NASA/TM-2000-210616

Software Fault Tolerance: A Tutorial

Wilfredo Torres-Pomales
Langley Research Center, Hampton, Virginia

October 2000

NASA/TM-2000-210616

Software Fault Tolerance: A Tutorial

Wilfredo Torres-Pomales
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2000

Abstract

Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems. The root cause of software design errors is the complexity of the systems. Compounding the problems in building correct software is the difficulty in assessing the correctness of software for highly complex systems. This paper presents a review of software fault tolerance. After a brief overview of the software development processes, we note how hard-to-detect design faults are likely to be introduced during development and how software faults tend to be state-dependent and activated by particular input sequences. Although component reliability is an important quality measure for system level analysis, software reliability is hard to characterize and the use of post-verification reliability estimates remains a controversial issue. For some applications software safety is more important than reliability, and fault tolerance techniques used in those applications are aimed at preventing catastrophes. Single version software fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling, and others. Multiversion techniques are based on the assumption that software built differently should fail differently and thus, if one of the redundant versions fails, at least one of the others should provide an acceptable output. Recovery blocks, N-version programming, N self-checking programming, consensus recovery blocks, and t/(n-1) techniques are reviewed. Current research in software engineering focuses on establishing patterns in the software structure and trying to understand the practice of software engineering. It is expected that software fault tolerance research will benefit from this research by enabling greater predictability of the dependability of software.

Acknowledgements

I would like to acknowledge my gratitude to Dr. F. Gail Gray of the Virginia Polytechnic Institute and State University (VA Tech) for providing advice during the preparation and reviewing of this report. I also wish to acknowledge my appreciation to Mr. Ricky Butler of the NASA Langley Research Center for his help in reviewing the final version of this report.

Table of Contents

List of Figures

List of Tables

1. Introduction

Software permeates every aspect of modern society. Government, transportation, manufacturing, utilities, and almost every other sector that influences our way of life is affected directly or indirectly by software systems. The flexibility provided by software-controlled systems, the insatiable appetite of society for new and better products, and competition for business drive the continued expansion of the domain ruled by software systems. Without software, many of our modern conveniences would be virtually impossible.

Despite its widespread use, software is hardly ever "perfect". For a myriad of reasons, it is extremely difficult to produce a flawless piece of software. According to [Lyu 95], "software is a systematic representation and processing of human knowledge". For humans, perfect knowledge of a problem and its solution is rarely achieved. [Abbott 90] states that "programs are really not much more than the programmer's best guess about what a system should do". Even if a programmer had sufficient knowledge to solve a problem, that knowledge must be transformed into a systematic representation adequate for automatic processing. Our computers today are merciless when it comes to processing software: if there is an error in the logic, sooner or later that error will show up in the output independently of the consequences. Only the most trivial software problems can be solved without some trial and error. As computers are applied to solve more complex problems, the probability of logic errors being present in the software grows.

F. P. Brooks [Brooks 87] conjectured that the hard part about building software is not so much the representing of the solution to a problem in a particular computer language, but rather what he called the "essence" of a software entity. This essence is the algorithms, data structures, functions, and their interrelationships. Specification, design, and testing of this conceptual construct is the "hard part" of software engineering. This is not to say that capturing the software description in a textual or graphical manner is not difficult in itself; it certainly is. To Brooks, the labor intensiveness associated with software is really more of an "accidental" difficulty. Brooks enumerates four inherent properties that make software hard: complexity, conformity, changeability, and invisibility. Software is complex because of the extremely large number of states present in a design and the nonlinear interactions among these states. Software is forced to conform because it is perceived as the most conformable of all the components in a system. Software design complexity often follows from requirements to accommodate interfaces designed with no apparent consideration for homogeneity and ease of use. Also, it is often left to the software to handle deficiencies and incompatibilities among other system components. Software changes continuously because it is extremely malleable. As such, new and revised system functionality is often implemented through software changes. Even if the software is not the direct target of a change in system functionality, the software is forced to change to accommodate changes in other system components. Lastly, software is invisible. We use computer languages to try to capture the essence of software, but the concepts are so intricate that they generally defy attempts to completely visualize them in a practical manner and require the use of techniques to simplify relationships and enable communication among designers.

Software engineering is the discipline concerned with the establishment and application of sound engineering practices to the development of reliable and efficient software. The IEEE defines software engineering as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [IEEE 93]. This discipline has been around for more than forty years, and in that time software engineering practices have made possible significant accomplishments. Textbooks exist on the subject (e.g., [Pressman 97]) and guidelines for the development of software abound (e.g., [Mazza 96]). We will review some high-level concepts of the design and verification of software from the perspective of realizing what is involved in a complete and disciplined development effort.

Because absolute certainty of design correctness is rarely achieved, software fault tolerance techniques are sometimes employed to meet design dependability requirements. Software fault tolerance refers to the use of techniques to increase the likelihood that the final design embodiment will produce correct and/or safe outputs. Since correctness and safety are really system level concepts, the need and degree to use software fault tolerance is directly dependent on the intended application and the overall system design.

This paper reviews the concepts of software fault tolerance. Our aim is to survey the literature and present the material as an introduction to the field. We emphasize breadth and variety of the concepts to serve as a starting point for those interested in research and as a tutorial for people wanting some exposure to the field. The next section is an overlook of the software development process.

2. Software Development

The goal here is to present some of the ideas in software engineering. The information in this section is based on [Pressman 97] and [DO178B]. The reader should consult those and other references for a more detailed and precise treatment of the subject.

The software life cycle is composed of three types of processes: planning, development, and supporting processes. The planning process is the first step in the cycle and its goal is to define the activities of the development and supporting processes, including their interactions and sequencing. The data produced by the processes and the design development environments, with definitions of the methods and tools to be used, are also determined during the planning process. The planning process scopes the complexity of the software and estimates the resources needed in the development activities. Software development standards and methods are selected during the planning activity. Standards exist for all the processes in the software life cycle and are used as part of the quality assurance strategy. For better performance in terms of development time and quality of the final product, the plans should be generated at a point in time that provides timely direction to those involved in the life cycle process. The planning process should provide mechanisms for further refinement of the plans as the project advances.

The strategy used to develop the software, also known as the process model or software engineering paradigm, is chosen based on the particular characteristics of the application to be developed. Further consideration in selecting a process model is given to project elements like the maturity level of the developers and the organization, tools to be used, existent process control, and the expected deliverables. Various process models have been proposed in the software engineering literature [Pressman 97]. The Linear Sequential Model is the most basic and straightforward process model (see Figure 1). Following the system engineering analysis where requirements are allocated to each element of a full system, the software development process goes through the steps of analyzing its allocated requirements and categorizing them in terms of functional, performance, interface and safety-related requirements. After this phase is complete, the high level design is built and the code is generated. Testing is then performed on the final coded version.

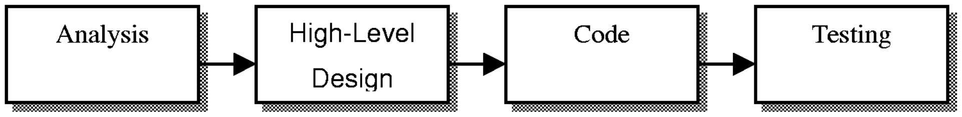

Figure 1: Linear Sequential Process Model

Figure 2 presents the Prototyping Process Model. This process model is appropriate for projects where the requirements are incompletely specified or when the developers are unsure whether a proposed design solution is adequate. The process begins with a requirements capture activity, followed by a quick design and build of a prototype or mock-up of the product. After analyzing the prototype, further refinements to the requirements are generated and the process begins again. This cycle activity not only helps develop the requirements, but it also helps the developers better understand the problem.

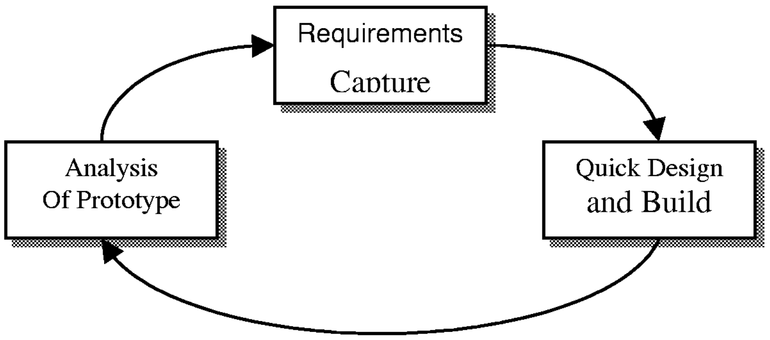

Figure 2: Prototyping Process Model

Other process models have been proposed. The Rapid Application Development (RAD) process model uses multiple teams of developers working simultaneously on different modules of an application with all the teams following what is basically the Linear Sequential Model to develop their corresponding module. The Incremental Model is an evolutionary process model that combines the Linear Sequential Model with prototyping activity in an iterative fashion; after each iteration the result is an incremental improvement on the software product. The Spiral Model, the Component Assembly Model, and the Concurrent Development Model are other evolutionary process models.

Regardless of the process model chosen, actual development of the software has four main processes: requirements capture, design, coding, and integration. The high-level software requirements are developed during the requirements capture process. These requirements include functional, performance, interface and safety requirements derived from the system level analysis and development. The capturing of requirements is usually an iterative process with corrections being added to rectify omissions, ambiguities, and inconsistencies found during the other processes. Further corrections and additions to the requirements can also originate from changing system-level requirements and design.

The design process produces the software architecture and corresponding lower level requirements for the coding process. The architectural design is a modular hierarchical decomposition of the software, including the control relationships and data flow between the modules. Complementary to the architectural design, this process also includes activities for performing the data design, interface design and procedural design. The data design is the selection of the data structures to be used and the identification of the program modules that operate directly on these structures. The interface design considers the interactions between software modules, between the software and other non-human external entities, and between the computer and human operators. A general guideline applicable to all software designs is the implementation of data validation and error handling capability within each structural module in order to control the propagation of side effects associated with the processing of erroneous data. Procedural design is the selection of the algorithms to be used by the software components. The design process should allocate requirements to software elements, identify available system resources with their limitations and selected managing strategies, identify scheduling procedures and inter-processor and/or inter-task communication mechanisms, and select design methods and their implementation. This part of the software development deals with what F. P. Brooks [Brooks 87] called the "essence" of a software entity. In general, errors originating during this phase of development tend to be systemic errors stemming from the inherent difficulty in mastering the complexity of the problem being solved and the proposed design solution.

The coding process develops the source code implementing the software architecture, including the data, interface, and procedural designs together with any other low-level requirements. If the output of the design process is detailed enough, the source code can be generated using automatic code generators. Any errors or inadequacies found during the coding activities generate problem reports to be resolved by changes to the requirements or the design. According to F. P. Brooks [Brooks 87], the difficulty in developing the source code is really "accidental", meaning that the errors originating here tend to be random oversight errors or systemic errors caused by the difficulty of mapping the design onto a particular computer language representation.

The integration process is the phase of development when the source code is linked and transformed into the executable object code to be loaded on the target computer hardware. If the design and coding were done properly, this step should flow smoothly. However, errors in integration do appear often, and any interfacing problem found during this process should generate a problem report to be resolved by the previous development processes.

The supporting processes are three: verification, configuration management, and quality assurance. The purpose of the verification process is to search and report errors in the software requirements, design, source code, and integration. Verification is composed of three types of activities: reviews, analysis and testing. Reviews are qualitative inspections of the software items targeting compliance with requirements, accuracy and consistency of the structural modules and their outputs, compatibility with the target computer, verifiability, and conformance to standards. A review can take several forms: informal meetings to discuss technical problems; formal presentations of the design to customers, management, and the technical staff; and formal technical reviews or walkthroughs. Verification through analysis uses quantitative techniques to examine the software for functionality, performance, and safety in a system context. Analyses check the information domain to see how data and events are processed. Functional analysis is used to check the input-output functionality of the system and its components. Behavioral analysis studies the stimulus-response characteristics, including internal and external events and state sequencing. Testing is used as a complementary verification activity to provide assurance of correctness and completeness. There are four types of tests performed at each of three levels of the software system hierarchy. Tests should be performed at the low-level structural decomposition modules, at the software integration level where the modules are "wired" to perform some architectural functionality, and at the hardware-software integration level to verify proper operation of the software on the target computer. At each level, normal input range tests, input robustness tests, requirements-based tests, and structural-coverage tests should be performed. Normal input range tests demonstrate the ability of the software to respond properly to normal inputs. These tests should be developed considering internal state transitions, timing, and sequencing. Robustness tests check the response of the software to abnormal inputs in value, timing, or sequencing. Requirements-based testing develops tests for each software requirement, be it derived from system-level requirements, a high-level requirement, or a low-level derived requirement. Structural coverage testing is based on structural coverage analysis that identifies code structures not exercised by the previous tests. Testing techniques like path testing, data flow testing, condition testing and loop testing are used to develop tests that help in increasing the structural test coverage.

The Configuration Management Process identifies, organizes, and controls modifications to the software being built with the objective of minimizing the level of confusion generated by those changes. The changes include not only changes in the requirements coming from the system level analysis, but also the natural changes that occur in the software design as the project advances and the design items are created. This supporting process includes four main activities: configuration identification, baseline establishment, change control, and archiving of the software product. Configuration identification is the unambiguous labeling of each configuration item and its versions or variants. Baselines are established as milestones in the development and are defined as groups of items that have been formally reviewed and accepted as a basis for further development. Change control is the activity of managing the items under configuration through a formal process to accommodate changes in the requirements or problems found during the development activities. Archiving of the software ensures that the software items remain accessible when they are needed for further development, changes, or future reference.

The Quality Assurance Process is composed of the activities which ensure that the software development organization does "the right thing at the right time in the right way" [Pressman 97]. The basis for quality assurance is the planning activity. In that sense, quality assurance translates to ensuring compliance with the plans and all the processes and activities defined in them.

In spite of what is known about the discipline of software engineering, software often fails. A frequent source of this problem is the use of informal approaches to develop software. [Prowell 99] states:

> "The vast majority of the software today is handcrafted by artisans using craft-based techniques that cannot produce consistent results. These techniques have little in common with the rigorous, theory-based processes characteristic of other engineering disciplines. As a result, software failure is a common occurrence, often with substantial societal and economic consequences. Many software projects simply collapse under the weight of the unmastered complexity and never result in usable systems at all".

The next section covers the area of software failures, reliability, and safety as a preamble to the techniques of software fault tolerance.

3. Software Design Faults

This section discusses the problem of software faults and provides background for software fault tolerance techniques. Because software does not exist in a physical sense, it cannot degrade in the same manner as physical hardware components do. Errors that could arise in the bit pattern representation of the software (e.g., on a hard disk) are really faults in the media used to store the representation, and they can be dealt with and corrected using standard hardware redundancy techniques. The only type of fault possible in software is a design fault introduced during the software development. Software faults are what we commonly call "bugs". According to [Chou 97], software faults are the root cause in a high percentage of operational system failures. The consequences of these failures depend on the application and the particular characteristic of the faults. The immediate effects can range from minor inconveniences (e.g., having to restart a hung personal computer) to catastrophic events (e.g., software in an aircraft that prevents the pilot from recovering from an input error) [Weinstock 97]. From a business perspective, operational failures caused by software faults can translate into loss of potential customers, lower sales, higher warranty repair costs, and losses due to legal actions from the people affected by the failures.

There are four ways of dealing with software faults: prevention, removal, fault tolerance, and input sequence workarounds. Fault prevention is concerned with the use of design methodologies, techniques, and technologies aimed at preventing the introduction of faults into the design. Fault removal considers the use of techniques like reviews, analyses, and testing to check an implementation and remove any faults thereby exposed. The proper use of software engineering during the development processes is a way of realizing fault prevention and fault removal (i.e., fault avoidance). The use of fault avoidance is the standard approach for dealing with software faults and the many developments in the software field target the improvement of the fault avoidance techniques. [Chou 97] states that the software development process usually removes most of the deterministic design faults. This type of fault is activated by the inputs independently of the internal state of the software. A large number of the faults in operational software are state-dependent faults activated by particular input sequences.

Given the lack of techniques that can guarantee that complex software designs are free of design fault, fault tolerance is sometimes used as an extra layer of protection. Software fault tolerance is the use of techniques to enable the continued delivery of services at an acceptable level of performance and safety after a design fault becomes active. The selection of particular fault tolerance techniques is based on system level and software design considerations.

The last line of defense against design faults is to use input sequence workarounds. This is nothing more than accepting that a particular software design has faults and taking those faults as "features". This fix is employed by the system operator to work around known faults while still maintaining availability of the system. An example of this is not entering a particular input sequence to which the system has demonstrated susceptibility. Also if an unknown fault is activated, the operator could try a series of inputs to try to return the system to an acceptable state. The ultimate workaround is to restart the system to recover from a fault. Evidently, this type of system wouldn't be dependable, as defined in [Laprie 92], because the operator cannot rely on proper delivery of services.

From a system perspective, two very important software quality measures are reliability and safety. Reliability is "the probability of failure free operation of a computer program in a specified environment for a specified period of time", where failure free operation in the context of software is interpreted as adherence to its requirements [Pressman 97]. A measure of software reliability is the Mean Time Between Failures (MTBF) [Pressman 97]:

$$MTBF = MTTF + MTTR$$

where MTTF is an acronym for the Mean Time To Failure and MTTR is the Mean Time To Repair. The MTTF is a measure of how long a software item is expected to operate properly before a failure occurs. The MTTR measures the maintainability of the software (i.e., the degree of difficulty in repairing the software after a failure occurs). As mentioned above, software does not degrade with time and its failures are due to the activation of design faults by the input sequences. So, if a fault exists in a piece of software, it will manifest itself the first time the relevant condition occurs [Abbott 90]. What allows the use of reliability as a measure of software quality is the fact that the software is embedded in a stochastic environment that generates input sequences to the software over time [Butler 91]. Some of those inputs will result in software failures. Thus, reliability becomes a weighted measure of correctness, with the weights being dependent on the actual use of the software. Overall, different environments will result in different reliability values [Abbott 90].

Much controversy exists on the use of reliability to characterize software. Reliability is an important quality measure of a system. Since software is viewed as one of many system components, system analysts often consider the estimation of software reliability essential in order to estimate the full system reliability [Parnas 90]. To some people, the apparently random behavior of software failures is nothing more than a reflection of our ignorance and lack of understanding of the software [Parnas 90]. Since software does not fail like hardware does, some reliability engineers argue that software is either correct (reliability 1) or incorrect (reliability 0), and in order to get any meaningful system reliability estimates they assume a software reliability of 1 [Parnas 90]. To others, the notion of reliability of software is of no use unless it can help in reducing the total number of errors in a program [Abbott 90]. An interesting implication of the root cause of the stochastic nature of software failures is that a program with a high number of errors is not necessarily less reliable than another with a lower error count [Parnas 90]; the environment is as important as the software itself in determining the reliability.

