerospace applications.

The paper concludes that fault-tolerant software has progressed beyond the pure research state. The paper also finds that, although not perfectly matched, newer architectural and language capabilities provide many of the notations and functions needed to effectively and efficiently implement software fault-tolerance.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault-tolerance software; Computer software, Recovery blocks, and Multiversion programming.	[redacted]

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
UNCLASSIFIED	UNCLASSIFIED	155	