Perhaps a more critical issue to the use of software reliability is the physical limitations on achieving accurate estimates. The key assumption that enables the design and reliability estimation of highly reliable hardware systems is that the components fail independently [Butler 91]. This does not apply to software, where the results of one component are directly or indirectly dependent on the results of other components, and thus errors in some modules can result in problems in other modules. When it comes to software reliability estimation, the only reasonable approach is to treat the software as a black box [Parnas 90]. This approach is applicable to systems with low to moderate reliability requirements. However, according to [Butler 91], testing of software with a very high target reliability is prohibitively impractical. In addition, [DO178B] states:

> "...methods for estimating the post-verification probabilities of software errors were examined. The goal was to develop numerical requirements for such probabilities for software in computer-based airborne systems or equipment. The conclusion reached, however, was that currently available methods do not provide results in which confidence can be placed to the level required for this purpose."

Thus, it seems that much research work remains to be done in the area of software reliability modeling before adequate results can be achieved for complex software with high reliability requirements.

Reliability measures the probability of failure, not the consequences of those failures. Software safety is concerned with the consequences of failures from a global system safety perspective. Leveson [Leveson 95] defines software system safety as "the software will execute within a system context without contributing to hazards". A hazard is defined as "a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event)". A system safety design begins by performing modeling and analysis to identify and categorize potential hazards. This is followed by the use of analysis techniques to assign a level of severity and probability of occurrence to the identified hazards. Software is considered as one of the many system components during this analysis. After the analysis is complete, safety related requirements are assigned to the software. [DO178B] states:

> "The goal of [software] fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process."

Thus the function of software fault tolerance is to prevent system accidents (or undesirable events, in general), and mask out faults if possible.

4. Software Fault Tolerance

In this section we present fault tolerance techniques applicable to software. These techniques are divided into two groups [Lyu 95]: single version and multi-version software techniques. Single version techniques focus on improving the fault tolerance of a single piece of software by adding mechanisms into the design targeting the detection, containment, and handling of errors caused by the activation of design faults. Multi-version fault tolerance techniques use multiple versions (or variants) of a piece of software in a structured way to ensure that design faults in one version do not cause system failures. A characteristic of the software fault tolerance techniques is that they can, in principle, be applied at any level in a software system: procedure, process, full application program, or the whole system including the operating system (e.g., [Randell 95]). Also, the techniques can be applied selectively to those components deemed most like to have design faults due to their complexity [Lyu 95].

4.1. Single-Version Software Fault Tolerance Techniques

Single-version fault tolerance is based on the use of redundancy applied to a single version of a piece of software to detect and recover from faults. Among others, single-version software fault tolerance techniques include considerations on program structure and actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity [Lyu 95].

4.1.1. Software Structure and Actions

The software architecture provides the basis for implementation of fault tolerance. The use of modularizing techniques to decompose a problem into manageable components is as important to the efficient application of fault tolerance as it is to the design of a system. The modular decomposition of a design should consider built-in protections to keep aberrant component behavior in one module from propagating to other modules. Control hierarchy issues like visibility (i.e., the set of components that may be invoked directly and indirectly by a particular component [Pressman 97]) and connectivity (i.e., the set of components that may be invoked directly or used by a given component [Pressman 97]) should be considered in the context of error propagation for their potential to enable uncontrolled corruption of the system state.

Partitioning is a technique for providing isolation between functionally independent modules [DO178B]. Partitioning can be performed in the horizontal and vertical dimensions of the modular hierarchy of the software architecture [Pressman 97]. Horizontal partitioning separates the major software functions into highly independent structural branches communicating through interfaces to control modules whose function is to coordinate communication and execution of the functions. Vertical partitioning (or factoring) focuses distributing the control and processing work in a top-down hierarchy, where high level modules tend to focus on control functions and low level modules do most of the processing. Advantages of using partitioning in a design include simplified testing, easier maintenance, and lower propagation of side effects [Pressman 97].

System closure is a fault tolerance principle stating that no action is permissible unless explicitly authorized [Denning 76]. Under the guidance of this principle, no system element is granted any more capability than is needed to perform its function, and any restrictions must be expressly removed before a particular capability can be used. The rationale for system closure is that it is easier (and safer) to handle errors by limiting their chances of propagating and creating more damage before being detected. In a closed environment all the interactions are known and visible, and this simplifies the task of positioning and developing error detection checks. With system closure, any capability damaged by errors only disables a valid action. In a system with relaxed control over allowable capabilities, a damaged capability can result in the execution of undesirable actions and unexpected interference between components.

Temporal structuring of the activity between interacting structural modules is also important for fault tolerance. An atomic action among a group of components is an activity in which the components interact exclusively with each other and there is no interaction with the rest of the system for the duration of the activity [Anderson 81]. Within an atomic action, the participating components neither import nor export any type of information from other non-participating components. From the perspective of the non-participating components, all the activity within the atomic action appears as one and indivisible occurring instantaneously at any time during the duration of the action. The advantage of using atomic actions in defining the interaction between system components is that they provide a framework for error confinement and recovery. There are only two possible outcomes of an atomic action: either it terminates normally or it is aborted upon error detection. If an atomic action terminates normally, its results are complete and committed. If a failure is detected during an atomic action, it is known before hand that only the participating components can be affected. Thus error confinement is defined (and need not be diagnosed) and recovery is limited to the participating set of components.

4.1.2. Error Detection

Effective application of fault tolerance techniques in single version systems requires that the structural modules have two basic properties: self-protection and self-checking [Abbott 90]. The self-protection property means that a component must be able to protect itself from external contamination by detecting errors in the information passed to it by other interacting components. Self-checking means that a component must be able to detect internal errors and take appropriate actions to prevent the propagation of those errors to other components. The degree (and coverage) to which error detection mechanisms are used in a design is determined by the cost of the additional redundancy and the run-time overhead. Note that the fault tolerance redundancy is not intended to contribute to system functionality but rather to the quality of the product. Similarly, detection mechanisms detract from system performance. Actual usage of fault tolerance in a design is based on trade-offs of functionality, performance, complexity, and safety.

Anderson [Anderson 81] has proposed a classification of error detection checks, some of which can be chosen for the implementation of the module properties mentioned above. The location of the checks can be within the modules or at their outputs, as needed. The checks include replication, timing, reversal, coding, reasonableness, and structural checks.

- Replication checks make use of matching components with error detection based on comparison of their outputs. This is applicable to multi-version software fault tolerance discussed in section 4.2.

- Timing checks are applicable to systems and modules whose specifications include timing constraints, including deadlines. Based on these constraints, checks can be developed to look for deviations from the acceptable module behavior. Watchdog timers are a type of timing check with general applicability that can be used to monitor for satisfactory behavior and detect "lost or locked out" components.

- Reversal checks use the output of a module to compute the corresponding inputs based on the function of the module. An error is detected if the computed inputs do not match the actual inputs. Reversal checks are applicable to modules whose inverse computation is relatively straightforward.

- Coding checks use redundancy in the representation of information with fixed relationships between the actual and the redundant information. Error detection is based on checking those relationships before and after operations. Checksums are a type of coding check. Similarly, many techniques developed for hardware (e.g., Hamming, M-out-of-N, cyclic codes) can be used in software, especially in cases where the information is supposed to be merely referenced or transported by a module from one point to another without changing its contents. Many arithmetic operations preserve some particular properties between the actual and redundant information, and can thus enable the use of this type of check to detect errors in their execution.

- Reasonableness checks use known semantic properties of data (e.g., range, rate of change, and sequence) to detect errors. These properties can be based on the requirements or the particular design of a module.

- Structural checks use known properties of data structures. For example, lists, queues, and trees can be inspected for number of elements in the structure, their links and pointers, and any other particular information that could be articulated. Structural checks could be made more effective by augmenting data structures with redundant structural data like extra pointers, embedded counts of the number of items on a particular structure, and individual identifiers for all the items ([TaylorD 80A], [TaylorD 80B], [Black 80], [Black 81]).

Another fault detection tool is run-time checks [Pradhan 96]. These are provided as standard error detection mechanisms in hardware systems (e.g., divide by zero, overflow, underflow). Although they are not application specific, they do represent an effective means of detecting design errors.

Error detection strategies can be developed in an ad-hoc fashion or using structured methodologies. Ad-hoc strategies can be used by experienced designers guided by their judgement to identify the types of checks and their location needed to achieve a high degree of error coverage. A problem with this approach stems from the nature of software design faults. It is impossible to anticipate all the faults (and their generated errors) in a module. In fact, according to Abbott [Abbott 90]:

> "If one had a list of anticipated design faults, it makes much more sense to eliminate those faults during design reviews than to add features to the system to tolerate those faults after deployment. …The problem, of course, is that it is unanticipated design faults that one would really like to tolerate."

Fault trees have been proposed as a design aid in the development of fault detection strategies [Hecht 96]. Fault trees can be used to identify general classes of failures and conditions that can trigger those failures. Fault trees represent a top-down approach which, although not guaranteeing complete coverage, is very helpful in documenting assumptions, simplifying design reviews, identifying omissions, and allowing the designer to visualize component interactions and their consequences through structured graphical means. Fault trees enable the designer to perform qualitative analysis of the complexity and degree of independence in the error checks of a proposed fault tolerance strategy. In general, as a fault tree is elaborated, the structuring of the tree goes from high-level functional concepts to more design dependent elements. Therefore, by means of a fault tree a designer can "tune" a fault detection strategy trading-off independence and requirements emphasis on the tests (by staying with relatively shallow and mostly functional fault trees) versus ease of development of the tests (by moving deeper down the design structure and creating tests that target particular aspects of the design).

4.1.3. Exception Handling

Exception handling is the interruption of normal operation to handle abnormal responses. In the context of software fault tolerance, exceptions are signaled by the implemented error detection mechanisms as a request for initiation of an appropriate recovery. The design of exception handlers requires that consideration be given to the possible events triggering the exceptions, the effects of those events on the system, and the selection of appropriate mitigating actions [Pradhan 96]. [Randell 95] lists three classes of exception triggering events for a software component: interface exceptions, internal local exceptions, and failure exceptions.

- Interface exceptions are signaled by a component when it detects an invalid service request. This type of exception is triggered by the self-protection mechanisms of a module and is meant to be handled by the module that requested the service.

- Local exceptions are signaled by a module when its error detection mechanisms find an error in its own internal operations. These exceptions should be handled by the module's fault tolerant capabilities.

- Failure exceptions are signaled by a module after it has detected an error which its fault processing mechanisms have been unable to handle successfully. In effect, failure exceptions tell the module requesting the service that some other means must be found to accomplish its function.

If the system structure, its actions, and error detection mechanisms are designed properly, the effects of errors will be contained within a particular set of interacting components at the moment the error is detected. This knowledge of error containment is essential to the design of effective exception handlers.

4.1.4. Checkpoint and Restart

For single-version software there are few recovery mechanisms. The most often mentioned is the checkpoint and restart mechanism (e.g., [Pradhan 96]). As mentioned in previous sections, most of the software faults remaining after development are unanticipated, state-dependent faults. This type of fault behaves similarly to transient hardware faults: they appear, do the damage, and then apparently just go away, leaving behind no obvious reason for their activation in the first place [Gray 86]. Because of these characteristics, simply restarting a module is usually enough to allow successful completion of its execution [Gray 86]. A restart, or backward error recovery (see Figure 3), has the advantages of being independent of the damage caused by a fault, applicable to unanticipated faults, general enough that it can be used at multiple levels in a system, and conceptually simple [Anderson 81].

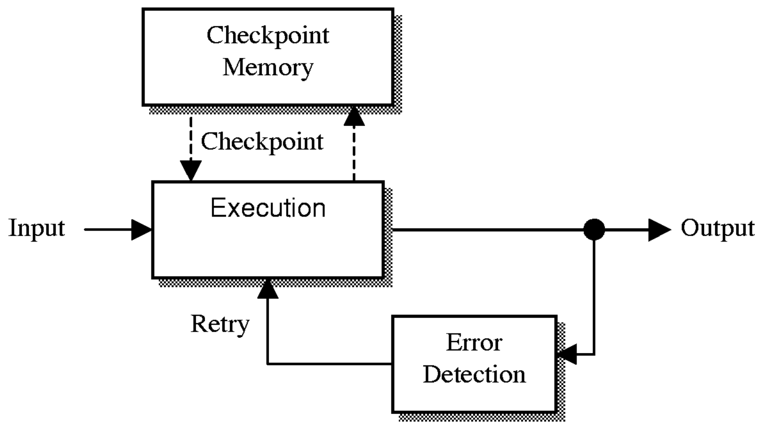

Figure 3: Logical Representation of Checkpoint and Restart

There exist two kinds of restart recovery: static and dynamic. A static restart is based on returning the module to a predetermined state. This can be a direct return to the initial reset state, or to one of a set of possible states, with the selection being made based on the operational situation at the moment the error detection occurred. Dynamic restart uses dynamically created checkpoints that are snapshots of the state at various points during the execution. Checkpoints can be created at fixed intervals or at particular points during the computation determined by some optimizing rule. The advantage of these checkpoints is that they are based on states created during operation, and can thus be used to allow forward progress of execution without having to discard all the work done up to the time of error detection.

An issue of particular importance to backward error recovery is the existence of unrecoverable actions [Anderson 81]. These tend to be associated with external events that cannot be cleared by the simple process of reloading the state and restarting a module. Examples of unrecoverable actions include firing a missile or soldering a pair of wires. These actions must be given special treatment, including compensating for their consequences (e.g., undoing a solder) or just delaying their output until after additional confirmation checks are complete (e.g., do a friend-or-foe confirmation before firing).

4.1.5. Process Pairs

A process pair uses two identical versions of the software that run on separate processors [Pradhan 96] (Figure 4). The recovery mechanism is checkpoint and restart. Here the processors are labeled as primary and secondary. At first the primary processor is actively processing the input and creating the output while generating checkpoint information that is sent to the backup or secondary processor. Upon error detection, the secondary processor loads the last checkpoint as its starting state and takes over the role of primary processor. As this happens, the faulty processor goes offline and executes diagnostic checks. If required, maintenance and replacement is performed on the faulty processor. After returning to service the repaired processor becomes the secondary processor and begins taking checkpoints from the primary. The main advantage of this recovery technique is that the delivery of services continues uninterrupted after the occurrence of a failure in the system.

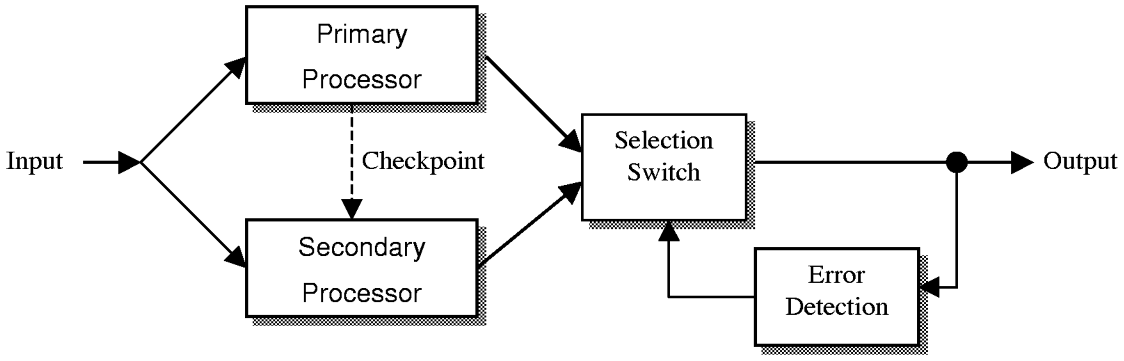

Figure 4: Logical Representation of Process Pairs

4.1.6. Data Diversity

In a previous section we mentioned that the last line of defense against design faults is to use "input sequence workarounds". Data diversity can be seen as the automatic implementation of "input sequence workarounds" combined with checkpoint and restart. Again, the rationale for this technique is that faults in deployed software are usually input sequence dependent. Data diversity has the potential of increasing the effectiveness of the checkpoint and restart by using different input re-expressions on each retry [Ammann 88] (see Figure 5). The goal of each retry is to generate output results that are either exactly the same or semantically equivalent in some way. In general, the notion of equivalence is application dependent.

[Ammann 88] presents three basic data diversity models:

- Input Data Re-Expression, where only the input is changed (Figures 5 and 6);
- Input Re-Expression with Post-Execution Adjustment, where the output is also processed as necessary to achieve the required output value or format (Figure 7);
- Re-Expression via Decomposition and Recombination, where the input is broken down into smaller elements and then recombined after processing to form the desired output (Figure 8).

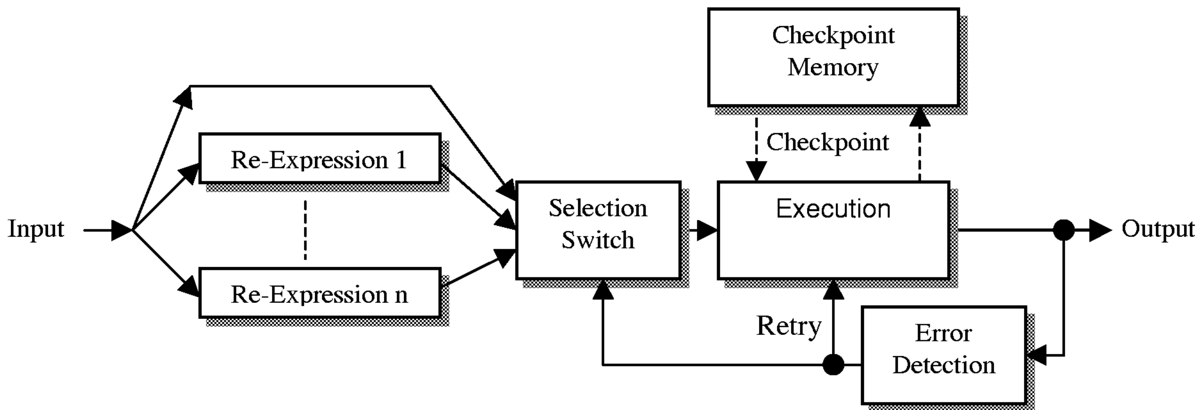

Figure 5: Checkpoint and Restart using Data Diversity (with Input Re-Expression Model)

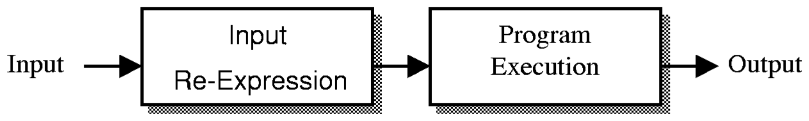

Figure 6: Data Diversity using Input Data Re-Expression

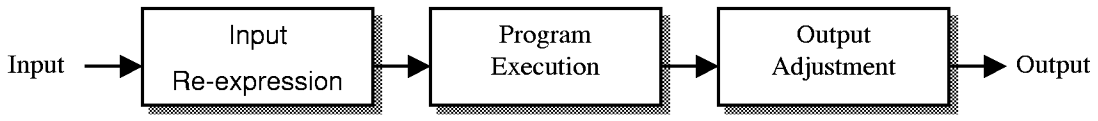

Figure 7: Data Diversity using Input Re-expression with Post-Execution Adjustment

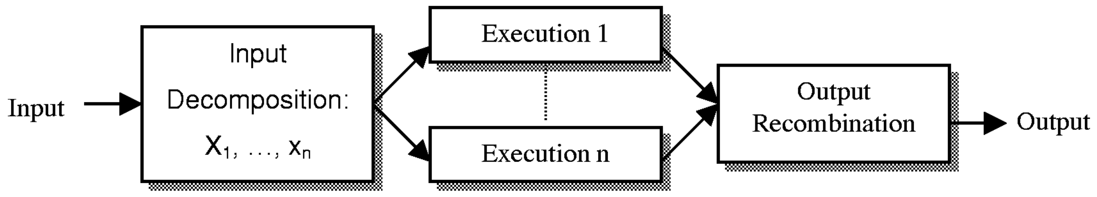

Figure 8: Data Diversity using Re-expression via Decomposition and Recombination

Data diversity is compatible with the Process Pairs technique using different re-expressions of the input in the primary and secondary. Also it seems plausible to be able to incorporate some degree of execution flexibility into the design of the software components to simplify the use of the data diversity concept. Finally, data diversity could be used in conjunction with the multi-version fault tolerance techniques presented in the next section.

4.1.6. Considerations on the use of Checkpointing

We are concerned in this section with the use of checkpointing during execution of a program. The results referenced here assume instantaneous detection of errors from the moment a fault is activated. In real systems these detection delays are non-zero and should be taken into account when selecting a checkpointing strategy. Non-zero detection delays can invalidate checkpoints if the time to detect errors is larger than the interval between checkpoints.

As mentioned above, there exist two kinds of checkpointing that can be used with the checkpoint and restart technique: static and dynamic checkpointing. Static checkpoints take single snapshots of the state at the beginning of a program or module execution. With this approach, the system returns to the beginning of that module when an error is detected and restarts execution all over again. This basic approach to checkpointing provides a generic capability to recover from errors that appear during execution. The use of the single static checkpoint strategy allows the use of error detection checks placed at the output of the module without necessarily having to embed checks in the code. A problem with this approach is that under the presence of random faults, the expected time to complete the execution grows exponentially with the processing requirement. Nevertheless, because of the overhead associated with the use of checkpoints (e.g., creating the checkpoints, reloading checkpoints, restarting), the single checkpoint approach is the most effective when the processing requirement is relatively small.

Dynamic checkpointing is aimed at reducing the execution time for large processing requirements in the presence of random faults by saving the state information at intermediate points during the execution. In general, with dynamic checkpointing it is possible to achieve a linear increase in actual execution time as the processing requirements grow. Because of the overhead associated with checkpointing and restart, there exist an optimal number of checkpoints that optimizes a certain performance measure. Factors that influence the checkpointing performance include the execution requirement, the fault tolerance overhead (i.e., error detection checks, creating checkpoints, recovery, etc.), the fault activation rate, and the interval between checkpoints. Because checkpoints are created dynamically during processing, the error detection checks must be embedded in the code and executed before the checkpoints are created. This increases the effectiveness of the checks and the likelihood that the checkpoints are valid and usable upon error detection.

[Nicola 95] presents three basic dynamic checkpointing strategies: equidistant, modular, and random. Equidistant checkpointing uses a deterministic fixed time between checkpoints. [Nicola 95] shows that for an arbitrary duration between equidistant checkpoints, the expected execution time increases linearly as the processing requirement grows. The optimal time between checkpoints that minimizes the total execution time is shown to be directly dependent on the fault rate and independent of the processing requirements.

Modular checkpointing is the placement of checkpoints at the end of the sub-modular components of a piece of software right after the error detection checks for each sub-module are complete. Assuming a component with a fixed number of sub-modules, the expected execution time is directly related to the processing distribution of the sub-modules (i.e., the processing time between checkpoints). For a given failure rate, a linear dependence between the execution time and the processing requirement is achieved when the processing distribution is the same throughout the modules. For the more general case of a variable processing requirement and an exponential distribution in the duration of the sub-modules, the execution time becomes a linear function of the processing requirements when the checkpointing rate is larger than the failure rate.

In random checkpointing the process of checkpoint creation is triggered at random without consideration of the status of the software execution. Here it is found that the optimal average checkpointing rate is directly dependent on the failure rate and independent of the processing requirements. With this optimal checkpointing rate, the execution time is linearly dependent on the processing requirement.

4.2. Multi-Version Software Fault Tolerance Techniques

Multi-version fault tolerance is based on the use of two or more versions (or "variants") of a piece of software, executed either in sequence or in parallel. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting). The rationale for the use of multiple versions is the expectation that components built differently (i.e, different designers, different algorithms, different design tools, etc) should fail differently [Avizienis 77]. Therefore, if one version fails on a particular input, at least one of the alternate versions should be able to provide an appropriate output. This section covers some of these "design diversity" approaches to software reliability and safety.

4.2.1. Recovery Blocks

The Recovery Blocks technique ([Randell 75], [Randell 95A]) combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected (see Figure 9). Checkpoints are created before a version executes. Checkpoints are needed to recover the state after a version fails to provide a valid operational starting point for the next version if an error is detected. The acceptance test need not be an output-only test and can be implemented by various embedded checks to increase the effectiveness of the error detection. Also, because the primary version will be executed successfully most of the time, the alternates could be designed to provide degraded performance in some sense (e.g., by computing values to a lesser accuracy). Like data diversity, the output of the alternates could be designed to be equivalent to that of the primary, with the definition of equivalence being application dependent. Actual execution of the multiple versions can be sequential or in parallel depending on the available processing capability and performance requirements. If all the alternates are tried unsuccessfully, the component must raise an exception to communicate to the rest of the system its failure to complete its function. Note that such a failure occurrence does not imply a permanent failure of the component, which may be reusable after changes in its inputs or state. The possibility of coincident faults is the source of much controversy concerning all the multi-version software fault tolerance techniques.

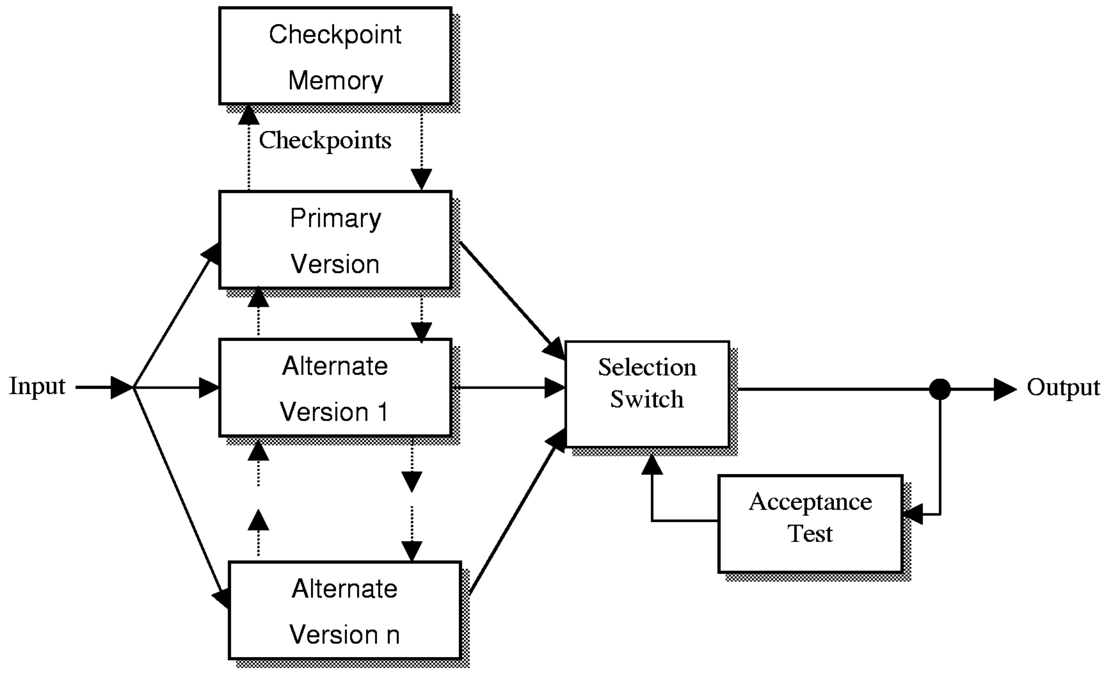

Figure 9: Recovery Block Model

4.2.2. N-Version Programming

N-Version programming [Avizienis 95B] is a multi-version technique in which all the versions are designed to satisfy the same basic requirements and the decision of output correctness is based on the comparison of all the outputs (see Figure 10). The use of a generic decision algorithm (usually a voter) to select the correct output is the fundamental difference of this approach from the Recovery Blocks approach, which requires an application dependent acceptance test. Since all the versions are built to satisfy the same requirements, the use of N-version programming requires considerable development effort but the complexity (i.e., development difficulty) is not necessarily much greater than the inherent complexity of building a single version. Design of the voter can be complicated by the need to perform inexact voting (see section 4.2.7.2). Much research has gone into development of methodologies that increase the likelihood of achieving effective diversity in the final product (see section 4.2.7.1). Actual execution of the versions can be sequential or in parallel. Sequential execution may require the use of checkpoints to reload the state before an alternate version is executed.

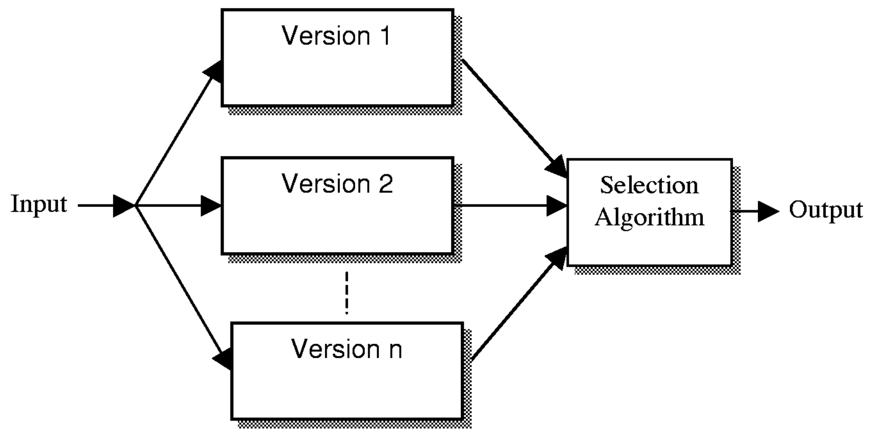

Figure 10: N-Version Programming Model

4.2.3. N Self-Checking Programming

N Self-Checking programming ([Laprie 87], [Laprie 90], [Laprie 95]) is the use of multiple software versions combined with structural variations of the Recovery Blocks and N-Version Programming. N Self-Checking programming using acceptance tests is shown on Figure 11. Here the versions and the acceptance tests are developed independently from common requirements. This use of separate acceptance tests for each version is the main difference of this N Self-Checking model from the Recovery Blocks approach. Similar to Recovery Blocks, execution of the versions and their tests can be done sequentially or in parallel but the output is taken from the highest-ranking version that passes its acceptance test. Sequential execution requires the use of checkpoints, and parallel execution requires the use of input and state consistency algorithms.

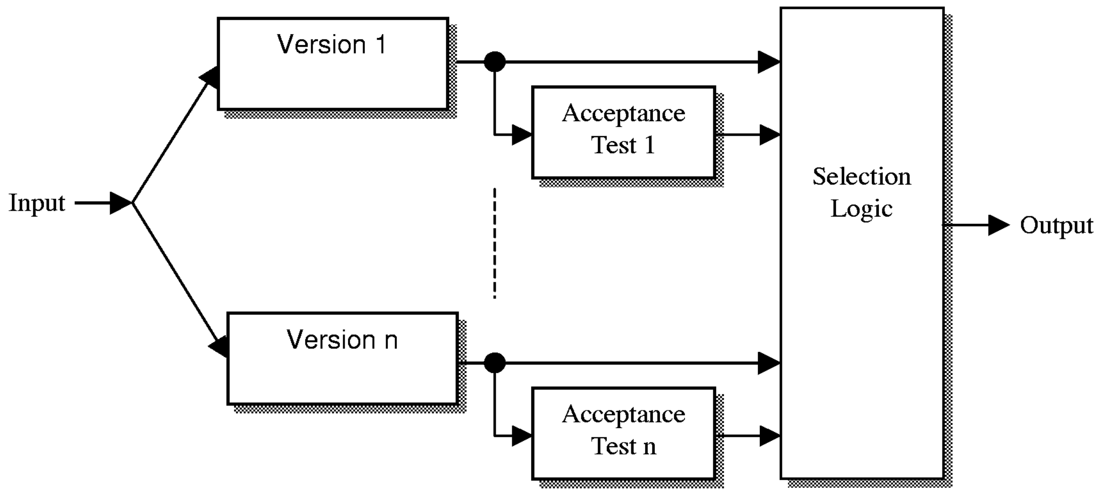

Figure 11: N Self-Checking Programming using Acceptance Tests

N self-checking programming using comparison for error detection is shown in Figure 12. Similar to N-Version Programming, this model has the advantage of using an application independent decision algorithm to select a correct output. This variation of self-checking programming has the theoretical vulnerability of encountering situations where multiple pairs pass their comparisons each with different outputs. That case must be considered and an appropriate decision policy should be selected during design.

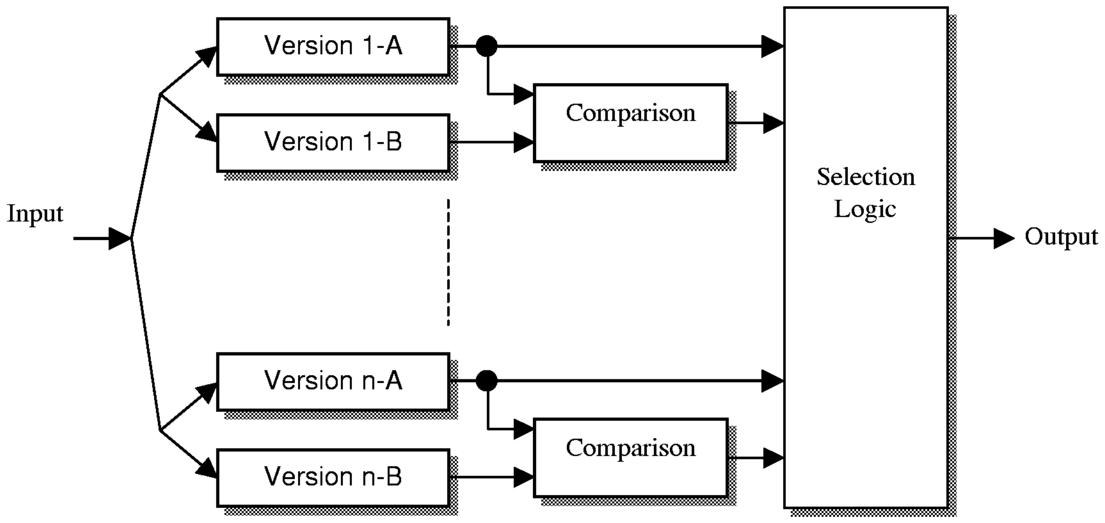

Figure 12: N Self-Checking Programming using Comparison

4.2.4. Consensus Recovery Blocks

The Consensus Recovery Blocks [Scott 87] (see Figure 13) approach combines N-Version Programming and Recovery Blocks to improve the reliability over that achievable by using just one of the approaches. According to Scott [Scott 87], the acceptance tests in the Recovery Blocks suffer from lack of guidelines for their development and a general proneness to design faults due to the inherent difficulty in creating effective tests. The use of voters as in N-Version Programming may not be appropriate in all situations, especially when multiple correct outputs are possible. In that case a voter, for example, would declare a failure in selecting an appropriate output. Consensus Recovery Blocks uses a decision algorithm similar to N-Version Programming as a first layer of decision. If this first layer declares a failure, a second layer using acceptance tests similar to those used in the Recovery Blocks approach is invoked. Although obviously much more complex than either of the individual techniques, the reliability models indicate that this combined approach has the potential of producing a more reliable piece of software [Scott 87]. The use of the word potential is important here because the added complexity could actually work against the design and result in a less reliable system.

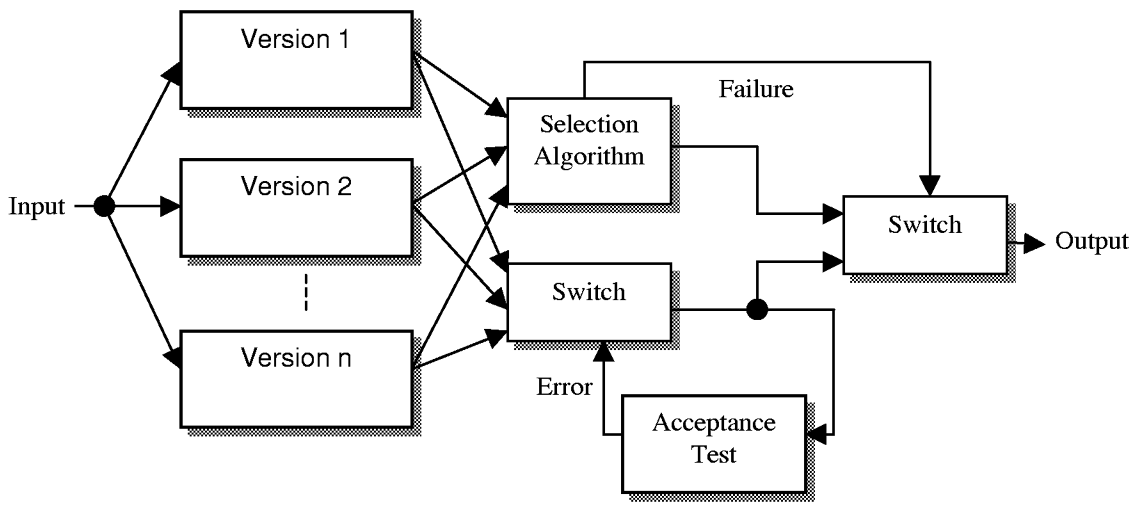

Figure 13: Consensus Recovery Blocks

4.2.6. t/(n-1)-Variant Programming

t/(n-1)-Variant Programming (VP) was proposed by Xu and Randell in [Xu 97]. The main difference between this approach and the ones mentioned above is in the mechanism used to select the output from among the multiple variants. The design of the selection logic is based on the theory of system-level fault diagnosis, which is beyond the scope of this paper (see [Pradhan 96] for a presentation of the subject). Basically, a t/(n-1)-VP architecture consists of n variants and uses the t/(n-1) diagnosibility measure to isolate the faulty units to a subset of size at most (n-1) assuming there are at most t faulty units [Xu 97]. Thus, at least one non-faulty unit exists such that its output is correct and can be used as the result of computation for the module. t/(n-1)-VP compares favorably with other approaches in that the complexity of the selection mechanism grows with order O(n) and it can potentially tolerate multiple dependent faults among the versions. It also has a lower probability of failure than N Self-Checking Programming and N-Version Programming when they use a simple voter as selection logic.

4.2.7. Additional Considerations

Two critical issues in the use of multi-version software fault tolerance techniques are the guaranteeing of independence of failure of the multiple versions and the development of the output selection algorithms.

4.2.7.1. Multi-Version Software Development

Design diversity is "protection against uncertainty" [Bishop 95]. In the case of software design, the uncertainty is in the presence of design faults and the failure modes due to those faults. The goal of design diversity techniques applied to software design is to build program versions that fail independently and with low probability of coincidental failures. If this goal is achieved, the probability of not being able to select a good output at a particular point during program execution is greatly reduced or eliminated.

Due to the complexity of software, the use of design diversity for software fault tolerance is today more of an art rather than a science. Some researchers have developed guidelines and methodologies to achieve a desired level of diversity, but the implementation of design diversity remains a rather complex (and controversial) subject. Presently, the assessment of the achieved improvement over single version software design is difficult (if not impossible) and is based mostly on qualitative arguments.

Perhaps the most comprehensive effort to develop the methodology of multi-version software design was carried out by Algirdas Avizienis and his colleagues at UCLA starting in the 1970s ([Avizienis 85A], [Avizienis 85B], [Avizienis 86], [Avizienis 88], [Avizienis 89], [Avizienis 95A], [Avizienis 95B], [Avizienis 97]). Although focused mainly on software, their research considered the use of design diversity concepts for other aspects of systems like the operating system, the hardware, and the user interfaces. [Avizienis 95B] presents a design methodology for multi-version software that considers the full development effort from the system requirements phase to the operational phase. The objectives of the design paradigm are to reduce the probability of design errors, eliminate any sources of similar design faults and minimize the probability of similar output errors. The presented methodology basically follows the same software engineering principles presented in Section 2 of this paper and it is augmented with activities to support the introduction of design diversity. Decisions to be made include: the selection of the number of software versions; assessment of the required diversity (i.e., diverse specification, design, code, and/or testing); assessment of the use of random (or unmanaged) diversity versus forced (or managed) diversity to minimize the common causes of errors; rules of isolation between the development teams to reduce the probability of similar design errors; the establishment of a coordinating team to serve as an interface between the development teams; and the definition of a rigorous communication protocol between the design teams and the coordinating team to prevent the flow of information that could result in common design errors.

An approach to introducing software fault tolerance is to implement the fault tolerance at the host system level while allowing the application programs to be developed with a minimum of concern for the fault tolerance services. This allows the application developers to focus on their application specialties without being overwhelmed by the fault tolerance aspects of the system. To implement this approach, a framework must be developed which expands the capabilities of the basic operating system with fault tolerance services like cross-version communication, error recovery, and output value selection ([Avizienis 95B], [Bresoud 98]).

As mentioned above, it is hard to determine the benefits of using design diversity for software fault tolerance. There are some inherent difficulties with this approach including the elimination of failure dependencies and the cost of development. Assuming that the development is rigorous and design diversity is adequately applied to the product, there is still the common error source of the identical input profile. [Saglietti 90B] points out that experiments (e,g, [Knight 85], [Knight 86], [Eckhardt 91]) have shown that the probability of error manifestations are not equally distributed over the input space and the probability of coincident errors is impacted by the chosen inputs. Certainly data diversity techniques could be used to reduce the impact of this error source, but the problem of quantifying the effectiveness of the approach still remains.

The cost of using multi-version software is also an important issue. A direct replication of the full development effort, including testing, would certainly be an expensive proposition. Since a supporting execution environment is needed to complete the implementation, the total cost could be prohibitive for some applications. However, cost savings can be effected by judicious use of acceptable alternatives. For example, in some applications where only a small part of the functionality is safety critical, development and production cost can be reduced by applying design diversity only to those critical parts [Bishop 95]. In situations where demonstrating safety attributes to an official regulatory authority tends to be more costly than the actual development cost, design diversity can be used to make a more convincing safety case with a smaller safety assessment effort. Also, when the cost of alternative design assurance techniques is rather high because of the need for specialized staff and tools, the use of design diversity could actually result in a cost saving.

4.2.7.2. Output Selection Algorithms

The basic difference among the multi-version software fault tolerance techniques presented above is in the output selection algorithms. For some techniques, inline acceptance tests simplify the output selection. Some problems with the acceptance tests is that they are highly application dependent, they tend to be difficult to develop, and they cannot test for a specific correct answer but only for "acceptable" values. The ranking of versions based on their individual expected reliabilities can supplement the acceptance tests for cases where multiple versions pass the tests. However, when all the versions are considered equally reliable, the output selection must be based on cross-comparison of the available version outputs, possibly augmented by knowledge of the application. As noted in the particular case of the Consensus Recovery Block approach, the output reliability can be increased by the combination of multiple output selection techniques.

The development of output selection algorithms should consider the consequences of erroneous output selection in terms of critical application issues like safety, reliability, and availability. In general, the output of properly developed versions should be correct for the vast majority of inputs and input sequences, and therefore the reliability of a single version will tend to be relatively good. Nevertheless, for increased reliability in a multi-version arrangement cross-comparison techniques should be designed such that the selected output is correct with a very high probability. For applications where safety is a main concern, it is important that the output selection algorithm be capable of detecting erroneous version outputs and prevent the propagation of bad values to the main output. For these applications the selection algorithm must be given the capability to declare an error condition or initiate an acceptable safe output sequence when it cannot achieve a high confidence of selecting a correct output. In those cases where availability is more important, the output selection can be designed such that it will always produce an output even if it is incorrect. Such approach could be acceptable as long as the program execution is not subsequently dependent on previously generated and possibly erroneous outputs.

[Anderson 86] presents a generic two step structure for the output selection process. The first step is a filtering process where individual version outputs are analyzed by acceptance tests for likelihood of correctness, timing, completeness, and other characteristics. In general the function of the filtering step is to remove any outputs which can be declared bad by direct inline examination. Those outputs that pass the filtering step are then forwarded to the arbitration step where a selection algorithm is used to produce a final output value. Because the values used in the arbitration have been pre-screened, the selection algorithm and the overall approach is likely to be more effective.

Cross-comparison of the available version outputs is usually performed by means of a voting algorithm. [Lorczak 89] presents four generalized voters for use in redundant systems: Formalized Majority Voter, Generalized Median Voter, Formalized Plurality Voter, and Weighted Averaging Techniques. The proposed generalization of the voting techniques is based on a framework of metric spaces[1]. By assuming the use of a metric space, the voters are given the capability of performing inexact voting by declaring values to be equal if their metric distance is less than some predefined threshold ε. In the Formalized Majority Voter version outputs are compared for metric equality and if more than half of the values agree, the voter output is selected as one of the values in the agreement group. The Generalized Median Voter selects the median of the version values as the output. In the metric framework defined here, the median is determined by successively eliminating pairs of values that are the farthest apart until only one value remains (assuming an odd number of versions, of course). The Formalized Plurality Voter partitions the set of version outputs based on metric equality and the selected output is one of the elements in the partition with the largest cardinality. The Weighted Averaging Technique combines the version outputs in a weighted average to produce a new output. The weights can be selected a-priori based on the characteristics of the individual versions and the application. When all the weights are equal this technique becomes a mean selection technique. The weights can also be selected dynamically based on the pair-wise distances of the version outputs [Broen 75] or the success history of the versions measured by some performance metric ([Lorczak 89], [Gersting 91]).

Other voting techniques have been proposed. For example, [Croll 95] proposed a selection function that always produces an acceptable output through the use of artificial intelligence techniques. Specifically, the voting system would behave like a majority or plurality voter when version outputs are sufficiently close to each other and within an acceptable normal range. When there is disagreement, the voter would behave like a weighted averaging voter that assigns the weights based on “fault records” generated from normal cases when the voter is able to form a majority for output selection. These fault records contain information about the disagreements in the value and time domains for each individual version. These records are then used when there is a disagreement beyond the capabilities of the majority or plurality voter. In those cases the output selection would be based on the reliability information contained in the fault records. The authors propose the use of neural networks or genetic algorithms to implement the voter in such a way that its performance is related to the application and the particular characteristics of the software versions. [Bass 95] proposes the use of predictive voters (e.g. Linear Predictor and First Order Predictor) that use history of previous results to produce an expected output value and then select the output based on which version output value is closest to the expected value. [Broen 75] proposed eight weighted average voters for control applications. The voters are designed to produce smooth continuous outputs and to have various degrees of transient failure suppression for failed channels (or versions) of a redundant system.

An important parameter for output selection algorithms is the granularity of the arbitration.

[1] Definition of a metric space [Lorczak 89]: Let X denote the output space of the software. Let d denote a real-valued function defined on the Cartesian Product X x X with the following properties: (1) d(x,y) = 0; (2) d(x,y) = 0 implies x = y; (3) d(x,y) = d(y,x); (4) d(x,z) = d(x,y) + d(y,z), for all x, y, z in X. Then (X,d) is a metric space.

This concept of granularity applies to systems where the version outputs are composites or matrices with various sub-elements. The decision to be made here concerns the level at which output selection will be performed: at coarse level, at fine level, or at some intermediate level. [Kelly 86], [Tso 87], and [Saglietti 91] discuss the problem in some detail. The most obvious characteristic of this perspective on the output selection problem is that coarse level arbitration will result in many more disagreements in output selection among the versions. In general, the higher the level of arbitration, the more likely it is that the selected output will be correct but the likelihood of achieving agreement is diminished. Similarly, applying the output selection algorithm at the lower levels will increase the availability of the system but it will also increase the probability of having inconsistent outputs. An interesting characteristic of voter granularity is that the output selection can dynamically select the granularity. For example, if a voter is unable to detect a majority or plurality at a coarse level, it can automatically switch to progressively lower granularities until agreement is achieved. In doing so, the selection logic would be trading safety and reliability for an increase in system availability. As mentioned above, the characteristics of the output selection algorithm must be based on system level issues like reliability, safety and availability, as well the particular details of the application.

4.3. Fault Tolerance in Operating Systems

Application level software relies on the correct behavior of the operating system. In theory, the previously mentioned techniques to achieve software fault tolerance can be applied to the design of operating systems (e.g., [Denning 76]). However, in general, designing and building operating systems tends to be a rather complex, lengthy and costly endeavor. For safety critical applications it may be necessary to develop custom operating systems through highly structured design processes (e.g., [DO178B]) including highly experienced programmers and advanced verification techniques in order to gain a high degree of confidence on the correctness of the software. For many other applications where time to market and cost are driving factors, such highly structured approaches are not viable. Tradeoffs are necessary in those cases. For example, as mentioned previously, in some applications where only a small part of the functionality is safety critical, development and production cost can be reduced by applying design diversity only to those critical parts. This, of course, requires analysis and insight into the workings of the applications and the operating system.

Another approach to the development of fault tolerant operating systems for mission critical applications is the use of wrappers on off-the-shelf operating systems to boost their robustness to faults. A problem with the use of off-the-shelf software on dependable systems is that the system developers are not sure if the off-the-shelf components are reliable enough for the application [Voas 98A]. It is known that the development process for commercial off-the-shelf software does not consider de facto standards for safety or mission critical applications and the available documentation for the design and validation activities tend to be rather weak [Salles 99]. A point in favor of using commercial operating systems is that they often include the latest developments in operating system technology. Also, widely deployed commercial operating systems could have fewer bugs overall than custom developed software due to the corrective actions performed in response to bug complaints from the users [Koopman 97]. Because modifications to the internals of the operating system could increase the risk of introducing design faults, it is preferred to apply techniques that use the software as is.

A wrapper is a piece of software put around another component to limit what that component can do without modifying the component's source code [Voas 98A]. Wrappers monitor the flow of information into and out of the component and try to keep undesirable values from being propagated. In this manner, the wrapper limits the component's input and output spaces. As with other inline software fault tolerance techniques, wrappers are not a fix-all solution. Their error detection techniques are based on anticipated fault models. As mentioned previously, it is unanticipated faults that are the main source of concern. Also, wrappers cannot protect against illegal outputs explicitly generated by the off-the-shelf component which are not part of the component's specification. Again, the wrapper cannot protect against unanticipated events. Nevertheless, within their inherent limitations, wrappers can be an acceptable technique to achieve the robustness and cost goals for certain applications.

Wrappers have been used as middleware located between the operating system and the application software ([Russinovich 93], [Russinovich 94], [Russinovich 95]). The wrappers (called "sentries" in the referenced work) encapsulate operating system services to provide application-transparent fault tolerant functionality and can augment or change the characteristics of the services as seen by the application layer. In this design the sentries provide the mechanism to implement fault tolerance policies that can be dynamically assigned to particular applications based on the individual fault tolerance, cost and performance needs. The sentries have the capability to implement fault detection and recovery policies through checkpointing and journaling. Journaling is a technique that allows recovery by guiding an application through the replay and synchronization of key input events that occurred from the last checkpointed state to a state close to that just before the fault was detected. The sentries can also perform error correction by performing consistency and validity checks on operating system data structures and doing corrections when errors are detected. Tests performed by the researchers seem to indicate the viability of their approach for effectively implementing fault tolerance policies with acceptable performance penalties.

[Salles 99] proposed the use of wrappers at the microkernel level for off-the-shelf operating systems. The wrappers proposed by these researchers aim at verifying consistency constraints at a semantic level by utilizing information beyond what is available at the interface of the wrapped component. Their approach uses abstractions (i.e., models) of the expected component functionality. Fault containment is based on verifying dynamic predicates defined to assert the correct behavior of the component. As with other error detection techniques, there is a tradeoff between developing costly detailed models of the targeted component that enable more accurate error detection versus the performance achievable using simpler models which might not be as effective in detecting errors. The authors of the referenced work deliberately targeted microkernels instead of the full general-purpose operating system built on top of it because their functionality is easier to understand and manageable from a modeling perspective. The proposed wrappers require access to information internal to the microkernel to verify the predicates and enable corrective actions when a fault is detected. In order to do this the addition of a "metainterface" that would allow observation and control of the microkernel data structures is proposed. This additional interface would protect the source code developed by the microkernel manufacturer while enabling full access to the critical internal data structures. The requirement for the additional metainterface is a drawback of this approach to wrapper design, but it does enable fault tolerance capabilities beyond those achievable by a simpler interface wrapper.

One way to increase the effectiveness of wrappers is by carrying out fault injection experiments on the targeted operating system before designing the wrappers in order to gain knowledge of the weaknesses and pitfalls of the operating system error detection and recovery mechanisms. Section 4.4 covers the area of software fault injection in more detail.

4.4. Software Fault Injection for Fault Tolerance Assessment

Software fault injection (SFI) is the process of testing software under anomalous circumstances involving erroneous external inputs or internal state information. The main reason for using software fault injection is to assess the goodness of a design [Voas98B]. Basically, SFI tries to measure the degree of confidence that can be placed on the proper delivery of services. Since it is very hard to produce correct software, SFI tries to show what could happen when faults are activated. The collected information can be used to make code less likely to hide faults and also less likely to propagate faults to the outputs either by reworking the existing code or by augmenting its capabilities with additional code as done with wrappers [Voas 98B]. SFI can be used to target both objectives of the dependability validation process: fault removal and fault forecasting [Avresky 92]. In the context of fault removal, SFI can be used as part of the testing strategy during the software development process to see if the designed algorithms and mechanisms work as intended. In fault forecasting, SFI is used to assess the fault tolerance robustness of a piece of software (e.g., an off-the-shelf operating system). In this context, SFI enables a performance estimate for the fault tolerance mechanisms in terms of their coverage (i.e., the percentage of faults handled properly) and latency (i.e., the time from fault occurrence to error manifestation at the observation point). The use of SFI has two important advantages over the traditional input sequence test cases [Lai 95]. First, by actively injecting faults into the software we are in effect accelerating the failure rate and this allows a thorough testing in a controlled environment within a limited time frame. Second, by systematically injecting faults to target particular mechanisms we are able to better understand the behavior of that mechanism including error propagation and output response characteristics.

There exist two basic models of software injection: fault injection and error injection. Fault injection simulates software design faults by targeting the code. Here the injection considers the syntax of the software to modify it in various ways with the goal of replacing existing code with new code that is semantically different [Voas 98B]. This "code mutation" can be performed at the source code level before compilation if the source code is available. The mutation can also be done by modifying the text segment of a program's object code after compilation. Error injection, called "data-state mutation" in [Voas 98B], targets the state of the program to simulate fault manifestations. Actual state injection can be performed by modifying the data of a program using any of various available mechanisms: high priority processes that modify lower priority processes with the support of the operating system; debuggers that directly change the program state; message-based mechanisms where one component corrupts the messages received by another component; storage-based mechanisms by using storage (e.g., cache, primary, or secondary memory) manipulation tools; or command-based approaches that change the state by means of the system administration and maintenance interface commands [Lai 95]. An important aspect of both types of fault injection is the operational profile of the software [Voas 98B]. Fault injection is a dynamic-type testing because it must be used in the context of running software following a particular input sequence and internal state profile. The operational profile must be similar to the actual profile in order to realistically assess the robustness of software. However, for the purpose of removing weaknesses in the code or characterizing the code under special or unlikely circumstances, the operational profile can be manipulated to improve other aspects of a test like observability and test duration.

Software fault injection is but one element of the larger area of experimental system testing. A large amount of work has been done in this area by many researchers. The reader is encouraged to review the reported experiments and experimental tools to gather a deeper understanding of the pros and cons of this approach to robustness assessment. Examples of reported works include [Iyer 96], [Kao 93], [Fabre 99], [Koopman 97], [LeeI 95], [Arlat 90].

5. Hardware and Software Fault Tolerance

System fault tolerance is a vast area of knowledge well beyond what can be covered in a single paper. The concepts presented in this section are purposely treated at a high level with details considered only where regarded as appropriate. Readers interested in a more thorough treatment of the concepts of computer system fault tolerance should consult additional reference material (for example, [Pradhan 96], [Suri 95], [Randell 95B]).

5.1. Computer Fault Tolerance

Computer fault tolerance is one of the means available to increase dependability of delivered computational services. Dependability is a quality measure encompassing the concepts of reliability, availability, safety, performability, maintainability and testability [Johnson 96].

- Reliability is the probability that a system continues to operate correctly during a particular time interval given that it was operational at the beginning of the interval.
- Availability is the probability that a system is operating correctly at a given time instant.
- Safety is the probability that the system will perform in a non-hazardous way. A hazard is defined as "a state or condition of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident" [Leveson 95].
- Performability is the probability that the system performance will be equal to or greater than some particular level at a given instant of time.
- Maintainability is the probability that a failed system will be returned to operation within a particular time period. Maintainability measures the ease with which a system can be repaired.
- Testability is a measure of the ability to characterize a system through testing. Testability includes the ease of test development (i.e., controllability) and effect observation (i.e., observability).

The main direct concern for fault tolerant designs is the ability to continue delivery of services in the presence of faults in the system. A fault is an anomalous condition occurring in the system hardware or software. [Suri 95] presents a general fault classification table (see Table 1) which is excellent for understanding the types of faults that fault tolerant designs are called upon to handle. A latent fault is a fault that is present in the system but has not caused errors; after errors occur, the fault is said to be active. Permanent faults are present in the system until they are removed; transient faults appear and disappear on their own with no explicit intervention from the system. Symmetric faults are those perceived identically by all good subsystems; asymmetric faults are perceived differently by the good subsystems. A random fault is caused by the environment (e.g., heat, humidity, vibration, etc.) or by component degradation; generic faults are built-in faults accidentally introduced during design or manufacturing of the system. Benign faults are detectable by all good subsystems; malicious faults are not directly detectable by all good subsystems. The fault count classification is relative to the modularity of the system. A single fault is a fault in a single system module; a group of multiple faults affects more than one module. The time classification is relative to the time granularity. Coincident multiple faults appear during the same time interval; distinct-time faults appear in different time intervals. Independent faults are faults originating from different causes or nature. Common mode faults, in the context of multiple faults, are faults that have the same cause and are present in multiple components.

Table 1: Fault classification (source: [Suri 95])

Criteria	Fault
Activity	Latent vs. Active
Duration	Transient vs. Permanent
Perception	Symmetric vs. Asymmetric
Cause	Random vs. Generic
Intent	Benign vs. Malicious
Count	Single vs. Multiple
Time (multiple faults)	Coincident vs. Distinct
Cause (multiple faults)	Independent vs. Common Mode

The selection of the fault tolerance techniques used in a system depends on the requirements of the application. Fault tolerance is used in a varied set of applications. These include critical, long-life, delayed-maintenance, high-availability, and commercial applications:

- Critical applications require a high degree of confidence on the correct and safe operation of the computer system in order to prevent loss of life or damage to expensive machinery.

- Long-life applications require that computer systems operate as intended with a high probability when the time between scheduled maintenance is extremely long (e.g., on the order of years or tens of years).

- Delayed-maintenance applications involve situations where maintenance actions are extremely costly, inconvenient, or difficult to perform. For this reason the system must be designed to have a high probability of being able to continue operating without requiring unscheduled maintenance actions.

- High-availability applications require a very high probability that the system will be ready to provide the intended service when so requested. This type of system allows frequent service interruptions if they are all short in duration.

- Commercial applications are typically less demanding than the previous applications. The main use of fault tolerance in these systems is to provided added value and prevent nuisance faults from affecting the perceived dependability from a user perspective.

The design of systems with fault tolerance capabilities to satisfy particular application requirements is a complex process loaded with theoretical and experimental analysis in order to find the most appropriate tradeoffs within the design space. [Suri 95] offers a high-level design paradigm (see Table 2) extracted from the more detailed description presented in [Avizienis 87]. System properties to be considered include dependability (i.e., reliability, availability, maintainability, etc), performance, failure modes, environmental resilience, weight, cost, volume, power, design effort, and verification effort. In addition to these, development programs must also weigh in the development risks associated with using technologies that in theory could result in a better system but that could also drive the whole development effort to failure due to the inability of the design team to manage the complexity of the system within a reasonable time frame.

Table 2: Fault Tolerant System Design Paradigm (source: [Suri 95])

1. Identify the classes of faults expected over the life of the system.
2. Specify goals for the system dependability.
3. Partition the system into subsystems, both hardware and software, taking both performance and fault tolerance into account.
4. Select error detection and fault diagnosis algorithms for every subsystem.
5. Devise state recovery and fault removal techniques for every subsystem.
6. Integrate subsystem fault tolerance on a global (system wide) scale.
7. Evaluate the effectiveness of fault tolerance and its relationship with performance.
8. Refine the design by iteration of steps 3 through 7.

Every fault tolerant design must deal with one or more of the following aspects ([Nelson 90], [Anderson 81]):

- Detection: A basic element of a fault tolerant design is error detection. Error detection is a critical prerequisite for other fault tolerant mechanisms.

- Containment: In order to be able to deal with the large number of possible effects of faults in a complex computer system it is necessary to define confinement boundaries for the propagation of errors. Containment regions are usually arranged hierarchically throughout the modular structure of the system. Each boundary protects the rest of the system from errors occurred within it and enable the designer to count on a certain number of correctly operating components by means of which the system can continue to perform its function.

- Masking: For some applications, the timely flow of information is a critical design issue. In such cases, it is not possible to just stop the information processing to deal with detected errors. Masking is the dynamic correction of errors. In general, masking errors is difficult to perform inline with a complex component. Masking, however, is much simpler when redundant copies of the data in question are available.

- Diagnosis: After an error is detected, the system must assess its health in order to decide how to proceed. If the containment boundaries are highly secure, diagnosis is reduced to just identifying the enclosed components. If the established boundaries are not completely secure, then more involved diagnosis is required to identify which other areas are affected by propagated errors.

- Repair/reconfiguration: In general, systems do not actually try to repair component-level faults in order to continue operating. Because faults are either physical or design-related, repair techniques are based on finding ways to work around faults by either effectively removing from operation the affected components or by rearranging the activity within the system in order to prevent the activation of the faults.

- Recovery and Continued Service: After an error is detected, a system must be returned to proper service by ensuring an error-free state. This usually involves the restoration to a previous or predefined state, or rebuilding the state by means of known-good external information.

Redundancy in computer systems is the use of resources beyond the minimum needed to deliver the specified services. Fault tolerance is achieved through the use of redundancy in the hardware, software, information, or time domain ([Johnson 96], [Nelson 90]). In what follows we presents some basic concepts of hardware redundancy to achieve hardware fault tolerance. Good examples of information domain redundancy for hardware fault tolerance are error detecting and correcting codes [Wicker 95]. Time redundancy is the repetition of computations in ways that allow faults to be detected [Johnson 96].

Hardware redundancy can be implemented in static, dynamic, or hybrid configurations. Static (or passive) redundancy techniques do not detect or explicitly perform any reactive action to control errors, but rather rely on masking to simply prevent their propagation beyond predefined error containment boundaries. Dynamic (or active) redundancy techniques use fault detection followed by diagnosis and reconfiguration. Masking is not used in dynamic redundancy, and errors are handled by actively diagnosing error propagation and isolating or replacing faulty components. Hybrid redundancy techniques combine elements of both static and dynamic redundancy. In hybrid redundancy approaches, masking is used prevent the propagation of errors, and error detection, diagnosis, and reconfiguration are used to handle faulty components.

Figure 14 is an example of passive hardware redundancy. Here the modules are replicated multiple times depending on the desired fault tolerance capability. A selection mechanism (usually a voter) is used to mask errors that reach the outputs of the modules. Figure 15 shows a different approach where the voters are moved to the input of the modules to eliminate the single point of failure that is the single voter in Figure 14. This configuration protects the computations performed by the replicated components but requires that redundant components reading the outputs use the same approach to prevent the propagation of errors and single point of failure.

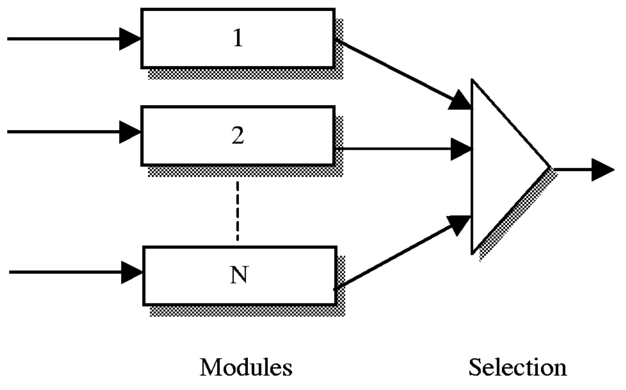

Figure 14: Example of Passive Redundancy

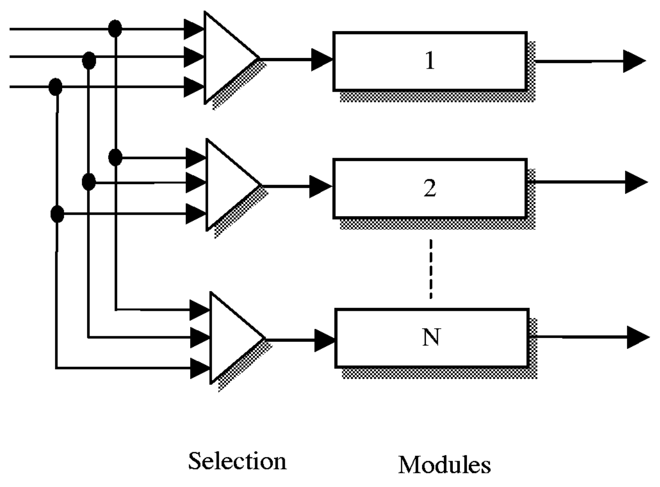

Figure 15: Passive Redundancy with Input Voting

Figure 16 shows an active redundancy approach. In duplication with comparison, error detection is achieved by comparing the outputs of two modules performing the same function. If the outputs of the modules disagree, an error condition is raised followed by diagnosis and repair actions to return the system to operation. In a similar approach only one module would actually perform the intended function with the other component being a dissimilar monitor that checks the outputs looking for errors. Figure 17 shows four modules arranged in a self-checking pair configuration (or dual-dual configuration). In this configuration the comparators perform the error detection function. Normally the output is taken from one of the pairs known as the primary pair, with the other pair acting as a spare or backup. When an error on the primary is detected, the spare is brought online and the primary is taken offline for diagnosis and maintenance if necessary.

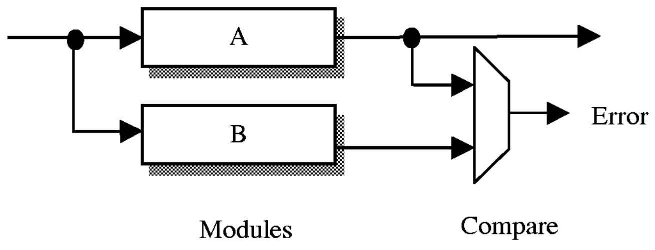

Figure 16: Dynamic Redundancy using Duplication with Comparison

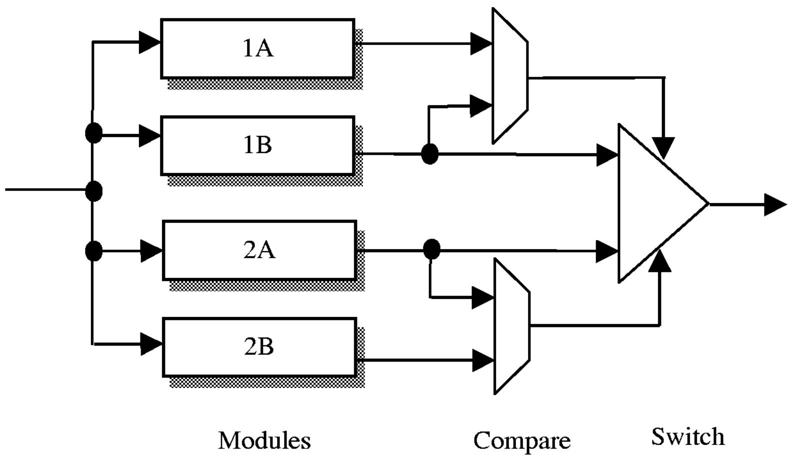

Figure 17: Dynamic Redundancy using Self-Checking Pairs

Figure 18 shows an example of hybrid redundancy using an N-modular masking configuration with spares. Here we are combining the masking approach used in passive redundancy with the error detection, diagnosis, and reconfiguration used in dynamic approaches. The system in Figure 18 uses a set of primary modules to provide inputs to the voter to implement error masking. Simultaneously an error detection component monitors the outputs of the active modules looking for errors. When an error is detected, the faulty module is taken offline for diagnosis and a spare module is brought online to participate in the error-masking configuration. Implemented properly, this configuration has better dependability characteristics than purely passive or active configurations. However, the cost and complexity are higher for the hybrid approach. The selection of one of the three approaches is highly dependent on the application.

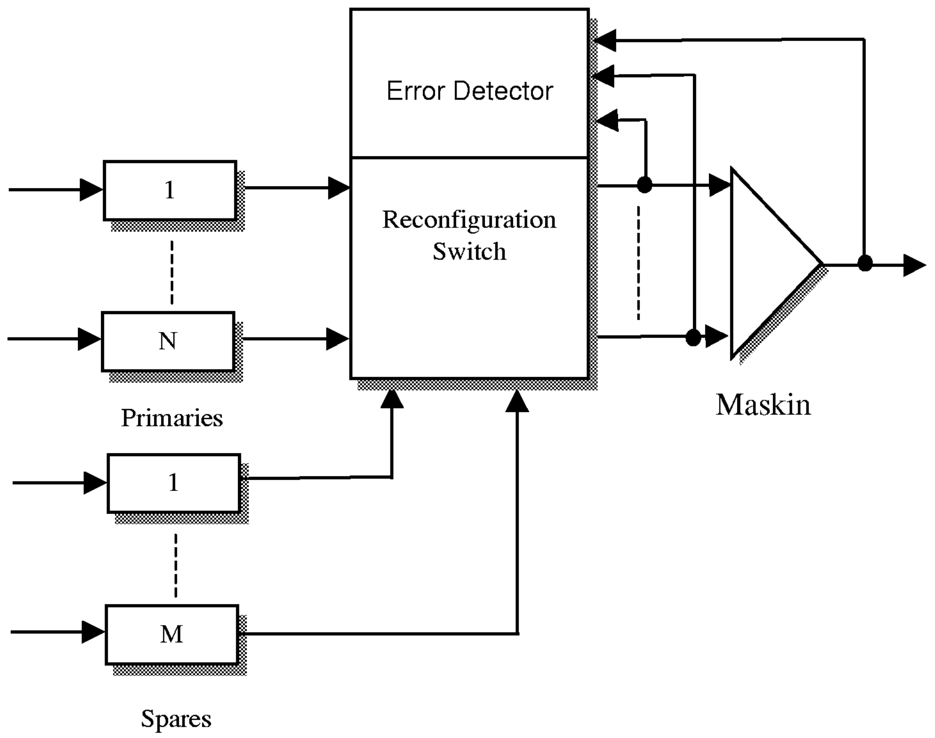

Figure 18: Hybrid Redundancy using N-Modular Redundancy with Spares

It is worth noting that although redundancy is required for fault tolerance, it is not sufficient to just put a group of components together in a "fault tolerant" configuration. How the redundancy is used is as important as the redundancy itself in order to contribute to higher dependability. The following is quoted from [LalaJ 94]:

> "Redundancy alone does not guarantee fault tolerance. The only thing it does guarantee is a higher fault arrival rate compared to a nonredundanct system of the same functionality. For a redundant system to continue correct operation in the presence of a fault, the redundancy must be managed properly. Redundancy management issues are deeply interrelated and determine not only the ultimate system reliability but also the performance penalty paid for fault tolerance."

5.2. Examples of Fault Tolerant Architectures

In this section we present two examples of fault tolerant architectures for safety critical applications. These architectures are used on the flight control computers of the fly-by-wire systems of two types of commercial jet transport aircraft. The first computer is used on the Boeing 777 airplane. The second computer is used on the AIRBUS A320/A330/A340 series aircraft.

5.2.1. B777 Primary Flight Control Computer

The fly-by-wire system of the Boeing 777 airplane departs from old-style mechanical systems that directly connect the pilot's control instruments to the external control surfaces. A fly-by-wire system (see Figure 19) enables the creation of artificial airplane flight characteristics that allow crew workload alleviation and flight safety enhancement, as well as simplifying maintenance procedures through modularization and automatic periodic self-inspection ([Bleeg 88], [Hills 88], [Yeh 96], [Aleska 97], [McKinzie 96]).

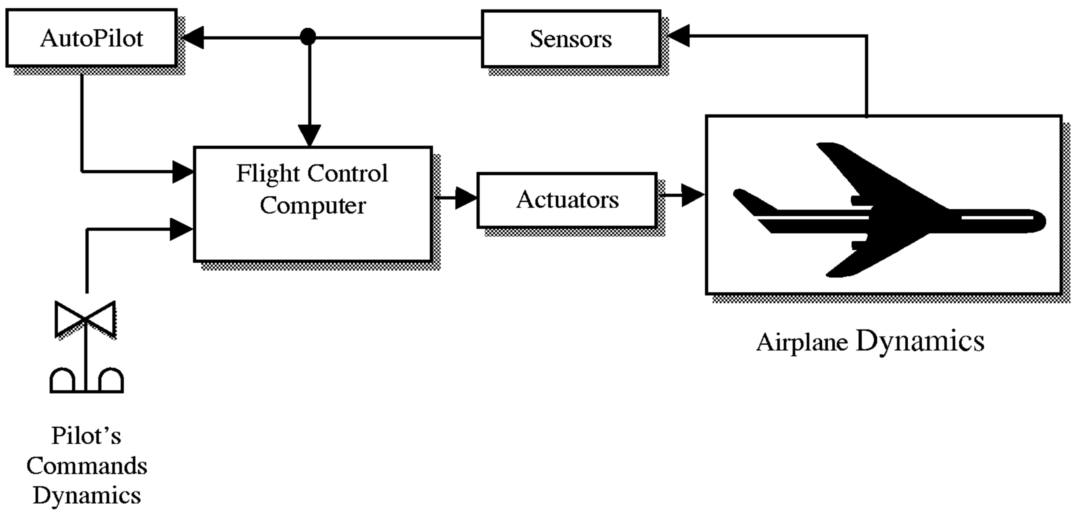

Figure 19: Abstract Representation of a Fly-By-Wire Flight System

Some of the requirements for the 777 flight control computer include:

- No single fault of any kind should cause degradation below MIN-OP (i.e., minimum configuration to meet requirements)
- $1x10^{-10}$ (i.e., 1 in 10 billion) probability of degrading below MIN-OP configuration due to random hardware faults, generic faults, or common mode faults
- No single fault should result in the transmission of erroneous outputs without a failure indication.
- Components should be located in separate compartments throughout the airplane to assure continued safe flight despite physical damage to the airplane and its systems.
- "Never give up" redundancy management strategy for situations when the flight control computer degrades below MIN-OP configuration. This includes considerations for keeping the computer operational if there are any good resources, preventing improper

removal of resources, and recovering resources after being improperly removed.

- Fully automatic redundancy management
- Fully automatic Minimum Dispatch Configuration assessment prior to a flight
- Mean-Time-Between-Maintenance-Actions of 25,000 operating hours assuming 13.6 operating hours per day.

Figure 20 presents the architecture of the B777 flight control computer. It is a triple-triple configuration of three identical channels, each composed of three redundant computation lanes. The computers are connected to the flight control data buses that serve to exchange information among the fly-by-wire system components. Each channel transmits on a preassigned data bus and receives on all the busses. This setup enables the channels to communicate with each other without the possibility of one bad channel interrupting all the communications. The channels are placed in separate equipment bays on the aircraft to allow continued safe flight despite structural damage. Normally the lanes are arranged in a command-monitor-standby arrangement where one lane writes to the bus while the others monitor its operation. The spare lanes in each channel enable rapid reconfiguration in case of a lane failure. The lanes exchange information for redundancy management and for time and data synchronization in order to allow tighter cross-lane monitoring. When the command lane is declared bad, it is taken offline and one of the spare lanes is upgraded to the command assignment. Before sending a computed output to the actuators, the channels perform an exchange of their proposed output values, do a median select, and then finally declare the selected value as the actual computed control value. The channels also exchange information for critical variable equalization to ensure tracking of their outputs within acceptable bounds. The channels must also monitor the operation on the data busses to ensure that data flow is taking place according data bus requirements.

The initial design of this flight control computer was a four by three configuration including hardware and software dissimilarity in all the channels [Hills 88]. Software diversity was to be achieved through the use of different programming languages targeting different lane processors. The final and current implementation uses only one programming language with the executable code being generated by three different compilers still targeting dissimilar lane processors. The lane processors are dissimilar because they are the single most complex hardware devices, and thus there is a perceived risk of design faults associated with their use.

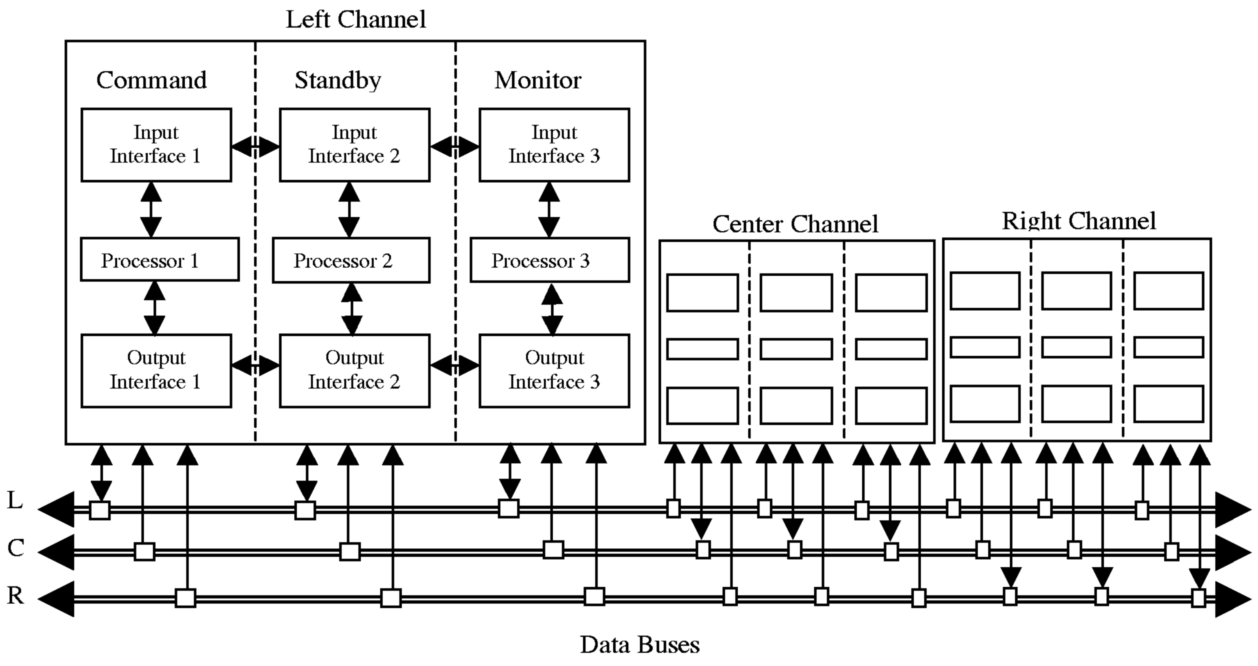

Figure 20: Architecture of B777 Flight Control Computer

(Adapted from [Hills 88] and [Yeh 96])

5.2.2. AIRBUS A320/A330/A340 Flight Control Computer

The requirements for the flight control computer on the Airbus A320/A330/A340 include many of the same considerations as in the B777 fly-by-wire system ([Traverse 91], [Briere 93]). The selected architecture, however, is much different. Figure 21 shows the architecture used on the Airbus aircraft. The basic building block is the fail-stop control and monitor module. Each module is composed of a control computer performing the flight control function and a completely independent monitoring computer performing functions not necessarily identical to the flight control function. The specifications for the control and monitoring computers are developed independently from a common functional specification for the computer module. The software for the control and monitoring computers are designed and built by independent design teams to reduce the likelihood of common design errors. As part of the software development, forced diversity rules are applied to ensure different designs for those areas deemed more complex (and thus, more likely to have errors in the final design). The primary and secondary computer modules are designed by different manufacturers to reduce the likelihood of any kind of software or hardware generic errors. In effect, there are four dissimilar types of computers working together to perform the flight control function. In the basic configuration, the primary module sends its commands to the actuators, with the secondary module remaining in standby. When the primary module fails, it is taken offline and the secondary module takes over the command function. In addition, a second pair of modules (Primary 2 and Secondary 2 in Figure 21) is also available and sending commands to redundant actuators. At any particular time, only one computer module is driving a control surface. Upon detection of a computer or actuator failure, control is passed to another computer based on a predetermined hand over sequence.

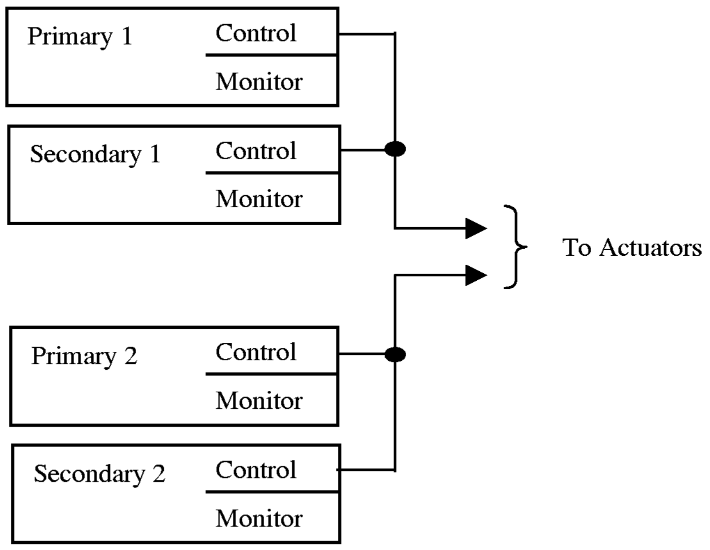

Figure 21: Architecture of A3XX Flight Control Computer
(Adapted from [Traverse 91])

6. Summary and Concluding Remarks

In this paper we have presented a review of software fault tolerance. We gave a brief overview of the software development processes and noted how hard-to-detect design faults are likely to be introduced during development. We noted how software faults tend to be state-dependent and activated by particular input sequences. Although component reliability is an important quality measure for system level analysis, software reliability is hard to estimate and the use of post-verification reliability estimates remains a controversial issue. For some applications software safety is more important than reliability, and fault tolerance techniques used in those applications are aimed at preventing catastrophes. Single version software fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling, and checkpoint and restart. Process pairs exploit the state dependence characteristic of most software faults to allow uninterrupted delivery of services despite the activation of faults. Similarly, data diversity aims at preventing the activation of design faults by trying multiple alternate input sequences. Multiversion techniques are based on the assumption that software built differently should fail differently and thus, if one of the redundant versions fails, at least one of the others should provide an acceptable output. Recovery blocks, N-version programming, N self-checking programming, consensus recovery blocks, and t/(n-1)-variant techniques were presented. Special consideration was given to multiversion software development and output selection algorithms. Operating systems must be given special treatment when designing a fault tolerant software system because of the cost and complexity associated with their development, as well as their criticality for correct system functionality. Software fault injection was presented as a technique to experimentally assess the robustness of software to design faults and errors. Finally, we presented a brief high level overview of fault tolerant computer design followed by the review of two safety critical flight control computer systems.

Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems. The root cause of software design errors is the complexity of the systems. Compounding the problems in building correct software is the difficulty in assessing the correctness of software for highly complex systems. Current research in software engineering focuses on establishing patterns in the software structure and trying to understand the practice of software engineering [Weinstock 97]. It is expected that software fault tolerance research will benefit from this research by enabling greater predictability of the dependability of software [Weinstock 97].

7. Bibliography

[Abbott 90]	Russell J. Abbott, *Resourceful Systems for Fault Tolerance, Reliability, and Safety*, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 – 68.
[Aleska 97]	Brian D. Aleska and Joseph P. Carter, *Boeing 777 Airplane Information Management System Operational Experience*, AIAA/IEEE Digital Avionics Systems Conference, Vol. II, 1997, pp. 3.1-21 – 3.1-27.
[Ammann 88]	Paul E. Ammann and John C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*, IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 418 - 425.
[Anderson 81]	T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice/Hall, 1981.
[Anderson 85A]	T. Anderson, Ed., *Resilient Computing Systems*, Vol. I, John Wiley & Sons, 1985.
[Anderson 85B]	T. Anderson, et al, *An Evaluation of Software Fault Tolerance in a Practical System*, Digest of Papers FTCS-15: The Fifteenth Annual International Symposium on Fault-Tolerant Computing, June 1985, pp. 140 - 145.
[Anderson 86]	Tom Anderson, *A Structured Mechanism for Diverse Software*, Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp. 125 – 129.
[Andrews 79A]	Dorothy M. Andrews, *Using Executable Assertions for Testing and Fault Tolerance*, Digest of Papers: The Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS 9), June 1979, pp. 102 - 105.
[Andrews 79B]	D. M. Andrews, *Software Fault Tolerance Through Executable Assertions*, 12th Asilomar Conference on Circuits and Systems and Computers, November 1979, pp. 641 – 645.

[Arlat 90]	Jean Arlat, et al, *Fault Injection for Dependability Validation: A Methodology and Some Applications*, IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990, pp. 166 – 182.
[Avizienis 77]	A. Avizienis and L. Chen, *On the Implementation of N-Version Programming for Software Fault Tolerance During Execution*, Proceedings of the IEEE COMPSAC'77, November 1977, pp. 149 – 155.
[Avizienis 85A]	A. Avizienis, et al, *The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software*, Digest of Papers: The Fifteenth Annual International Symposium on Fault-Tolerant Computing (FTCS 15), Ann Arbor, Michigan, June 19 – 21, 1985, pp. 126 – 134.
[Avizienis 85B]	Algirdas Avizienis, *The N-Version Approach to Fault-Tolerant Software*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 290 - 300.
[Avizienis 86]	Algirdas Avizienis and Jean-Claude Laprie, *Dependable Computing: From Concepts to Design Diversity*, Proceedings of the IEEE, Vol. 74, No. 5, May 1986, pp. 629 – 638.
[Avizienis 87]	Algirdas Avizienis, *A Design Paradigm for Fault Tolerant Systems*, Proceedings of the AIAA/IEEE Digital Avionics Systems Conference (DASC), Washington, D.C., 1987.
[Avizienis 88]	Algirdas Avizienis, *In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software*, Digest of Papers FTCS-18: The Eighteenth International Symposium on Fault-Tolerant Computing, June 27 – 30, 1988, pp. 15 - 22.
[Avizienis 89]	Algirdas Avizienis, *Software Fault Tolerance*, Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, 1989, pp. 491 – 498.
[Avizienis 95A]	Algirdas Avizienis, *Dependable Computing Depends on Structured Fault Tolerance*, Proceedings of the 1995 6th International Symposium on Software Reliability Engineering, Toulouse, France, 1995, pp. 158 – 168.
[Avizienis 95B]	Algirdas Avizienis, *The Methodology of N-Version Programming*, in R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.
[Avizienis 97]	Algirdas Avizienis, *Toward Systematic Design of Fault-Tolerant Systems*, Computer, April 1997, pp. 51 – 58.
[Avresky 92]	Dimitri Avresky, et al, *Fault Injection for the Formal Testing of Fault Tolerance*, Digest of Papers of the Twenty-Second International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, July 8 – 10, 1992, pp. 345 - 354.
[Ayache 96]	S. Ayache, et al, *Formal Methods for the Validation of Fault Tolerance in Autonomous Spacecraft*, Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing, Sendai, Japan, June 25 – 27, 1996, pp. 353 – 357.
[Babikyan 90]	Carol A. Babikyan, *The Fault Tolerant Processor Operating System Concepts*

and Performance Measurement Overview, IEEE/AIAA/NASA 9th Digital Avionics Systems Conference, October 1990, pp. 366 – 371.

[Bass 95] J. M. Bass, *Voting in Real-Time Distributed Computer Control Systems*, PhD Thesis, University of Sheffield, October 1995.

[Bass 97] J. M. Bass, et al, *Experimental Comparison of Voting Algorithms in Cases of Disagreement*, Proceedings of the 1997 23rd EUROMICRO Conference, 1997, pp. 516 – 523.

[Beedubail 95] Ganesha Beedubail, et al, *An Algorithm for Supporting Fault Tolerant Objects in Distributed Object Oriented Operating Systems*, Proceedings of the 4th International Workshop on Object Orientation in Operating Systems, 1995, pp. 142 – 148.

[Bishop 95] Peter Bishop, *Software Fault Tolerance by Design Diversity*, in R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.

[Black 80] J. P. Black, et al, *Introduction to Robust Data Structures*, Digest of Papers FTCS-10: The Eleventh Annual International Symposium on Fault-Tolerant Computing, October 1 – 3, 1980, pp. 110 - 112.

[Black 81] J. P. Black, et al, *A Compendium of Robust Data Structures*, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-Tolerant Computing, June 24 – 26, 1981, pp. 129 - 131.

[Bleeg 88] Robert J. Bleeg, *Commercial Jet Transport Fly-By-Wire Architecture Considerations*, AIAA/IEEE 8th Digital Avionics Systems Conference, October 1988, pp. 399 – 406.

[Blough 90] Douglas M. Blough and Gregory F. Sullivan, *A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems*, Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, AL, 1990, pp. 136 – 145.

[Boi 81] L. Boi, et al, *Exception Handling and Error Recovery Techniques in Modular Systems – An Application to the ISAURE System* -, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-Tolerant Computing, 1981, pp. 62 - 64.

[Bresoud 98] Thomas C. Bressoud, *TFT: A Software System for Application-Transparent Fault Tolerance*, Digest of Papers: Twenty-Eight Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23 – 25, 1998, pp. 128 – 137.

[Briere 93] Dominique Briere and Pascal Traverse, *AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems*, Digest of Papers FTCS-23: The Twenty-Third International Symposium on Fault-Tolerant Computing, June 1993, pp. 616 - 623.

[Brilliant 89] Susan S. Brilliant, et al, *The Consistent Comparison Problem in N-Version Software*, IEEE Transactions on Software Engineering, Vol. 15, No. 11, November 1989, pp. 1481 - 1485.

[Broen 75] R. B. Broen, *New Voters for Redundant Systems*, Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control, March 1975, pp. 41 – 45.

[Brooks 87] Frederick P. Brooks, Jr., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, March 1987, pp. 10 - 19.

[Butler 91] Ricky W. Butler and George B. Finelli, *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*, IEEE Transactions on Software Engineering, Vol. 19, No. 1, January 1991, pp. 3 – 12.

[Caglayan 85] Alper K. Caglayan and Dave E. Eckhardt, Jr., *Systems Approach to Software Fault Tolerance*, AIAA Paper 85-6018, 1985.

[Cha 87] Sung D. Cha, *An Empirical Study of Software Error Detection Using Self-Checks*, Digest of Papers FTCS-17: The Seventh International Symposium on Fault-Tolerant Computing, July 6 – 8, 1987, pp. 156 - 161.

[Chandra 98] Subhachandra Chandra and Peter M. Chen, *How Fail-Stop are Faulty Programs?*, Digest of Papers: Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23 – 25, 1998, pp. 240 – 249.

[Chang 94] Elizabeth Chang and Tharam S. Dillon, *Achieving Software Reliability and Fault Tolerance Using the Object Oriented Paradigm*, Computer Systems Science and Engineering, Vol. 9, No. 2, April 1994, pp. 118 – 121.

[Chisholm 99] G. H. Chisholm and A S. Wojcik, *An Application of Formal Analysis to Software in a Fault-Tolerant Environment*, IEEE Transactions on Computers, Vol. 48, No. 10, October 1999, pp. 1053 – 1064.

[Christmansson 94] J. Christmansson, et al, *Dependable Flight Control System Using Data Diversity with Error Recovery*, Computer Systems Science and Engineering, Vol. 9, No. 2, April 1994, pp. 142 – 150.

[Christmansson 98] J. Christmansson, et al, *Dependable Flight Control System Using Data Diversity with Error Recovery*, Doktorsavhandlingar vid Chalmers Tekniska Hogskola, No. 1362, Sweden, 1998, pp. A.1 – A.10.

[Cristian 80] Flaviu Cristian, *Exception Handling and Software-Fault Tolerance*, Digest of Papers FTCS-10: The 10th International Symposium on Fault-Tolerant Computing, October 1 – 3, 1980, pp. 97 - 103.

[Cristian 82] Flaviu Cristian, *Exception Handling and Software Fault Tolerance*, IEEE Transactions on Computers, Vol. C-31, No. 6, June 1982, pp. 531 - 540.

[Croll 95] P. R. Croll, et al, *Dependable, Intelligent Voting for Real-Time Control Software*, Engineering Applications of Artificial Intelligence, vol. 8, no. 6, December 1995, pp. 615 – 623.

[Chou 97] Timothy C. K. Chou, *Beyond Fault Tolerance*, IEEE Computer, April 1997, pp. 47 – 49.

[Damm 86] Andreas Damm, *The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems*, Digest of Papers: 16th Annual International

Symposium on Fault-Tolerant Computing Systems (FTCS 16), Vienna, Austria, July 1 – 4, 1986, pp. 171 – 176.

[Daughan 94] Michael G. Daughan, *Seawolf Submarine Ship Control System: A Case Study of a Fault-Tolerant Design*, Naval Engineering Journal, January 1994, pp. 54 – 70.

[David 93] Philippe David and Claude Guidal, *Development of a Fault Tolerant Computer System for the HERMES Space Shuttle*, Digest of Papers: The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS 23), Toulouse, France, June 22 – 24, 1993, pp. 641 – 646.

[Deck 98] Michael Deck, *Software Reliability and the "Cleanroom" Approach: A Position Paper*, Proceedings of the Annual Reliability and Maintainability Symposium, Anaheim, CA, January 19 – 22, 1998, pp. 218 – 223.

[Denning 76] Peter J. Denning, *Fault Tolerant Operating Systems*, ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 359 - 389.

[Dingman 95] Christopher P. Dingman, et al, *Measuring Robustness of a Fault Tolerant Aerospace System*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 27 – 30, 1995, p. 522 – 527.

[Dong 99] Libin Dong, et al, *Implementation of a Transient-Fault-Tolerance Scheme on DEOS*, Proceedings of the 1999 5th IEEE Real-Time Technology and Applications Symposium, 1999, pp. 56 – 65.

[Doyle 95] Stacy A. Doyle and Jane Latin Mackey, *Comparative Analysis of Two Architectural Alternatives for the N-Version Programming (NVP) System*, Proceedings of the 1995 Annual Reliability and Maintainability Symposium, 1995, pp. 275 – 282.

[DO178B] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc, 1992.

[Echtle 95] Klaus Echtle and Tomislav Lovric, *Hardware and Software Fault Tolerance Using Fail-Silent Virtual Duplex Systems*, Proceedings of the 1995 Conference on Fault Tolerant Parallel and Distributed Systems, 1995, pp. 10 – 17.

[Eckhardt 85A] Dave E. Eckhardt and Larry D. Lee, *A Theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors*, National Aeronautics and Space Administration (NASA), Technical Memorandum 86369, January 1985.

[Eckhardt 85B] Dave E. Eckhardt and Larry D. Lee, *An Analysis of the Effects of Coincident Errors on Multi-Version Software*, AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, October 21 – 23, 1985, pp. 370 - 373.

[Eckhardt 88] Dave E. Eckhardt and Larry D. Lee, *Fundamental Differences in the Reliability of N-Modular Redundancy and N-Version Programming*, The Journal of Systems and Software 8, 1988, pp. 313 – 318.

[Eckhardt 91] Dave E. Eckhardt, et al, *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, IEEE Transactions on Software Engineering, Vol. 17, No. 7, July 1991, pp. 692 - 702.

[Elmendorf 72]	William R. Elmendorf, *Fault-Tolerant Programming*, Digest of Papers FTCS-2: The 2nd Annual International Sympoium on Fault Tolerant Computing, June 1972, pp. 79 - 83.
[Everett 98]	William Everett, Samual Keene and Allen Nikora, *Applying Software Reliability Engineering in the 1990s*, IEEE Transactions on Reliability, Vol. 47, No. 3-SP, September 1998, pp. 372-SP – 378-SP.
[Fabre 88]	Jean-Charles Fabre, et al, *Saturation: Reduced Idleness for Improved Fault-Tolerance*, Digest of Papers: The Eighteenth International Symposium on Fault-Tolerant Computing (FTCS 18), June 1988, pp. 200 – 205.
[Fabre 99]	J. C. Fabre, et al, *Assessment of COTS Microkernels by Fault Injection*, Proceedings IFIP DCCA-7, 1999, pp. 19 – 38.
[Fraser 99]	Timothy Fraser, et al, *Hardening COTS Software with Generic Software Wrappers*, Proceedings of the 1999 IEEE Computer Society Symposium on Research in Security and Privacy, 1999, pp. 2 – 16.
[Gersting 91]	Judith Gersting, et al, *A Comparison of Voting Algorithms for N-Version Programming*, Proceedings of the 24th Annual Hawaii International Conference on System Sciences, Volume II, January 1991, pp. 253 – 262.
[Gluch 86]	David P. Gluch and Michael J. Paul, *Fault-Tolerance in Distributed Fly-By-wire Flight Control Systems*, Proceedings IEEE/AIAA 7th Digital Avionics Systems Conference, October 1986, pp. 507 – 514.
[Goseva 93]	K. Goseva-Popstojanova and A. Grnarov, *N Version Programming with Majority Decision: Dependability Modeling and Evaluation*, Microprocessing and Microprogramming, Vol. 38, No. 1 - 5, September 1993, pp. 811 – 818.
[Gray 86]	Jim Gray, *Why Do Computers Stop and What Can Be Done About It?*, Proceedings of the Fifth Symposium On Reliability in Distributed Software and Database Systems, January 13-15, 1986, pp. 3 - 12.
[Greeley 87]	Gregory L. Greeley, *The Effects of Voting Algorithms on N-Version Software Reliability*, Master of Science Thesis, Massachusetts Institute of Technology, June 1987.
[Guerraoui 97]	Rachid Guerraoui and Andre Schipper, *Software-Based Replication for Fault Tolerance*, IEEE Computer, April 1997, pp. 68 - 74.
[Hamilton 96]	Major Lynne Hamilton-Jones, et al, *Software Technology for Next-Generation Strike Fighter Avionics*, Proceedings of the 1996 15th AIAA/IEEE Digital Avionics Systems Conference, October 1996, pp. 37 – 42.
[Hamlet 93]	Dick Hamlet and Jeff Voas, *Faults in Its Sleeve: Amplifying Software Reliability Testing*, Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), June 1993, pp. 88 - 98.
[Hecht 79]	Herbert Hecht, *Fault-Tolerant Software*, IEEE Transactions on Reliability, Vol. R-28, No. 3, August 1979, pp. 227 – 232.
[Hecht 86]	Herbert Hecht and Myron Hecht, *Software Reliability in the System Context*,

IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 51 – 58.

[Hecht 96] Herbert Hecht and Myron Hecht, *Fault-Tolerance in Software*, in Fault-Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, 1996.

[Hills 88] Andy D. Hills and Dr. Nisar A. Mirza, *Fault Tolerant Avionics*, AIAA/IEEE 8th Digital Avionics Systems Conference, October 1988, pp. 407 – 414.

[Huang 84] Kuang-Hua Huang and Jacob A. Abraham, *Algorithm-Based Fault Tolerance for Matrix Operations*, IEEE Transactions on Computers, Vol. C-33, No. 6, June 1986, pp. 518 - 528.

[IEEE 93] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.

[Iyer 96] Ravishankar K. Iyer and Dong Tang, *Experimental Analysis of Computer System Dependability*, in Fault Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, 1996, pp. 282 – 392.

[Jewett 91] Doug Jewett, *Integrity S2: A Fault-Tolerant Unix Platform*, Digest of Papers Fault-Tolerant Computing: The Twenty-First International Symposium, Montreal, Canada, June 25 – 27, 1991, pp. 512 – 519.

[Johnson 96] Barry W. Johnson, *An Introduction to the Design and Analysis of Fault-Tolerant Systems*, in Fault-Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, Inc., 1996, pp. 1 – 87.

[Kanekawa 89] Nobuyasu Kanekawa, et al, *Dependable Onboard Computer Systems with a New Method – Stepwise Negotiating Voting*, Digest of Papers: The Nineteenth International Symposium on Fault-Tolerant Computing (FTCS-19), 1989, pp. 13 – 19.

[Kao 93] Wei-lun Kao, et al, *FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults*, IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993, pp. 1105 – 1118.

[Kelly 86] John P. J. Kelly, et al, *Multi-Version Software Development*, Proceeding of the Fifth IFAC Workshop, Safety of Computer Control Systems, October 1986, pp. 43 - 49.

[Kelly 91] John P. J. Kelly, *Implementing Design Diversity to Achieve Fault Tolerance*, IEEE Software, July 1991, pp. 61 – 71.

[Kersken 92] M. Kersken and F. Saglietti, Editors, Software Fault Tolerance: Achievement and Assessment Strategies, *Springer-Verlag, 1992.*

[Kieckhafer 87] R. M. Kieckhafer, *Task Reconfiguration in a Distributed Real-Time System*, Proceedings Real-Time Systems Symposium, December 1987, pp. 25 – 32.

[Kieckhafer 88] Roger M. Kieckhafer, et al, *The MAFT Architecture for Distributed Fault Tolerance*, IEEE Transaction on Computers, Vol. 37, No. 4, April 1988, pp. 398 – 405.

[Kieckhafer 89] Roger M. Kieckhafer, *Fault-Tolerant Real-Time Task Scheduling in the MAFT Distributed System*, Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences: Architecture Track, 1989, pp. 143 – 151.

[KimH 95] Hyun C. Kim and V.S.S. Nair, *Application Layer Software Fault Tolerance for Distributed Object-Oriented Systems*, Proceedings of the Nineteenth Annual International Computer Software & Applications Conference (COMPSAC '95), August 1995, pp. 199 – 204.

[KimH 97] Hyun C. Kim and V.S.S. Nair, *Software Fault Tolerance for Distributed Object Based Computing*, Journal of Systems and Software, Vol. 39, No. 2, November 1997, pp. 103 – 117.

[KimK 97] K. H. Kim, and Chittur Subbaraman, *Fault-Tolerant Real-Time Objects*, Communications of the ACM, Vol. 40, No. 1, January 1997, pp. 75 – 82.

[Knight 85] J. C. Knight, et al, *A Large Scale Experiment in N-Version Programming*, Digest of Papers FTCS-15: The 15th Annual International Conference on Fault Tolerant Computing, June 1985, pp. 135 - 139.

[Knight 86] J. C. Knight and Nancy G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 96 - 109.

[Knight 91] J. C. Knight and P. E. Ammann, *Design Fault Tolerance*, Reliability Engineering and System Safety, Vol. 32, No. 1-2, 1991, pp. 25 – 49.

[Kojo 88] Takashi Kojo, et al, *Fault Tolerant Real-Time Operating System for 32 Bit Microprocessor V69/V70*, NEC Research and Development, No. 89, April 1988, pp. 123 – 129.

[Koo 87] Richard Koo and Sam Toueg, *Checkpointing and Rollback-Recovery for Distributed Systems*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, pp. 23 - 31.

[Koopman 97] Philip Koopman, et al, *Comparing Operating Systems Using Robustness Benchmarks*, Proceedings of the 1997 16th IEEE Symposium on Reliable Distributed Systems, October 1997, pp. 72 – 79.

[Koopman 99] Philip Koopman and John DeVale, *Comparing the Robustness of POSIX Operating Systems*, Digest of Papers: Twenty-Ninth Annual International Symposium Fault-Tolerant Computing, Madison, Wisconsin, June 15 – 18, 1999, pp. 30 – 37.

[Kopetz 99] Hermann Kopetz and Dietmar Millinger, *The Transparent Implementation of Fault Tolerance in the Time-Triggered Architecture*, Dependable Computing for Critical Applications 7, Volume 12, A. Avizienis, H. Kopetz, J.C. Laprie, editors, 1999, pp. 192 – 205.

[Kropp 98] Nathan P. Kropp, et al, *Automated Robustness Testing of Off-the-Shelf Software Components*, Digest of Papers: Twenty-Eighth Annual International Symposium Fault-Tolerant Computing, June 1998, pp. 230 – 239.

[Lai 95]	Ming-Yee Lai and Steve Y. Wang, *Software Fault Insertion Testing for Fault Tolerance*, in Software Fault Tolerance, Michael R. Lyu, editor, John Wiley & Sons, 1995, pp. 315 – 333.
[LalaJ 86]	Jaynarayan H. Lala, *A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications*, Digest of Papers: 16th Annual International Symposium on Fault-Tolerant Computing Systems (FTCS-16), Vienna, Austria, July 1 – 4, 1986, pp. 338 – 343.
[LalaJ 88]	Jaynarayan H. Lala and Linda S. Alger, *Hardware and Software Fault Tolerance: A Unified Architectural Approach*, Digest of Papers FTCS-18: The Eighteenth International Symposium on Fault-Tolerant Computing, June 1988, pp. 240 - 245.
[LalaJ 91]	Jaynarayan H. Lala, et al, *A Design Approach for Ultrareliable Real-Time Systems*, IEEE Computer, Vol. 24, No. 5, May 1991, pp. 12 – 22.
[LalaJ 93]	Jaynarayan H. Lala and Richard E. Harper, *Reducing the Probability of Common-Mode Failure in the Fault Tolerant Parallel Processor*, AIAA/IEEE 12th Digital Avionics Systems Conference, October 1993, pp. 221 – 230.
[LalaJ 94]	Jaynarayan H. Lala and Richard E. Harper, *Architectural Principles for Safety-Critical Real-Time Applications*, Proceedings of the IEEE, Vol. 82, No. 1, January 1994, pp. 25 – 40.
[LalaP 91]	P. K. Lala, *On Self-Checking Software Design*, IEEE Proceedings of the Southeastcon, Vol. 1, 1991, pp. 331 – 335.
[Laprie 87]	Jean-Claude Laprie, et al, *Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions*, Digest of Papers FTCS-17: The Seventeenth International Symposium on Fault-Tolerant Computing, July 1987, pp. 116 - 121.
[Laprie 90]	Jean-Claude Laprie, et al, *Definition and Analysis of Hardware- and Software-Fault-Tolerance Architectures*, IEEE Computer, July 1990, pp. 39 - 51.
[Laprie 92]	J.C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, International Federation for Information Processing (IFIP), WG10.4 Dependable Computing and Fault Tolerance, Springer-Verlag, 1992.
[Laprie 94]	J.C. Laprie, *How Much is Safety Worth?*, Proceedings of the IFIP 13th World Computer Congress, Vol. III, n. A-53, 1994, pp. 251 – 253.
[Laprie 95]	J.C. Laprie, et al, *Architectural Issues in Software Fault Tolerance*, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 47 – 80.
[LeeI 93A]	Inhwan Lee, et al, *Measurement-Based Evaluation of Operating System Fault Tolerance*, IEEE Transactions on Reliability, Vol. 42, No. 2, June 1993, pp. 238 – 249.
[LeeI 93B]	Inhwan Lee and Ravishankar K. Iyer, *Faults, Symptoms, and Software Fault Tolerance in Tandem GUARDIAN90 Operating System*, Digest of Papers: The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 22 – 24, 1993, pp. 20 – 29.

[LeeI 95]	Inhwan Lee and Ravishankar K. Iyer, *Software Dependability in the Tandem GUARDIAN System*, IEEE Transactions on Software Engineering, Vol. 21, No. 5, May 1995, pp. 455 – 467.
[LeeI 96]	Inhwan Lee, et al, *Efficient Service of Rediscovered Software Problems*, Proceedings of the Twenty-Sixth International Symposium on Fault Tolerant Computing, Sendai, Japan, June 25 – 27, 1996, pp. 348 – 352.
[LeeP 83]	P.A. Lee, *Structuring Software Systems for Fault Tolerance*, AIAA Paper 83-2322, 1983.
[LeeY 98]	Yann-Hang Lee, et al, *An Integrated Scheduling Mechanism for Fault-Tolerant Modular Avionics Systems*, Proceedings of the 1998 IEEE Aerospace Applications Conference, Vol. 4, 1998, pp. 21 – 29.
[Levendel 89]	Y. Levendel, D*efects and Reliability Analysis of Large Software Systems: Field Experience*, Digest of Papers: The Nineteenth International Symposium on Fault-Tolerant Computing (FTCS-19), 1989, pp. 238 – 244.
[Leveson 83]	Nancy G. Leveson, *Software Fault Tolerance: The Case for Forward Recovery*, AIAA Paper 83-2327, 1983.
[Leveson 86]	Nancy G. Leveson, S*oftware Safety: Why, What, and How*, ACM Computing Surveys, Vol. 18, No. 2, June 1986, pp. 125 – 163.
[Leveson 95]	Nancy G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
[Leung 97]	Yiu-Wing Leung, *Processor Assignment and Execution Sequence for Multiversion Software*, IEEE Transactions on Computers, Vol. 46, No. 12, December 1997, pp. 1371 – 1377.
[Littlewood 89]	Bev Littlewood and Douglas R. Miller, *Conceptual Modeling of Coincident Failures in Multiversion Software*, IEEE transactions on Software Engineering, Vol. 15, No. 12, December 1989, pp. 1596 - 1614.
[Littlewood 93]	Bev Littlewood and Lorenzo Stringini, *Validation of Ultrahigh Dependability for Software-Based Systems*, Communications of the ACM, Vol. 38, No. 11, November 1993, pp. 69 – 80.
[Littlewood 96]	Bev Littlewood, *The Impact of Diversity Upon Common Mode Failures*, Reliability Engineering and System Safety, Vol. 51, No. 1, 1996, pp. 101 - 113.
[Lorczak 89]	Paul R. Lorczack, et al, *A Theoretical Investigation of Generalized Voters for Redundant Systems*, Digest of Papers FTCS-19: The Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 444 - 451.
[Lyu 92]	Michael R. Lyu, et al, *Software Diversity Metrics and Measurements*, Proceedings IEEE COMPSAC 1992, September 1992, pp. 69 – 78.
[Lyu 95]	Michael R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.
[Lyu 99]	Michael R. Lyu and Veena B. Mendiratta, *Software Fault Tolerance in a Clustered Architecture: Techniques and Reliability Modeling*, Proceedings of

the 1999 IEEE Aerospace Conference, Vol. 5, 1999, pp. 141 – 150.

[Mahmood 84] Aamer Mahmood, et al, *Executable Assertions and Flight Software*, Proceedings of the 6th AIAA/IEEE Digital Avionics System Conference, 1984, pp. 346 – 351.

[Mancini 89] L. V. Mancini and S. K. Shrivastava, *Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality*, Digest of Papers FTCS-19: The Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 454 - 461.

[Maxiom 98] Roy A. Maxion and Robert T. Olszewski, *Improving Software Robustness with Dependability Cases*, Digest of Papers: Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23 – 25, 1998, pp. 346 – 355.

[Mazza 96] C. Mazza, et al, *Software Engineering Guides*, Prentice Hall, 1996.

[McAllister 90] David McAllister, et al, *Reliability of Voting in Fault-Tolerant Software Systems for Small Output-Spaces*, IEEE Transactions on Reliability, Vol. 39, No. 5, December 1990, pp. 524 – 534.

[McElvany 88] Michelle C. McElvany, *Guaranteeing Deadlines in MAFT*, Proceedings of the Real-Time Systems Symposium, Vol. 35, No. 6, 1988, pp. 130 – 139.

[McKinzie 96] Gordon McKinzie, *Summing Up the 777's First Year: Is This a Great Airplane, or What?*, Airliner, July – September 1996, pp. 22 – 25.

[Melliar 83] Peter Michael Melliar-Smith, *Development of Software Fault-Tolerance Techniques*, NASA Contractor Report 172122, March 1983.

[Metze 81] Gernot Metze and Ali Mili, *Self-Checking Programs: An Axiomatisation of Program Validation by Executable Assertions*, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-Tolerant Computing, June 24 – 26, 1981, pp. 118 - 120.

[Migneault 83] Gerard E. Migneault, *On Requirements for Software Fault Tolerance for Flight Controls*, AIAA Paper 83-2321, 1983.

[Moser 95] L.E. Moser, et al, *The Totem System*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, June 1995, pp. 61 – 66.

[Mukherjee 97] Fault Tolerant System Design Paradigm (source: [Suri 95]) [Mullender 90]
Sape J. Mullender, et al, *Amoeba: A Distributed Operating System for the 1990s*, Computer, Vol. 23, No. 5, May 1990, pp. 44 – 53.

[Nelson 90] Victor P. Nelson, *Fault-Tolerant Computing: Fundamental Concepts*, IEEE Computer, July 1990, pp. 19 – 25.

[Nicola 95] Victor F. Nicola, *Checkpointing and the Modeling of Program Execution Time*, in Software Fault Tolerance, Michael R. Lyu, Ed, Wiley, 1995, pp. 167 – 188.

[Parhami 96] Behrooz Parhami, *Design of Reliable Software via General Combination of N-Version Programming and Acceptance Testing*, Proceedings of the 1996 17th International Symposium on Software Reliability Engineering (ISREE'96), 1996, pp. 104 – 109.

[Pham 92] Hoang Pham, editor, *Fault-Tolerant Software Systems: Techniques and Applications*, IEEE Computer Society Press, 1992.

[Parnas 90] David L.Parnas, A. John van Schouwen, and Shu Po Kwan, *Evaluation of Safety-Critical Software*, Communications of the ACM, Vol. 33, No. 6, June 1990, pp. 636 – 648.

[Potter 86] James E. Potter and M. C. Suman, *Extension of the Midvalue Selection Technique for Redundancy Management of Inertial Sensors*, Journal of Guidance, Control, and Dynamics, Vol. 9, No. 1, January – February 1986, pp. 37 – 44.

[Pradhan 92] Dhiraj K. Pradhan and Nitin H. Vaidya, *Roll-Forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares*, The 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, July 6-7, 1992, pp. 166 - 174.

[Pradhan 96] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, Inc., 1996.

[Pressman 97] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, The McGraw-Hill Companies, Inc., 1997

[Prowell 99] Stacy J. Prowell, et al, *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999.

[Purtilo 89] James M. Purtilo and Pankaj Jalote, *A System for Supporting Multi-Language Versions for Software Fault Tolerance*, Digest of Papers: The Nineteenth International Symposium on Fault-Tolerant Computing (FTCS-19), 1989, pp. 268 – 274.

[Rabéjac 96] Christophe Rabéjac, et al, *Executable Assertions and Timed Traces for On-Line Software Error Detection*, Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing, Sendai, Japan, June 25 – 27, 1996, pp. 138 –147.

[Ramirez 99] John C. Ramirez and Rami G. Melhem, *Reducing Message Overhead in TMR Systems*, Proceedings of the 1999 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, pp. 45 – 54.

[Randell 75] Brian Randell, *System Structure for Software Fault Tolerance*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 220 – 232.

[Randell 95A] Brian Randell and Jie Xu, *The Evolution of the Recovery Block Concept*, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 1 – 21.

[Randell 95B] Brian Randell, et al, editors, *Predictably Dependable Computing Systems*, Springer, 1995.

[Russinovich 93] Mark Russinovich, et al, *Application Transparent Fault Management in Fault Tolerant Mach*, Digest of Papers: The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 22 – 24, 1993, pp. 10 – 19.

[Russinovich 94] Mark Russinovich, et al, *Application Transparent Fault Managemet in Fault Tolerant Mach*, in Foundations of Dependable Computing System Implementation, Gary M. Koob and Clifford G. Lau, editors, Kluwer Academic Publishers, 1994, pp. 215 – 241.

[Russinovich 95] Mark Russinovich and Zary Segall, *Fault-Tolerance for Off-The-Shelf Applications and Hardware*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 27 – 30, 1995, pp. 67 – 71.

[Saglietti 86] F. Saglietti and W. Ehrenberger, *Software Diversity – Some Considerations About Its Benefits and Its Limitations*, Proceedings of the Fifth International Federation of Automatic Control (IFAC) Workshop, Safety of Computer Control Systems, October 14 – 17, 1986, pp. 27 - 34.

[Saglietti 90A] F. Saglietti, *Software Diversity Metrics: Quantifying Dissimilarity in the Input Partition*, Software Engineering Journal, January 1990, pp. 59 – 63.

[Saglietti 90B] F. Saglietti, *Strategies for the Achievement and Assessment of Software Fault-Tolerance*, IFAC 1990 World Congress, Automatic Control. Vol. IV, IFAC Symposia Series, Number 4, 1991, pp. 303 - 308.

[Saglietti 91] Francesca Saglietti, *The Impact of Voter Granularity in Fault-Tolerant Software on System Reliability and Availability*, in Software Fault Tolerance: Achievement and Assessment Strategies, M. Kersken and F. Saglietti, editors, Springer-Verlag, 1991.

[Salles 99] Frédéric Salles, et al, *MetaKernels and Fault Containment Wrappers*, Digest of Papers: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, June 15 – 18, 1999, pp. 22 – 29.

[Schach 96] Stephen R. Schach, *Testing: Principles and Practice*, ACM Computing Surveys, Vol. 28, No. 1, March 1996, pp. 277 – 279.

[Scott 87] R. Keith Scott, James W. Gault, and David F. McAllister, *Fault-Tolerant Software Reliability Modeling*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 582 – 592.

[Scott 96] R. Keith Scott, and David F. McAllister, *Cost Modeling of N-Version Fault-Tolerant Software Systems for Large N*, IEEE Transactions on Reliability, Vol. 45, No. 2, June 1996, pp. 297 – 302.

[Shimeall 91] Timothy J. Shimeall and Nancy G. Leveson, *An Empirical Comparison of Software Fault Tolerance and Fault Elimination*, IEEE Transactions on Software Engineering, Vol. 17, No. 2, February 1991, pp. 173 - 182.

[Shokri 96A] Eltefaat H. Shokri, et al, *Development of Software Fault-Tolerant Applications with ADA95 Object-Oriented Support*, Proceedings of the 1996 IEEE National Aerospace and Electronics Conference, Vol. 2, 1996, pp. 519 – 526.

[Shokri 96B] Eltefaat H. Shokri and Kam S. Tso, *Ada95 Object-Oriented and Real-Time Support for Development of Software Fault Tolerance Reusable Components*, Proceedings of the 1996 2nd Workshop on Object-Oriented Real-Time Dependable Systems, 1996, pp. 93 – 100.

[Shrivastava 91] S.K. Shrivastava and A. Waterworth, *Using Objects and Actions to Provide Fault Tolerance in Distributed, Real-Time Applications*, Proceedings of the Twelfth Real-Time Systems Symposium, December 1991, pp. 276 – 285.

[Simcox 88] L. N. Simcox, *Software Fault Tolerance*, Royal Signal and Radar Establishment, Memorandum 4237, 90N12575, June 1988.

[Simon 90] D. Simon, et al, *A Software Fault Tolerance Experiment for Space Applications*, Digest of Papers Fault Tolerant Computing: 20th International Symposium, University of North Carolina at Chapel Hill, June 26 – 28, 1990, pp. 28 – 35.

[Sims 97] J. Terry Sims, *Redundancy Management Software Services for Seawolf Ship Control System*, Digest of Papers: Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, June 24 – 27, 1997, pp. 390 – 394.

[Spitzer 86] Cary R. Spitzer, *All-Digital Jets are Taking Off*, IEEE Spectrum, September 1986, pp. 51 – 56.

[Strom 88] Robert E. Strom, et al, *Towards Self-Recovering Operating Systems*, Proceedings of the International Conference on Parallel Processing and Applications, September 1987, pp. 475 – 483.

[Subramanian 89] C. Subramanian and D.K. Subramanian, *Performance Analysis of Voting Strategies for a Fly-By-Wire System of a Fighter Aircraft*, IEEE Transactions on Automatic Control, Vol. 34, No. 9, September 1989, pp. 1018 – 1021.

[Sullivan 90] Gregory F. Sullivan and Gerald M. Masson, *Using Certification Trails to Achieve Software Fault Tolerance*, Digest of Papers FTCS-20: The Twentieth International Symposium on Fault-Tolerant Computing, June 26 – 28, 1990, pp. 423 - 431.

[Sullivan 95] Gregory F. Sullivan, et al, *Certification of Computational Results*, IEEE Transactions on Computers, Vol. 44, No. 7, July 1995, pp. 833 – 847.

[Sundresh 98] Tippure S. Sundresh, *Software Hardening – Unifying Software Reliability Strategies*, IEEE International Conference on Systems, Man, and Cybernetics, San Diego, CA, October 11 – 14, 1998, pp. 4710 - 4715.

[Suri 95] N. Suri, et al, *Advances in Ultra-dependable Distributed Systems*, IEEE Computer Society Press, 1995.

[Takano 91] Tadashi Takano, et al, *Longlife Dependable Computers for Spacecraft*, In Dependable Computing for Critical Applications, Vol. 4, A. Avizienis, J.C. Laprie, editors, 1991, pp. 153 – 173.

[TaylorD 80A] David J. Taylor, et al, *Redundancy in Data Structures: Improving Software Fault Tolerance*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 585 - 594.

[TaylorD 80B] David J. Taylor, et al, *Redundancy in Data Structures: Some Theoretical*

Results, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 595 - 602.

[TaylorR 90] R. C Taylor, et al, *A Flexible Fault Tolerant Processor for Launch Vehicle Avionics Systems*, 9th Digital Avionics Systems Conference, 1990, pp. 147 – 152.

[Tong 88] Zhijun Tong and Richard Y. Kain, *Vote Assignments in Weighted Voting Mechanisms*, Proceedings of the Seventh Symposium on Reliable Distributed Systems, October 1988, pp. 138 – 143.

[Traverse 91] Pascal Traverse, *Dependability of Digital Computers on Board Airplanes*, Dependable Computing for Critical Applications, Volume 4, A. Avizienis, J.C. Laprie, editors, 1991, pp. 134 – 152.

[Tso 86] Kam Sing Tso, *Error Recovery in Multi-Version Software*, Proceeding of the Fifth IFAC Workshop, Safety of Computer Control Systems, October 1986, pp. 35 - 41.

[Tso 87] Kam Sing Tso and Algirdas Avizienis, *Community Error Recovery in N-Version Software: A Design Study with Experimentation*, Digest of Papers FTCS-17: The Seventeenth International Symposium on Fault-Tolerant Computing, July 6 – 8, 1987, pp. 127 - 133.

[Tso 95] K. S. Tso, et al, *A Reuse Framework for Software Fault Tolerance*, AIAA Paper 95-1012, 1995.

[Vaidya 93] N. H. Vaidya, et al, *Trade-Offs in Developing Fault Tolerant Software*, IEE Proceedings –E, Vol. 140, No. 6, November 1993, pp. 320 - 326.

[Voas 94] Jeffrey M. Voas and Keith W. Miller, *Dynamic Testability Analysis for Assessing Fault Tolerance*, High Integrity Systems Journal, Vol. 1, No. 2, 1994, pp. 171 – 178.

[Voas 98A] Jeffrey M. Voas, *Certifying Off-the-Shelf Software Components*, IEEE Computer, Vol. 31, June 1998, pp. 53 – 59.

[Voas 98B] Jeffrey M. Voas and Gary McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons, Inc., 1998.

[Vrsalovic 90] D. Vrsalovic, et al, *Is it Possible to Quantify the Fault Tolerance of Distributed/Parallel Computer Systems*, Digest of Papers: 35th IEEE Computer Society International Conference – COMPCON Spring '90, 1990, pp. 219 – 225.

[Walter 90] Chris J. Walter, *Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault Tolerance*, IEEE Transactions on Reliability, Vol. 39, No. 4, October 1990, pp. 492 – 49.

[Watanabe 92] Aki Watanabe, et al, *The Multi-Layered Design Diversity Architecture: Application of the Design Diversity Approach to Multiple System Layers*, Proceedings of the Ninth TRON Project Symposium, December 1992, pp. 116 – 121.

[Watanabe 94] Aki Watanabe and Ken Sakamura, *MLDD (Multi-Layered Design Diversity)*

Architecture for Achieving High Design Fault Tolerance Capabilities, Dependable Computing – EDCC-1, Klaus Echtle, Dieter Hammer and David Powell, editors, First European Dependable Computing Conference, October 1994, pp. 336 – 349.

[Watanabe 95] Aki Watanabe and Ken Sakamura, *Design Fault Tolerance in Operating Systems Based on a Standardization Project*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 27 – 30, 1995, pp. 372 – 380.

[Webber 91] Steve Webber and John Beirne, *The Stratus Architecture*, Digest of Papers Fault-Tolerant Computing: The Twenty-First International Symposium, Montreal, Canada, June 25 – 27, 1991, pp. 79 – 85.

[Weinstock 97] Charles B. Weinstock and David P. Gluch, *A Perspective on the State of Research in Fault-Tolerant Systems*, Software Engineering Institute, Special Report CMU/SEI-97-SR-008, June 1997.

[Wicker 95] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, 1995.

[Williams 88] Ronald D. Williams, et al, *An Operating System for a Fault-Tolerant Multiprocessor Controller*, IEEE Micro, Vol. 8, No. 4, August 1988, pp. 18 – 29.

[Wu 91] Jie Wu, *Software Fault Tolerance Using Hierarchical N-Version Programming*, IEEE Proceedings of the SOUTHEASTCON '91, 1991, pp. 124 – 128.

[Wu 94] J. Wu, et al, *A Uniform Approach to Software and Hardware Fault Tolerance*, Journal of Systems and Software, Vol. 26, No. 2, August 1994, pp. 117 – 127.

[Xu 95A] Jie Xu, et al, *Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 27 – 30, 1995, pp. 499 – 508.

[Xu 95B] J. Xu, et al, *Toward an Object-Oriented Approach to Software Fault Tolerance*, Proceedings of the 1995 Conference on Fault Tolerant Parallel and Distributed Systems, 1995, pp. 226 – 233.

[Xu 97] Jie Xu and Brian Randell, *Software Fault Tolerance: t/(n-1)-Variant Programming*, IEEE Transactions on Reliability, Vol. 46, No. 1, March 1997, pp. 60 - 68.

[Xu 99] J. Xu, et al, *Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions*, Digest of Papers: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, June 1999, pp. 68 - 75.

[Yau 96] Stephen S. Yau, et al, *Object-Oriented Software Development with Fault Tolerance for Distributed Real-Time Systems*, Proceedings of the 1996 2nd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 1996, pp. 160 – 167.

[Yeh 96] Y.C. Yeh, *Triple-Triple Redundant 777 Primary Flight Computer*, Proceedings of the 1996 IEEE Aerospace Applications Conference, Vol. 1, 1996, pp. 293 – 307.

[Yen 96] I-Ling Yen, *Specialized N-Modular Redundant Processors in Large-Scale Distributed Systems*, Proceedings of the 1996 15th Symposium on Reliable Distributed Systems, Ontario, Canada, pp. 12 – 21.

[Yen 97] I-Ling Yen, *An Object-Oriented Fault-Tolerance Framework Based on Specialization Techniques*, Proceedings of the 1997 3rd International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'97), 1997, pp. 291 – 297.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** October 2000	**3. REPORT TYPE AND DATES COVERED** Technical Memorandum

4. TITLE AND SUBTITLE Software Fault Tolerance: A Tutorial	**5. FUNDING NUMBERS** WU 522-61-21-03
6. AUTHOR(S) Wilfredo Torres-Pomales	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-18034
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA/TM-2000-210616

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 61 Distribution: Nonstandard Availability: NASA CASI (301) 621-0390	**12b. DISTRIBUTION CODE**

13. ABSTRACT (Maximum 200 words)

Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems. The root cause of software design errors is the complexity of the systems. Compounding the problems in building correct software is the difficulty in assessing the correctness of software for highly complex systems. After a brief overview of the software development processes, we note how hard-to-detect design faults are likely to be introduced during development and how software faults tend to be state-dependent and activated by particular input sequences. Although component reliability is an important quality measure for system level analysis, software reliability is hard to characterize and the use of post-verification reliability estimates remains a controversial issue. For some applications software safety is more important than reliability, and fault tolerance techniques used in those applications are aimed at preventing catastrophes. Single version software fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling, and others. Multiversion techniques are based on the assumption that software built differently should fail differently and thus, if one of the redundant versions fails, it is expected that at least one of the other versions will provide an acceptable output. Recovery blocks, N-version programming, and other multiversion techniques are reviewed.

14. SUBJECT TERMS Software fault tolerance; Reliablity; Safety; Software engineering	**15. NUMBER OF PAGES** 66
	16. PRICE CODE A04

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL