NASA CR-172,618

NASA-CR-172618
19850027329

NASA Contractor Report 172618

STUDY OF FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DYNAMIC SYSTEMS

Alper K. Caglayan and Greg L. Zacharias

Charles River Analytics Inc.
Cambridge, MA 02138

Contract NAS1-17705 Phase I

September 1985

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NF00738

OCT 7 1985

RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

NASA Contractor Report 172618

STUDY OF FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DYNAMIC SYSTEMS

Alper K. Caglayan and Greg L. Zacharias

Charles River Analytics Inc.
Cambridge, MA 02138

Contract NAS1-17705 Phase I

September 1985

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

N85-35642 #

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1 Study Objectives and Approach 1
1.2 Report Outline 4

2. FAULT TOLERANT SYSTEMS OVERVIEW 6

2.1 Software Fault Tolerance 6
2.2 Software Implemented Hardware Fault Tolerance 10
2.2.1 Software Implemented Hardware Fault Tolerance Techniques 13
2.2.2 Failure Recovery in Software Implemented Hardware Fault Tolerance 20
2.3 Software Implemented Hardware vs. Software Fault Tolerance 23

3. SYSTEM MODELS FOR SOFTWARE 25

3.1 System Models for Software Modules 25
3.2 Static and Dynamic Software System Models 29

4. SYSTEMS-BASED SOFTWARE FAULT DETECTION TECHNIQUES 32

4.1 Static Consistency Tests 34
4.2 Dynamic Consistency 38
4.3 Innovations Signature Analysis 43

5. SYSTEMS-BASED SOFTWARE ERROR RECOVERY TECHNIQUES 47

5.1 Systems-Based Software Error Recovery Procedures 50
5.2 Systems-Based Recovery Block Initialization 52
5.3 ARMA vs. State Space Models for Recovery Blocks 57

6. FAULT TOLERANT FLIGHT SOFTWARE 59

6.1 Stability Issues in Fault Tolerant Flight Software 59
6.1.1 Software Module Stability 59
6.1.2 System Stability with Fault Tolerant Software 65
6.2 Preservation of Functional Performance 68
6.3 Generic Flight Software Blocks for Software Fault Tolerance 70
6.4 N-Version vs. Recovery Blocks in Flight Software 72
6.5 Performance Improvement with Fault Tolerant Software 78
6.6 Adaptability of Fault Tolerant Software 83

7. CONCLUSIONS AND RECOMMENDATIONS 85

Appendix A

References

LIST OF FIGURES

		Page
Figure 2.1:	Software Implemented Hardware Tolerant System Structure	11
Figure 2.2:	Causal Dynamic System Model	14
Figure 3.1:	Software Fault Tolerant System Structure	26
Figure 4.1:	Software Fault Signature Analyzer	44
Figure 5.1:	Recovery Block State Initialization Problem	53
Figure 6.1:	Version Instability in an N-Version PID Controller	62
Figure 6.2:	Example of Closed-Loop Instability with Fault Tolerant Software	66
Figure 6.3:	Semioctahedran Sensor Array Geometry	79

LIST OF TABLES

Table 6.1:	N-version Programming Feasibility in Flight Software	76
Table 6.2:	Recovery Block Programming Feasibility in Flight Software	77

1. INTRODUCTION

Computing systems are crucial components of high reliability applications such as flight and space systems. For computing system hardware in these high reliability applications it has been necessary to introduce fault tolerance techniques since the use of fault prevention techniques alone during the design has not, in general, yielded the desired reliability performance. For computing hardware, the desired fault tolerance has been obtained by introducing hardware redundancy and/or software-implemented hardware fault tolerance techniques.

Since software is an essential component of computing systems, fault tolerance techniques, notably N-version programming and recovery block procedures, have been proposed to detect and compensate software design faults. Currently, carefully controlled experiments are being conducted in order to study the potential reliability improvement of fault tolerant software over conventional software.

1.1 Study Objectives and Approach

The major objective of our study is to investigate the relevance of systems-based software-implemented hardware fault tolerance techniques in fault tolerant software technology. Over the last decade, several system theoretic fault detection isola-

tion, and compensation (FDIC) techniques have been developed for tolerating the effects of faults in hardware (digital, analog, electromechanical, hydraulic, etc. components) based on finding an appropriate input/output functional model of the hardware block, and analyzing the consistency of the observed inputs and outputs for failures. The major aim of our study is to investigate the feasibility of using these systems-based FDIC techniques in developing fault tolerant software, and extending them, whenever possible, to the domain of software fault tolerance.

We have therefore undertaken a program to investigate, first, the functional modelling of software modules within the domain of high level application programming, in particular, for real-time process control problems. We show that systems-based failure detection, isolation, and compensation (FDIC) methods can be extended to develop software error detection techniques based on these system models for software modules. In particular, we demonstrate through the use of several examples that these system theoretic FDIC techniques can be used to develop static and dynamic consistency checks which are simpler to implement than acceptance tests based on software specifications. Our study also reveals that software error recovery can be integrated with software-implemented hardware fault tolerance within the framework of signature analysis.

We follow this by showing that systems-based failure compensation techniques can be generalized to the domain of software fault tolerance in developing software error recovery procedures. In particular, the systems approach yields forward error recovery procedures which do not depend on an exact assessment of the software damage. We also find a solution to the recovery block state initialization problem, for the case when the alternate block algorithm is represented by a linear dynamic system. Our study also reveals that an autoregressive moving average (ARMA) implementation of an algorithm is more advantageous to use in an alternate module than an equivalent state space implementation.

We also examine the feasibility of using software fault tolerance in flight software. In this and other real-time process control applications, we identify the major issues of concern: weak encapsulation due to memory and feedback, inexact voting and acceptance tests, closed-loop system and version stability, and preservation of functional performance. We illustrate system and version instabilities and functional performance degradation that may occur in N-version programming applications to flight software. In this study, we make a comparative analysis of N-version and recovery block techniques in the context of generic blocks in flight software. The most often cited advantage in using fault tolerant software is the potential software reliability gain. In this study, we give another reason for the introduction of software fault tolerance. In this regard, we

propose that software fault tolerance techniques offer the potential of raising functional performance to levels unattainable by conventional software methods.

In summary, our study shows that systems based failure detection, isolation and compensation methods can be extended to the domain of software fault tolerance by developing system models for software modules. In particular, we demonstrate that systems-approach can yield software error detection techniques and software error recovery procedures with advantages over those based solely on software specifications. Finally, we outline the potential problems that may arise due to an indiscriminate use of fault tolerant software techniques in developing flight software.

1.2 Report Outline

Chapter 2 contains an overview of the "software" and "software-implemented hardware" fault tolerance. While the review of software fault tolerance (Section 2.1) is brief, the software implemented hardware fault tolerance techniques are described, in detail: see Section 2.2.1 for a description of the techniques, and Section 2.2.2 for a discussion of the failure recovery algorithms. This chapter ends with Section 2.3 outlining the major differences between software and software-implemented hardware fault tolerance areas.

In Chapter 3, we discuss how to obtain system models for software modules (Section 3.1), and important attributes of

system models such as memory, feedback, and time invariance (Section 3.2).

Chapter 4 contains our generalization of software implemented hardware fault detection techniques to the domain of software fault tolerance. In particular, static consistency, dynamic consistency, and signature analysis based software error detection techniques are discussed in Sections 4.1-3.

In Chapter 5, we outline the major issues of concern in software fault recovery, and present systems based solutions to forward error recovery (Section 5.1), and recovery block initialization (Sections 5.2-3) problems.

In Chapter 6, we discuss the issues involved in applying software fault tolerance techniques to flight software. In particular, stability issues in fault tolerant flight software (Section 6.1), preservation of functional performance (Section 6.2), generic flight software blocks for software fault tolerance (Section 6.3), a comparative evaluation of N-version and recovery blocks (Section 6.4), the potential for performance improvement with fault tolerant flight software (Section 6.5), and adaptability of software fault tolerance to existing conventional flight software (Section 6.6) are discussed.

Conclusions and recommendations are presented in Chapter 7.

2. FAULT TOLERANT SYSTEMS OVERVIEW

In this chapter, we present a brief overview of fault tolerance relevant to our study. Our discussion is confined to software and software-implemented system-theoretic hardware fault tolerance as defined below.

Software fault tolerance is the set of techniques necessary to enable computing systems to tolerate faults in the design and implementation of the software itself. Hence, our usage of the expression "software fault tolerance" is consistent with use in Anderson and Lee (p.250) in [1]. Moreover, we use the expression "fault tolerant software" to mean software constructed by using software fault tolerance techniques. The other possible interpretation of software fault tolerance, that is, techniques for designing software to tolerate the effects of faults in the underlying hardware, will be termed "software-implemented hardware fault tolerance".

2.1 Software Fault Tolerance

There are two main methods which have been proposed for fault tolerant software development. These are

- o recovery blocks
- o N-version programming

In the recovery block method, [2]-[3] there are two or more versions -- primary and alternates -- of a given program block (module). A backup alternate program block is executed when the

corresponding primary program block is deemed faulty through an acceptance test. For instance, the primary module may be an efficient but incompletely validated program, whereas the alternate block may be a less efficient but fairly well tested version of the same program. The acceptance test is performed by using the variables accessible to other modules, rather than variables which are local to that program module. We view this recovery block technique as the software analog of the classical stand-by redundancy approach used for hardware fault tolerance in which the stand-by equipment is of a design similar to the primary, but obtained from a different manufacturer.

In N-version programming [4], two or more functionally equivalent programs are independently generated from the same initial specifications. The independence of programs is assured through the use of N different non-interacting software design groups assigned to the programming effort. Dissimilar algorithms and even different languages can be used to extend the independence of each version. Thus, N-version programming yields software which is markedly dissimilar to systems in which two or more identical replications of a program are executed concurrently in physically distinct hardware, such as in the Space Shuttle Computer System [5], SIFT [6], and FTMP [7]. In N-version programming, N programs are executed concurrently and checked against each other by comparing a certain subset of the generated program state variables. Hence, N-version programming is an

adaptation of the fault tolerant duplex, triplex, etc. hardware redundancy approach to software generation, in which each redundant hardware component is an independently designed, but functionally equivalent instrument, obtained from different manufacturers.

It is also possible to use the recovery block method with N-version programming in a hybrid framework to exploit the inherent advantages of each approach [8].

As stated in [1], software fault tolerance principles can be discussed in terms of the following four phases:

- error detection
- damage assessment
- error recovery
- fault treatment

The first stage in providing software fault tolerance is to detect the effect of a software fault. In addition to using internal interface checks in a module, software faults may be detected by acceptance tests applied to recovery blocks, and voting checks applied to N-version programming. Acceptance tests are usually performed at the output of an individual software module to check the reasonableness of the results computed by each module. In contrast, voting checks compare the results across the N computed versions to identify software faults.

Damage assessment involves finding the extent of the detected fault's spread within the system. This comes about

since there is, in general, a time delay between the occurrence of a fault, and the detection of its effects on the tested outputs. This issue is especially important in applications of a probabilistic nature involving, for example, programs performing floating point arithmetic on noisy data, as contrasted with applications of a deterministic nature involving simple integer arithmetic or character strings manipulation.

Following the detection of a software fault and damage assessment, error recovery techniques are used to restore the erroneous system state back to an error-free-state so that normal system operation can continue. There are two such recovery mechanisms:

- o backward error recovery
- o forward error recovery

In backward error recovery, the computational state is reset to an earlier (presumably known to be error-free) state. Backward error recovery techniques are usually associated with recovery block methods which, together with the use of a recovery cache, provide a means of storing previous computational states. In forward error recovery, the current state is changed to compensate for the effects of the detected software fault. Forward error recovery techniques are useful in real-time applications where backward error recovery is generally not feasible.

Fault treatment in recovery blocks consists of using the alternate module for execution. This is usually done temporarily

so that the primary block is used on subsequent executions. In N-version programming, fault treatment consists of ignoring the computations of versions that are determined to be faulty. One serious problem in N-version programming, especially in blocks involving programs with floating point arithmetic and feedback, is the possibility of unselected versions diverging from one another, thus reducing system fault tolerance.

2.2 Software Implemented Hardware Fault Tolerance

Here, we give an overview of the software-implemented hardware fault tolerance techniques for general dynamic systems. The methods we discuss are drawn largely from control, estimation, and communication theories, as well as from the field of mathematical statistics. Fault tolerance for dynamic physical systems has been traditionally achieved through the use of hardware redundancy (stand-by, duplex, triplex, etc.) Over the last decade, however, quite a number of fault tolerance techniques [9]-[13] have been developed in which hardware redundancy has been replaced or augmented with analytic redundancy which uses a functional model of the physical system under consideration.

In most of the system-theoretic fault tolerance methods, an estimator based on the causal functional model of the physical dynamic system (depicted in Figure 2.1) is first constructed in software. This software module is then driven by the actual inputs and outputs of the physical system to recursively con-

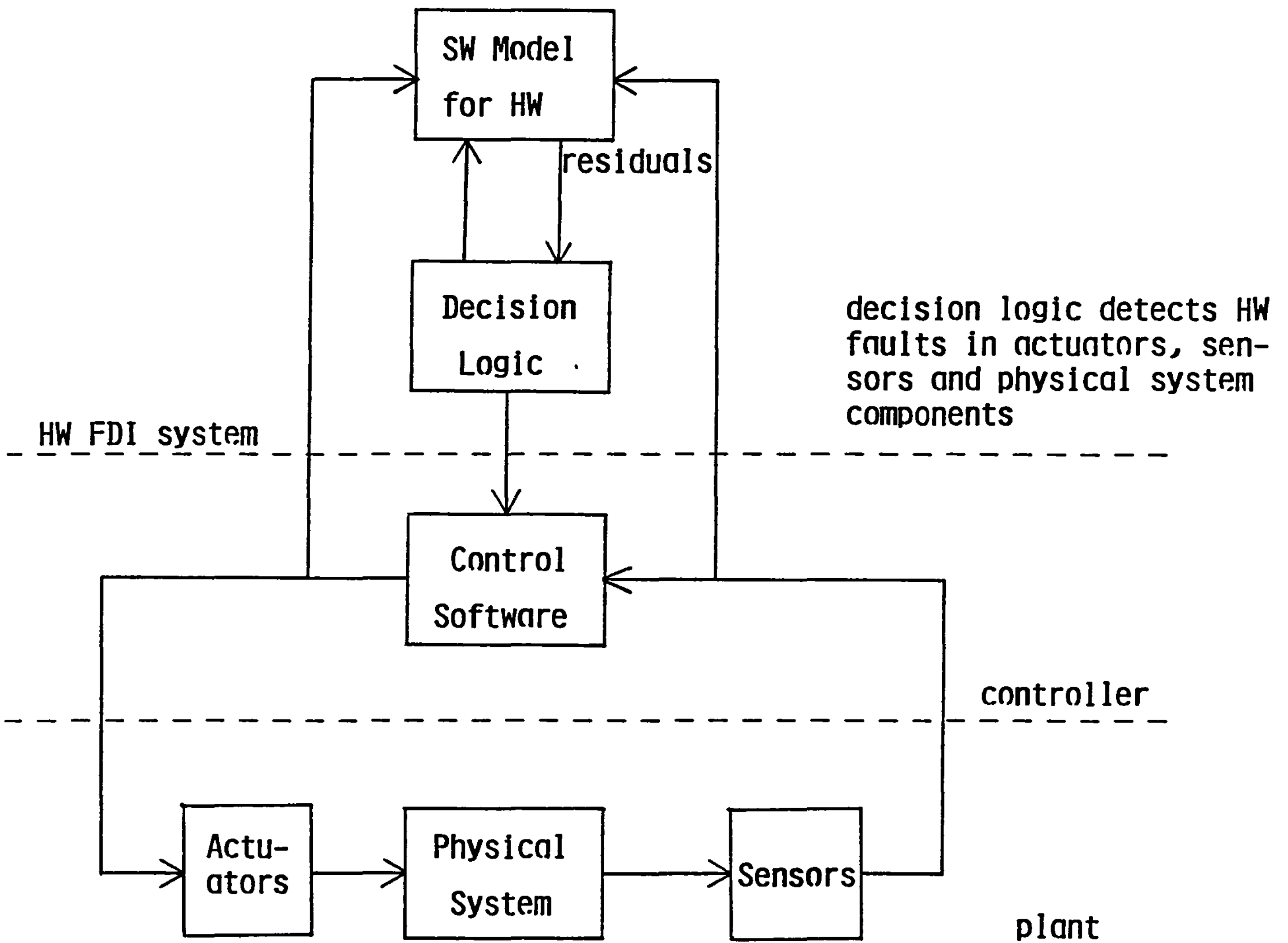

Figure 2.1: Software Implemented Hardware Fault Tolerant System Structure.

struct estimates for the physical system outputs. These model estimates are then differenced with the actual system outputs to form a residual sequence over time. Inconsistency (nonzero mean, deviation from theoretically computed statistics, etc.) in this residual sequence is then analyzed to detect and isolate faults in the inputs, outputs, and components of the physical system. The various system-theoretic methods differ mainly in the way the system model is constructed, and in the manner the output estimates are computed.

The systems approach to failure detection and isolation (FDI) problems in dynamic systems is based on using the analytic relationships between various sensor outputs, derived from a knowledge of the underlying system dynamics. Analytic redundancy can be either in the form of algebraic redundancy -- the instantaneous relationship between sensor outputs, or dynamic redundancy -- the relationship between the time histories of sensor outputs. The term "analytic redundancy" was coined in the early seventies to differentiate this technique from the traditional hardware redundancy approach in which the outputs of like sensors are compared for failure detection. Analytic redundancy comes about from the common estimation capability of various sensor groups. Sensor FDI algorithms make use of this inherent analytic redundancy by considering different sensor subsets. Hence, the analytic redundancy approach offers the capability of comparing dissimilar instrument outputs for failure detection and, thus,

allows the design of reliable systems with reduced hardware duplication.

Analytic redundancy research culminated in the development of aircraft sensor fault tolerant digital flight control systems, such as the USAF DIGITAC A-7 and the NASA/LRC F8-DFBW applications [14]-[15], engine sensor failure detection systems such as the NASA/LeRC F100 application [16], strap down navigation systems with skewed sensor arrays such as the NASA/LRC RSDIMU [17], and, more recently, sensor fault tolerant integrated flight control and navigation systems, such as the NASA/LRC TCV Research Aircraft application [18]-[20].

2.2.1 Software Implemented Hardware Fault Tolerance Techniques

General failure detection and isolation methods for dynamic systems can be divided into the following groups:

- Voting Methods
- Parity Techniques
- Failure Sensitive Filters
- Multiple Model Methods
- Innovations Signature Analysis

Voting Methods

Voting methods are comprised of mid-value select, and majority voting techniques. SIFT [6] is an example of a software implemented hardware fault tolerance voting technique developed for computing hardware.

Parity Techniques

Parity methods [21] encompass the standard voting techniques for systems with parallel hardware redundancy and their generalizations to systems with functional redundancy. Referring to Figure 2.2, which identifies the inputs, internal states, and

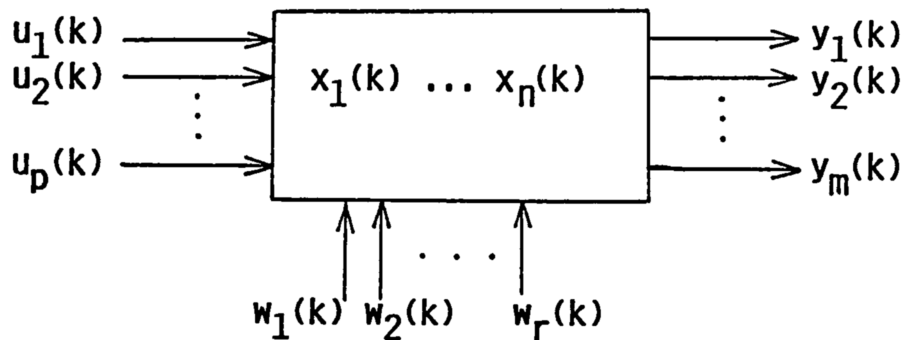

Figure 2.2: Causal Dynamic System Model

outputs of the software model in Figure 2.1, these techniques correspond to looking at only the output measurements, $\{y_1(.),...,y_m(.)\}$, to determine the failure modes. For instance, the standard voting techniques would require that each measurement, $y_i(.)$, has at least duplex redundancy for failure detection and triplex redundancy for failure isolation. Functional redundancy comes about when the measurements are related through the observed variables.

For instance, four or more non-coplanar instruments measuring a three dimensional state variable (such as acceleration) is an example of functional redundancy. In the parity approach, a least squares state estimator using all of the measurements is first constructed in software. Next, a predictor is constructed similarly in software to compute a prediction for each of the measurements on the basis of the estimate. Finally, the difference between the actual measurements and those predicted by the software model are analyzed for faults. Both parity techniques, using either parallel or functional redundancy, utilize the static analytic relationships (as opposed to dynamic), which exist between the sensed variables. For instance, parallel voting techniques assume that the observations measure precisely the same system output variable. Similarly, parity methods based on functional redundancy assume that the measurements are different transformations of precisely the same system variable. However, none of these techniques use the dynamic relationships which dictate how the measured variables evolve in time.

A recent generalization of the parity approach is proposed in [22], for general dynamic systems with a time-domain state-space description. This generalized parity approach uses the temporal analytic relationships which exist between the measured outputs and inputs in Figure 2.2. Therefore, this method is applicable to systems without parallel or functional redundancy.

<u>Failure Sensitive Filters</u>

The failure sensitive filter approach was developed by Beard in [23]. In this approach, a filter is first constructed in software using a time-domain state-space model of a linear time-invariant dynamic system. The gain parameters of this filter are chosen such that a particular failure mode results in a measurement residual sequence which remains fixed in a single direction or plane. In problems where the flexibility exists, the remaining gain parameters can be chosen to improve the convergence rate of the residual sequence to the desired direction.

While the failure sensitive approach has brought geometric insight to the failure detection problems, it has not found widespread use in applications due to its limitations to time-invariant systems. Similarly, jump process formulations [10] focus on steady state effects of a particular failure on the measurement residual sequence of a time-invariant filter. The failure diagnosis is achieved by implementing the recursive relationship for the a posteriori probability of the failure using the measurement residual. However, jump process for-

mulations suffer from the same basic limitations associated with failure sensitive filters.

Multiple Model Methods

Multiple model methods are based on constructing a different software model for each failure mode as done by Montgomery, Caglayan, and Price in [24], [25], and by Willsky, Deyst and Crawford in [26]. In this approach, for each failure mode, several software models are constructed, each modelling the effect of a particular postulated failure. For each such model, a measurement residual sequence is then generated by implementing the corresponding state estimator in software. For instance, referring to Figure 2.2, this approach would require the software implementation of a bank of state estimators, each of which would use a different m+p-1 measurement combination from the set $\{u,(.),...,u_p(.),y_1(.),...,y_m(.)\}$. Therefore, a bank of measurement residual sequences is available for analysis in determining the most likely failure mode. Statistical hypothesis testing procedures are then employed in reaching a failure decision. Although the multiple model method would, in general, yield the best failure detection performance for the widest class of failures, computational requirements of this brute force approach preclude its use in most practical applications.

Another class of failure detection techniques (which can also be classified under the multiple model method) would be those using filter assemblies [14] resulting from a different

grouping of monitored measurements than that employed in the multiple model method. These methods are also based on a time-domain state-space description of the system. In this approach, the measurement of a given output is predicted by using some or all of the other measurements except the one which is being estimated. Next, measurement residuals are formed by taking the difference between the actual measurements and the analytically constructed measurement predictions. Finally, standard voting techniques are employed to determine faults from appropriate truth tables.

Recent multiple model applications have been mostly in multitarget tracking problems [27] where the availability of large scale digital computers renders the bank of filters solution feasible. The relationship between the multiple model and likelihood ratio methods has been investigated by Caglayan in [31]. In this work, it is shown that the multiple model and GLR method solutions are equivalent for additive bias type sensor failure models.

Innovations Signature Analysis

The last group of failure detection methods involves the monitoring of the effects of a failure on the measurement residual sequence of a single filter corresponding to the normal operation of the system [28]-[31]. These methods use a time-domain state-space model of the unfailed system dynamics. In this method, a filter, based on the assumption of no failures, is

first implemented in software to estimate the system states. These estimates are then used to predict the measurements and to form the residual sequence. Statistical tests are then performed on the residuals of this filter to isolate failures. Prior statistical and structural knowledge of the effects (signatures) of such failures on the measurement residuals are then used to detect and isolate faults.

Innovations signature analysis techniques include performing statistical tests on measurement innovations as suggested by Mehra and Peschon in [32], and the generalized likelihood ratio (GLR) method which was originated by McAulay and Denlinger in [28] and formalized by Willsky and Jones in [29].

One of the developments in the signature analysis approach has been the modified GLR algorithm proposed by Basseville and Benveniste in [13]. In this approach, the GLR is constructed to test whether the failure level is greater than or less than an a priori fixed minimum amplitude. Another recent development has been the cumulative sum (CUSUM) type test proposed by Segen and Sanderson [33] for testing the change in statistical properties of a stochastic sequence. This test, which is related to Hinkley's mean shift testing results [13], is based on cumulative sums of squares of the innovations and has advantages in terms of robustness with respect to distributions after the time of failure.

Finally, recent work by Caglayan and Lancraft [18]-[20], has concentrated on extending the signature analysis methods to nonlinear dynamic systems. The sensor fault tolerant system developed in [18]-[20] can be viewed as a generalization of the GLR method to nonlinear systems. The major contribution of this effort has been the development of expressions for the linearized effects of bias type sensor failures on the measurement innovations in a nonlinear filtering framework. A second important contribution has been the compensation of "normal operating" sensor biases in the no-fail filter and investigation of the interaction between the normal operating bias estimates and the bias failure level estimates of the corresponding detectors.

2.2.2 Failure Recovery in Software-Implemented Hardware Fault Tolerance

Fault tolerant systems, in which analytic failure detection and isolation (FDI) techniques are used on-line to identify hardware failures, usually require some level of compensation to remove the accumulated effects of the detected failure on the system model.

In these software-implemented hardware fault tolerant systems, system failures must propagate through the software model of the physical plant (until a significant residual signature is generated) to get detected. Therefore, the software model must be reinitialized to remove the accumulated effects of the

detected failure on the model. In addition, the software model must be restructured after the isolation of a failure, to account for the loss of a system input or output, or a change in plant dynamics due to an internal component failure.

Failure recovery procedures for system-theoretic hardware fault tolerant systems can be grouped as follows [27]:

- o Reprocess Measurements
- o Reinitialize State Estimate and Uncertainty
- o Reset State Estimate and Increment Uncertainty
- o Increment Uncertainty
- o Probabilistic Weighting

We discuss these procedures in the following paragraphs.

Reprocess Measurements: If the exact time of failure can be estimated, and if the measurements from failure onset time are saved, then the software model can be restructured and then rerun with a measurement set containing only the healthy measurements. However, this approach is not feasible in most applications, since the exact time of failure is usually not estimated because of computational constraints. Moreover, even if a moving window of measurements were to be saved, it is possible that the reprocessing of the measurements could not be done in real time. This failure recovery method is analogous to the backward error recovery procedure in software fault tolerance.

Reinitialize State Estimate and Uncertainty: Here, the software model state covariance parameters are set to the values

originally specified by the initial conditions. The state estimate can possibly be reinitialized by following the procedure employed in selecting the plant state estimate initial conditions. Naturally, this approach would generate transients associated with the settling of the filter gains, in a manner similar to that encountered during system initialization.

Reset State Estimate and Increment Uncertainty: If the failure levels are estimated by the software model, then the state estimate can be reset following the detection and isolation of a failure. The uncertainty of the state estimate (due to uncertainty in failure time and/or level) can be increased by incrementing the associated covariance in a manner consistent with the specifics of the detected failure. However, in most practical applications, the failure onset time cannot be accurately estimated. Furthermore, in some applications, a sudden change in the state estimates would not be desirable due to the transient effects produced, for instance, by a control law using the state estimates. This failure recovery method is analogous to forward error recovery in software fault tolerance.

Increment Uncertainty: In applications where a sudden change in the estimates is not desired, or failure onset time cannot be accurately determined, the software model can be reinitialized by incrementing only the state estimation error covariance by an appropriate amount, following the isolation of a failure. In this manner, the state estimation error in the

software model can be gradually compensated. The appropriate covariance to be used is the conditional covariance of the no-fail filter, conditioned on the given observations under the decided failure mode [27]. The accumulated effects of the failure on the software model state are not immediately taken out with this approach; however, the additional uncertainty added to the state estimate, by incrementing the error covariance, gradually compensates for the error accumulation caused by the detected failure.

Probabilistic Weighting: Here, failure level estimates are incorporated into the software model state by weighting the a posteriori probabilities. Instead of the hard switching produced by a decision rule, this approach provides soft switching between failure modes.

2.3 Software-Implemented Hardware vs. Software Fault Tolerance

Systems-based fault tolerance methods have been used to reduce hardware redundancy in high reliability applications. By using these techniques, it has been possible to compare dissimilar instruments (e.g., an accelerometer with an airspeed indicator) as well. Since software implementation is indistinguishable from a hardware solution, from a functional input/output point of view, these model-based failure detection techniques are applicable to developing fault tolerant application software as well.

Although systems-based fault tolerance methods have been implemented in software, the issue of faults in the implemented software has been largely ignored in system-theoretic fault tolerance research. In essence, it has been assumed that the software associated with implementing the analytic redundancy has perfect reliability.

Another aspect of fault tolerant software that differentiates it from the system-theoretic fault tolerant systems work is the functional modelling aspect. In the system-theoretic approach, finding the appropriate reduced order dynamic model for the physical system, and handling of the various sensor noises is one of the most critical issues involved. In contrast, the functional models in fault tolerant software are better defined (via specifications) since they do not involve the modelling of physical processes. However, the noises associated with the inaccuracies in software functional models are usually due to truncations used in algorithms and finite precision arithmetic.

3. SYSTEM MODELS FOR SOFTWARE

To apply software-implemented hardware fault tolerance techniques to software fault tolerance, it is necessary to obtain an input/output functional system model of the software module. In this chapter, we discuss how to obtain such system models, and their relation to the software specification process. Examination of Figure 2.1 depicting a general software implemented hardware fault tolerant system structure suggests that if the "Software Model for Hardware" block is replaced by a system model of the control software, then systems-based FDIC techniques can be extended to software fault tolerance as well. Figure 3.1 shows a general block diagram for such a systems approach to software fault tolerance. Referring to this figure, the inputs and outputs of a monitored software module drive a system model for that software module. This system model generates a set of residuals which is the difference between the observed and predicted module behaviour. These residuals are, in turn, analyzed by a decision logic whose outputs are then used by the driver logic program to enable error recovery procedures. In the next section, we give a formal definition for system models of software modules.

3.1 System Models for Software Modules

A system model for a software module is a transformation relating the module's inputs to its outputs. In a functional

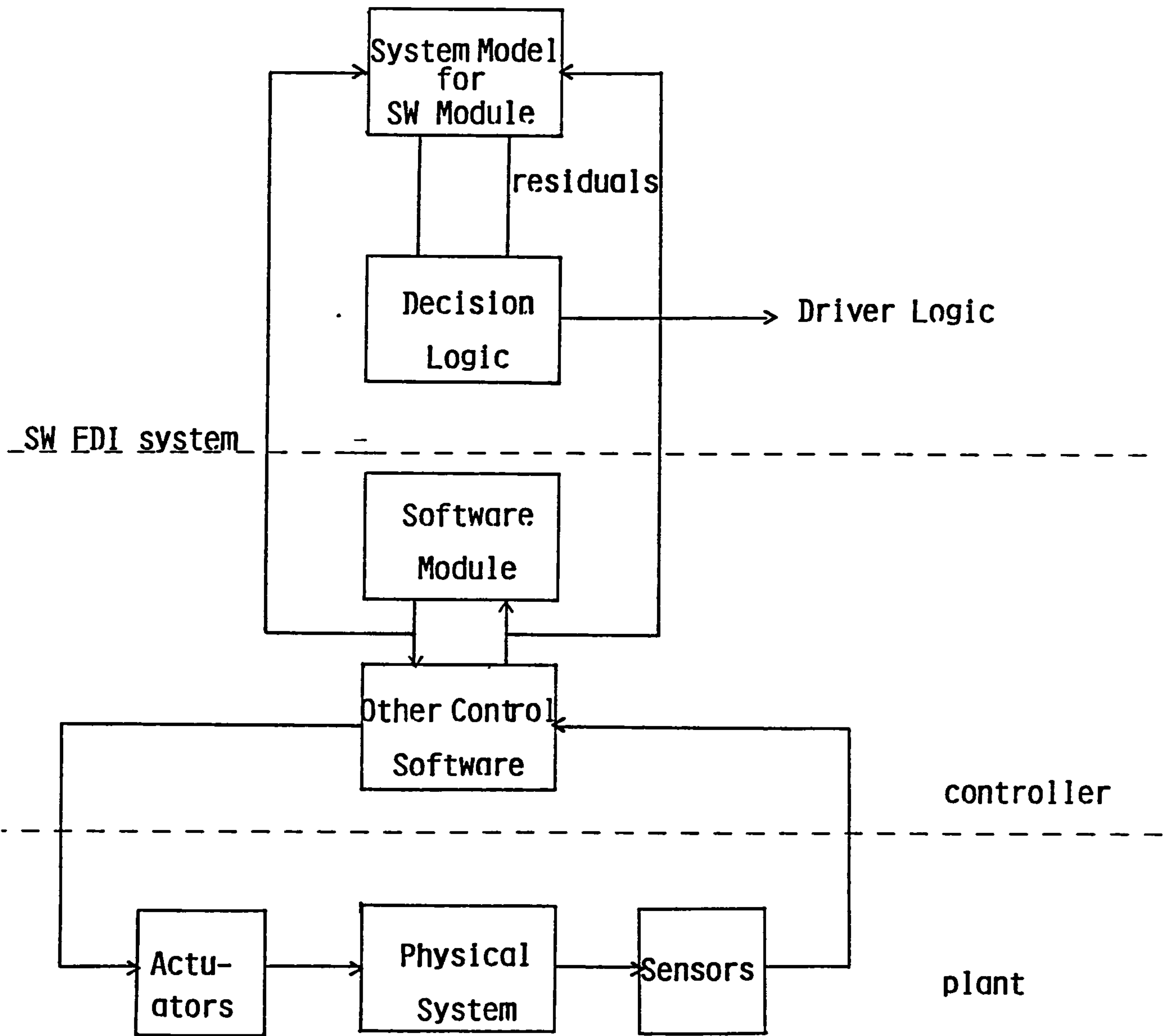

Figure 3.1: Software Fault Tolerant System Structure.

analysis framework (referring back to Figure 2.2), a transformation, T, is a single-valued function mapping each element in the input set into a single element in the output set. The set of inputs and outputs are called, respectively, the domain, D, and range, R, of T. Formally, we write

$$T:D \rightarrow R \quad \text{with } y = T(u) \qquad (3.1)$$

for all u in D and y in R. The domain and range of the transformation in Figure 2.2 may be considered to be the set of all R^p and R^m valued sequences respectively. An R^m valued sequence is, by definition, a transformation mapping the set of natural numbers into the m-dimensional Euclidean space.

Two important attributes of transformations are the properties of being "onto" and "one-to-one". Referring to Figure 2.2, if T maps D into all of R, i.e. for each r in R there exists a d in D such that r=T(d), then T is said to map D onto R. On the other hand, if the mapping T is such that distinct inputs are mapped to distinct images, T is said to be one-to-one. One immediate result is that T is invertible (i.e. the inverse of T is also a transformation) if and only if T is one-to-one and onto. For example, in Figure 2.2, the transformation is not onto the set of all R^m valued sequences since, by causality, the present output is a function of only past and present inputs. We shall say more about this subject in the section on static consistency checks (Section 4.1).

Ideally, a system model for a given software module should be computable from its specification. However, more often than not, the specification is not precise enough to completely define such a model. For example, a specification can be satisfied by implementing various algorithms with different accuracies, if the accuracy is not explicitly given in the specification. Consider, for instance, a software block for computing the inverse of real nonsingular square matrices. The specification for this module can be written as: find the transformation, f, mapping the set of all real nonsingular matrices with a fixed but arbitrary order onto itself such that for each input matrix A of order n:

$$B = f(A) \quad \text{with} \quad AB = BA = I \tag{3.2}$$

where I is the identity matrix of order n. (This transformation is an example of one-to-one and onto where the inverse transformation is identical to f).

Although the specification above is an acceptable one, it would not be sufficient to use as a system model. For instance, the desired accuracy of the implemented transformation is not mentioned in the specification. The constraint defined by (3.2) would not be exactly satisfied, because of the requirement to use finite precision arithmetic, and of the inherent inaccuracy of the algorithm used. For example, the implemented matrix inversion routine could be based on Gaussian elimination or Gram-Schmidt orthogonalization. For either of these algorithms, the implemented routine could use scaling and pivoting, or not. Each

of these decisions would affect the accuracy of the resulting computed inverse. Hence, an appropriate system model of the software inverse module could be of the form

$$B = f(A) \quad \text{with} \quad ||AB-I|| < s \tag{3.3}$$

where $||\cdot||$ is a specific matrix norm such as maximum row or column sum and s is a threshold which may be function of the input matrix elements, and the order of the input matrix. This issue of software module functional performance will be important in the design of acceptance tests and voting checks.

3.2 Static and Dynamic Software System Models

In system theory, it is important to differentiate between static and dynamic systems. Referring to Figure 2.2 again, and defining the p dimensional input vector $u(k)=[u_1(k), u_2(k)\ldots u_p(k)]$ and output vector $y(k)=[y_1(k), y_2(k)\ldots y_m(k)]$, a causal dynamic model in its most general form, will be given by

$$y(k)=f(u(1),u(2),\ldots,u(k),$$
$$y(1),y(2),\ldots,y(k-1),w(1),\ldots,w(k),k), \quad k=1,2.... \tag{3.4}$$

where the noise sequence, w(k), is a sequence of r-dimensional vector random variables with known statistics. This noise sequence w(k), in the software model could be used, for instance, to model the inaccuracy in the input sequence due to both finite precision representation and inaccuracies introduced by the algorithm used for generating the input. The noise sequence, w(k), could also be used to represent the errors introduced by

the algorithm implemented in the module itself, as well as the errors arising from its implementation in finite precision arithmetic.

Within the framework of this system model, we can consider a number of model attributes arising from the properties of the transformation, f, including:

- o memory
- o feedback
- o time-invariance

The model of (3.4) provides for <u>memory</u> since the current output can be a function of the previous inputs, $\{u(1),...,u(k-1)\}$, and outputs, $\{y(1),...,y(k-1)\}$. If the transformation, f, has no memory, then the system model is said to be static. For example, the matrix inversion example considered earlier would have a static system model. If the current input, u(k), is a function of the current and previous outputs, $\{y(1),y(2)..., y(k-1)\}$, the system model above is said to have <u>feedback</u>. This attribute is important in determining the stability properties of the implemented software in the presence of accumulated roundoff errors, for example, and is discussed later. Finally, the dependence of the transformation on the time index k, results in a time-varying system model. If there is no such dependence, the system model is said to be <u>time-invariant</u>.

The system model described by (3.4) is called an auto regressive moving average (ARMA) model. In system theoretic

applications, algorithms can also be implemented using an equivalent state space model. Referring to Figure 2.2 and denoting the internal n dimensional, state vector by $x(k)=[x_1(k)...x_n(k)]$, the equivalent state space model, if it exists, would be given by:

$$x(k) = f(x(k-1), u(k-1), w(k-1), k) \tag{3.5}$$

$$y(k) = h(x(k), u(k), w(k), k) \tag{3.6}$$

where the transformations f and h define the state space system model. For instance, for a time-invariant model, there would no dependence on k in either f or h.

We can formally write the transformations f and h:

$$f:R^n x R^p x R^r x N \rightarrow R^n \tag{3.7}$$

$$h:R^n x R^p x R^r x N \rightarrow R^m \tag{3.8}$$

where N is the set of natural numbers representing time. The selection of an ARMA state space model for the implementation of an algorithm in a software module is relevant in the recovery block state initialization problem which will be discussed in Chapter 5.

4. SYSTEMS-BASED SOFTWARE FAULT DETECTION TECHNIQUES

In this chapter, we outline how system-theoretic software-implemented hardware fault tolerance methods can be used to develop techniques for the detection of software faults. The results will mostly be applicable to the design of acceptance tests for recovery blocks, and voting checks for N-version programming. We will assume that a dynamic system model of the software module in the form detailed in the previous chapter has already been derived.

The complexity of this system model will, of course, determine to a great extent the complexity of the failure detection logic. For instance, in a redundancy management software module for a skewed sensor array, where software-implemented failure detection and isolation (FDI) logic is used to reduce hardware redundancy, it is mandatory that the additional software development and implementation cost be lower than the cost of the replaced sensor. Similarly, in fault tolerant software, it is critical to have the complexity of the failure detection software be substantially less than that of the software module being tested. This requirement is mainly due to the need to assure that the test module itself is free of software faults [1].

In system-theoretic software-implemented hardware fault tolerance, there are a number of methods for reducing the complexity of the failure detection logic; these include:

- reduced-order modelling
- choice of an FDI technique

In software implemented hardware fault tolerance, the order of the system model of the physical plant has a very significant impact on the complexity of the detection logic. For instance, a nonlinear, one hundred state, dynamic description (referring to Fig 2.2, n = 100) of an aircraft engine can be modelled by a fourth order linear dynamic model for the purpose of sensor failure detection design. Likewise,in fault tolerant software, a simple and appropriate description of the module algorithm may be quite sufficient for the purpose of designing the failure detection logic. For example, consider a software module performing a fourth-order Runge-Kutta integration of angular body rates to generate vehicle attitudes. The failure detection logic could be based on a simple rectangular integration model of the actual high-order integration process.

The choice of the failure detection and isolation (FDI) algorithm also has an impact on the complexity of the resulting detection logic. In software-implemented hardware fault tolerance, multiple model methods result in the highest detection logic complexity. When failure sensitive filter and signature analysis methods are employed, complexity is reduced, and when parity techniques are used, the simplest detection logics result.

In the next sections, we discuss how the various system-theoretic techniques can be used to design failure detection mechanisms for software faults.

4.1 Static Consistency Tests

Consistency analysis is an extension of the parity methods discussed in Chapter 2. These system-theoretic fault tolerance techniques will be extended to the design of acceptance tests employed in fault tolerant software. We begin our discussion of how this can be accomplished by first considering static consistency.

Static consistency relations are based on the static redundancy arising from a knowledge of the transformation which relates the internal program states of a given software module to the inputs and outputs of that program block, at a given instant of the computation cycle. Static redundancy comes about when two or more module outputs are related to each other through the algorithm used in the module. We describe this static consistency concept by first considering the following examples.

First, consider the example in Section 3.1, involving a subroutine providing an ordinary matrix inversion for real nonsingular square matrices. In this case, the system model description of the module would directly provide the necessary consistency check:

$$||AB - I|| < s \qquad (4.1)$$

where A is the input matrix and B is the computed output matrix. Note that the effort involved in the programming of the consistency relationship (matrix multiplication, subtraction, and

evaluation of a matrix norm) would be less than that of programming another matrix inversion module for checking.

For another example of a static consistency check, consider a subroutine for computing the eigenvalues of real square matrices. In this case, the software specification would be to compute n complex numbers such that

$$\det(A - \lambda_i I) = 0 \tag{4.2}$$

for each real square matrix A of order n. A system model for this module could be defined by the following transformation f:

$$f: R^{nxn} \rightarrow C^n \tag{4.3}$$

such that for each A in R^{nxn}, $f: R^{nxn} \rightarrow C^n$ maps A into λ, in accordance with

$$\lambda = f(A) + w \qquad \text{with } \det(A-\lambda_i I)=0 \quad i=1,..,n \tag{4.4}$$

where w is a zero mean vector with a specified variance depending on the algorithm used in computing the eigenvalue (Power, Jacobi etc.), the condition number of the input matrix A, and the precision of the computing hardware.

Consider an acceptance test based on finding the inverse of the implemented algorithm. Referring to our discussion in Section 3.2, the transformation in (4.4) is not one-to-one. That is, there are numerous matrices mapped into a single set of eigenvalues. Hence, there is not a functional inverse for this example. However, the inverse of the associated set function [34] can be used instead. In this case, the question becomes one of determining whether or not the input matrix A is in the

inverse set of $\{\lambda_1,...,\lambda_n\}$. This problem is equivalent to proving the existence or nonexistence of a nonsingular matrix, T, such that

$$A = TJ(\lambda) T^{-1} \tag{4.5}$$

where J is a Jordan canonical form [35] which is a function of the eigenvalues of A. The complexity in solving (4.5) is at a level similar to that of the original eigenvalue problem, however. Hence, the inversion approach does not, for this example, yield a feasible acceptance test.

Another possible acceptance test can be derived from a direct evaluation of the system model (or the software specification) which would be given by:

$$|\det(A-\lambda_i I)| < s \qquad i=1,..,n \tag{4.6}$$

where the scalar s is a threshold reflecting the inaccuracy of the implemented matrix inversion algorithm. Note that the evaluation of the determinant in (4.6) above can be quite involved, especially for large order matrices.

However, we can develop simpler consistency checks based on the properties of the implemented transformation. For instance, one such consistency check is given by the following

$$|\text{trace}(A) - \sum \lambda_i| < s \tag{4.7}$$

where the threshold is a function of the errors in the computed eigenvalues and trace function. Note that this consistency check is much simpler than either finding a solution to the inversion problem of (4.5), or evaluating the satisfaction of the system

model constraint of (4.4). Moreover, the threshold selection for (4.7) is simpler than for (4.6).

As a final example of a static consistency check, consider a software module performing the transformation of a 3 dimensional vector, x, (representing a physical variable such as acceleration or angular velocity) from a vehicle body-axis coordinate frame to, say, a local level navigation frame. In this case, the specification would be given by

$$y = Tx \tag{4.8}$$

where the 3 x 3 transformation matrix is given by:

$$T(\phi,\theta,\psi) = \begin{bmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & s\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \tag{4.9}$$

where ϕ,θ,ψ are the Euler angles representing the aircraft's attitude with respect to the local navigation frame, and c,s are abbreviations for cosine and sine functions respectively. For this problem, a consistency check based on the system specification constraint would be just as complex as the algorithm itself. However, we can develop a simpler consistency check based on the property that a transformation between two orthogonal reference frames does not change the length of a vector, so that, accounting for computational inaccuracies, we would require for consistency that

$$|y'y - x'x| < s \tag{4.10}$$

where x' is the transpose of the vector x, and x'x, is, of course, the length squared of x, and s is an appropriate thres-

hold value for acceptance. Again, the consistency check is much simpler to implement than a brute force approach based directly on the system specification.

There are quite a number of other known analytic relationships (available in function/ matrix/ systems theory literature) which can be used for consistency checks. Our point is that these consistency check candidates are already developed, and are not currently exploited in either current state-of-the-art software development packages such as LINPACK [36] or EISPACK [37] nor in real-time math libraries supporting, for instance, flight control software applications.

A library of well-tested consistency checks, such as the ones described above, can be developed for specific applications. This would reduce the programming effort involved in getting the required consistency checks, and allow for significant programming savings arising from the common consistency relations applicable to different program modules.

4.2 Dynamic Consistency

Dynamic consistency is based on temporal redundancy arising from the knowledge of the rules governing the time evolution of the inputs and outputs of a functional module. In the context of fault tolerant software considered here, dynamic consistency relations can be similarly obtained by using an appropriate dynamic system model for the program module. For instance, if

the model of the program module can be described by a linear (ARMA) autoregressive moving average (i.e., current module output is a finite linear combination of past inputs and outputs), then the corresponding consistency relations can be obtained using standard linear system theory concepts. Again, the practical requirement would be to obtain (through approximations, if necessary) consistency relations which require substantially less programming effort to implement than the actual code of the program module itself.

We can illustrate the dynamic consistency concept with a basic integration routine. The specification for the problem is to simulate, in a digital computer, the behavior of a physical system described by the differential equation

$$\dot{x}(t) = f(x(t),u(t),t) \qquad (4.11)$$

Referring to Figure 2.1, u is the input and x is the internal state vector of the software module. The output of the module is the computed state x, and f is an arbitrary function of the variables x, u, and t (A suitable sampling mechanism on u is implicitly assumed).

Suppose that the software module for the specification above is implemented using a fourth-order Runge-Kutta integration routine which requires the evaluation of the derivative expression (4.11) at four adjacent points. Now, to derive a dynamic consistency check we can use a simpler system model for the software block, such as

$$x(k+1) = x(k) + T\ f(x(k),u(k),k) + w(k) \qquad (4.12)$$

where k represents t_k, T is the sampling interval, and w(k) is a zero-mean sequence of random vectors representing the expected error of this model in simulating the behavior of the implemented software module (which, in turn, is simulating the behavior of the system defined by (4.11)). Based on our system model of (4.12), a dynamic consistency check is given by the following:

$$|x(k+1) - x(k) - T\ f(x(k),u(k),k)| < s \qquad (4.13)$$

A generalization of this dynamic consistency approach to linear dynamic systems described by the state space model, illustrated in Figure 2.2, and specified by:

$$x(k+1) = A\ x(k) + Bu(k) + w(k) \qquad (4.14)$$

$$y(k+1) = C\ x(k+1) + v(k+1) \qquad (4.15)$$

is given in Appendix A.

While the technical details presented in the Appendix are involved, we would like to stress that the system theoretic approach outlined provides a procedure for generating a dynamic consistency check for <u>any</u> software module algorithm that can be approximated by (4.14)-(4.15). This is in contrast to the static consistency check which must rely on known (problem specific) analytic relationships from the literature. We will now give an example utilizing the results obtained in Appendix A.

Consider the software specification for an aircraft state estimator within the coverage of a navigation aid. In this problem, the software is required to provide estimates for the vehicle states satisfying

$$\ddot{x}(t) = u(t) + w(t) \tag{4.16}$$

where x is the three dimensional aircraft position vector and w is a random process representing gust inputs into the aircraft, and u is the measured acceleration inputs. The estimator is asked to provide estimates for the vehicle states using the inputs u, and the measurements given by

$$y(t) = h(x(t)) + v(t) \tag{4.17}$$

where h is a nonlinear transformation relating states to the navigation aid measurements, and v is a random process representing the sensor noise.

Suppose that an application software module, on the order of a thousand lines of code, and implementing a nonlinear filter [20], has been written for this problem. This nonlinear filter would provide the sequences for state estimates, $\hat{x}(k)$, and, the measurement predictions, $\hat{y}(k)$.

Consider now the problem of obtaining a system model for this estimator software block in order to obtain an acceptance test. Denoting this model's state representing the software block's estimate by x_m, we can use the following <u>linear</u> model

$$x_m(k+1) = x_m(k) + T\dot{x}_m(k) \tag{4.18}$$

$$\dot{x}_m(k+1) = \dot{x}_m(k) + Tu(k) + w_m(k) \tag{4.19}$$

to develop the acceptance tests. In (4.18) above, $x_m(k)$ and $\dot{x}_m(k)$ represent available outputs of the software block representing the estimated position and velocity vectors, and u is the input into the software module. Eqns. (4.18-19) represent the

simplest linear model that can be used for this software module. Other linear models, for example, linear complementary filter models [18] are also possible. We will now apply the results obtained in Appendix A to find an acceptance test for this example.

The results of Appendix A provide a means for finding a dynamic consistency relation involving any selected subset of inputs and outputs of a software module. For instance, in order to find a dynamic consistency relation involving the input, u, and output, x_m, in the example considered, we let (using the notation in Appendix A):

$$A = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ T \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

In order to have dynamic redundancy, we need to consider at least a sequence of three outputs so that the observability matrix would be given by:

$$C(3) = \begin{bmatrix} 1 & 0 \\ 1 & T \\ 1 & 2T \end{bmatrix}$$

Carrying out the computations specified in Appendix A, we get:

$$I - C(3)C^{\#}(3) = 1/6 \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

and

$$B(3) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ T & 0 & 0 \end{bmatrix}$$

so that we obtain the following acceptance test using (A.13):

$$||x_m(k) - 2x_m(k-1) + x_m(k-2) - T^2u(k-2)||<s \qquad (4.20)$$

where the threshold s would be selected by considering the inaccuracy of the linear model in modelling the nonlinear software module.

4.3 Innovations Signature Analysis

System-theoretic innovations signature analysis can also be used to develop fault detection algorithms for software modules. This method can be used to obtain residual sequences between the inputs and the values predicted by the model of the software module. If the predicted values are already available, then this technique would not introduce additional undue computational complexity. We illustrate this concept with the following estimation problem.

Consider a software module generating a recursive least squares state estimate for the linear dynamic system state described by (4.14) and (4.15). The software specification for this state estimation problem results in the following algorithm depicted in Figure 4.1:

$$\hat{x}(k+1) = \hat{x}(k+1/k) + K(k+1)\ r(k+1) \qquad (4.21)$$

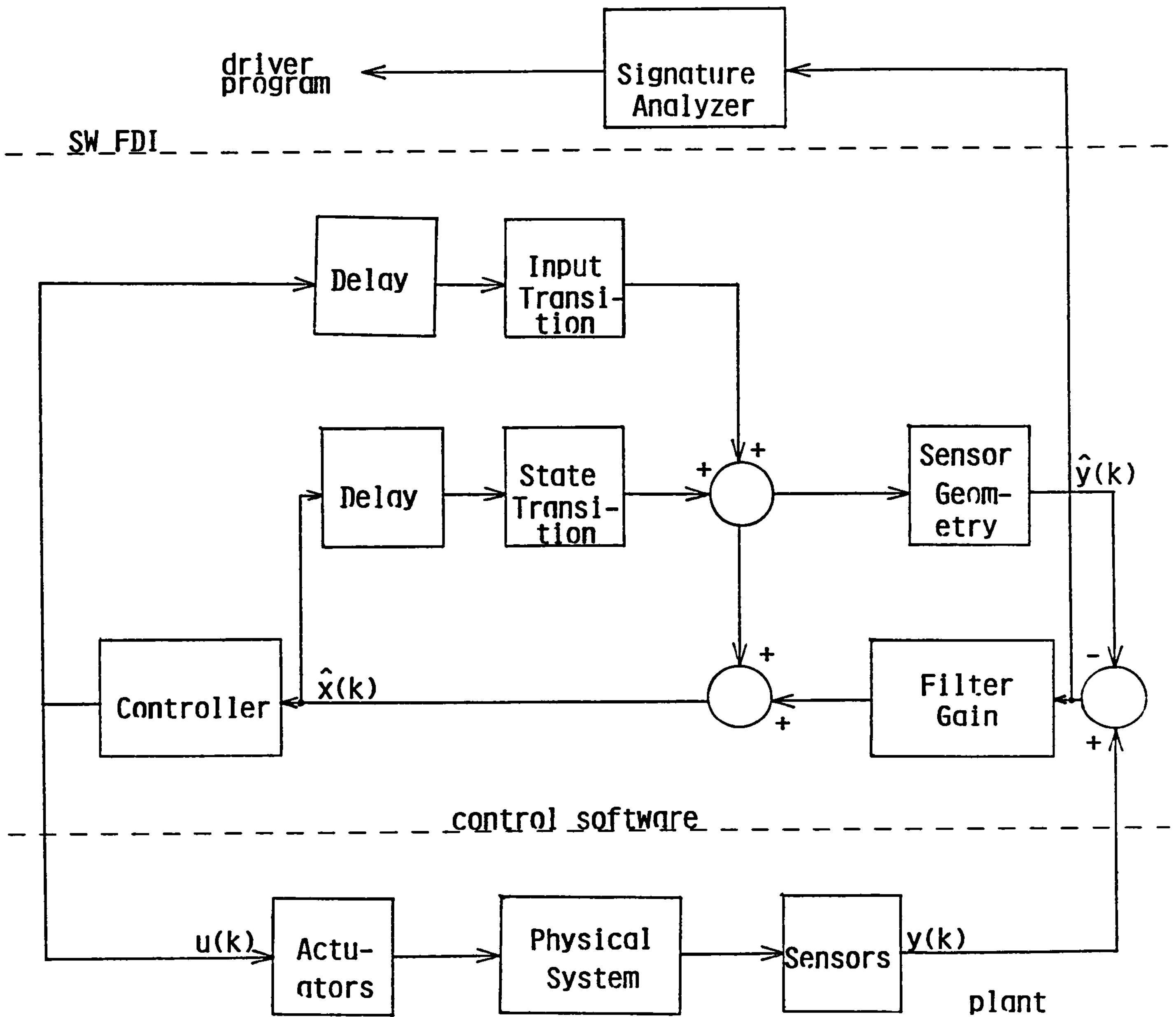

Figure 4.2: Software Fault Signature Analyzer

where $\hat{x}(k)$ is the optimal least squares estimate for the state $x(k)$ using all the current and past inputs u, and measurements y, up to time k. The other variables are defined by

$$\hat{x}(k+1/k) = A\hat{x}(k) + Bu(k) \quad (4.22)$$

$$r(k+1) = y(k+1) - \hat{y}(k+1/k) \quad (4.23)$$

$$\hat{y}(k+1/k) = C\hat{x}(k+1/k) \quad (4.24)$$

where $\hat{x}(k+1/k)$ is the single stage prediction for the state, and $r(k)$ is the measurement innovations sequence. The real complexity in programming this algorithm is in the recursive matrix computations required to generate the gain matrix $K(k+1)$, which involves several matrix multiplications, additions, and inversions.

Considered as a software module, the vectors u and y would be the inputs, and the state estimate $\hat{x}$ and the single stage measurement prediction $\hat{y}(k+1/k)$ would be the outputs. The innovations (residual) sequence, $r(k)$, would be analyzed to determine software faults in the module. Note that the residual sequence could be computed external to the module (and therefore checked externally), on the basis of (4.23) and the software module output, $\hat{y}(k+1/k)$, and, input, $y(k+1)$.

If this software module has no faults, and if the measurement sequence is generated by a physical system with a matching dynamic structure, then the innovations sequence, $r(k)$, would be a zero-mean uncorrelated sequence of random vectors. Hence, any appropriate statistical decision test could be implemented as an

acceptance test to determine whether or not the residual sequence has the postulated statistical properties. For example, the weighted sum of the squares of the residuals computed over a moving time window could be compared against a set threshold; exceedance of the threshold would indicate a model mismatch and a potential failure in the software module. The preceeding example allows also the potential for integrating hardware and software fault tolerance since the same residual sequence can be analyzed to detect hardware faults as well.

5. SYSTEMS-BASED SOFTWARE ERROR RECOVERY TECHNIQUES

In Section 2.4, we outlined major techniques for failure recovery in software-implemented hardware fault tolerance. These systems-based failure recovery techniques are extendable to software error recovery as well. The major issues of concern in software fault recovery, especially at the level of application level programming involving real-time process control, are the following:

- weak encapsulation due to memory and feedback
- detection delays
- noisy data
- state conversion in recovery blocks
- inexact voting and acceptance tests
- version instability.

We discuss these issues briefly in the following paragraphs.

Encapsulation is an operating system programming approach in which program computations are achieved using blocks with well-defined boundaries. The advantages of the encapsulation principle are to preserve data integrity in the presence of complex interactions of tasks which are subject to failure in the case of data encapsulation, and to preserve logical flow of control in the case of program encapsulation. In application level programming for dynamic systems, it is usually possible to group software functions into modules with well-defined boundaries. While the encapsulation principle imposes a desirable structured

approach to application programming development, its use in minimizing the effects of faults in fault tolerant application software for dynamic systems is limited both due to the extensive use of feedback and memory in algorithms, and the presence of detection delays.

The use of feedback is very common in application level programming for dynamic systems. Hence, even though the computational blocks may have well-defined boundaries, the use of feedback configurations, and transformations with memory, could propagate a software fault in a given module to other blocks. This problem is especially important if software faults are not instantaneously detected. In practical applications, there is usually a detection delay to ensure acceptable false alarm performance. This delay is naturally longer for "soft" failures in comparison with "hard" failures. These issues have been addressed in systems-based fault tolerant methods by various techniques, for instance, by modifying not only the value of a variable after the identification of a fault but also by modifying the level of confidence in that computation variable after a failure.

In application level programming, it is imperative to have consistency checks tolerant of noises in the inputs. This issue is critical both in dynamic (computations with memory) and static (memoryless computations) software blocks. For example, consider a static software module solving a specific least squares

problem. An acceptance test for this application can not take the least square fit and identically generate the inputs since the noise effects are filtered out by the least squares algorithm. Hence, in static least squares and in dynamic estimation problems, the acceptance test must compensate for the program input noise.

Another critical issue in software error recovery is the initialization of the program states in the recovery block algorithm. Clearly, if the variables needed in initializing the alternate block algorithm are directly available from the primary module computations, then this initialization process would be a straightforward process. However, in most applications, these variables are not readily available, since the use of different algorithms is essential in software fault tolerance.

In application level programming it is also crucial to have acceptance tests and voting checks compensating not only for finite precision arithmetic errors, but also for numerical errors associated with algorithm accuracy. For example, this issue would not arise in an operating system program involving the sorting of an array of integers. In contrast, any practical scientific computing application program would involve numerical errors due to the inaccuracy of the algorithm used. For instance, a polynominal root finder would not always compute zeros that identically satisfy the polynominal equation being solved.

Finally, a very important problem with N-version programming is the drift of unselected versions. While this issue is not related to error recovery since it is not caused by a software fault, we can consider it within the context of error recovery procedures. These drifts occur due to the accummulated rounding errors in the unselected versions in N-version programming.

In the next sections, we will discuss how the software error recovery issues outlined above can be addressed by applying software-implemented hardware failure recovery techniques, and by generalizing these methods to the domain of fault tolerant software.

5.1 Systems-Based Software Error Recovery Procedures

As discussed in Anderson and Lee [1], the drawbacks of using forward error recovery are its usual dependence on damage assessment, and anticipation of faults. The systems-based software implemented hardware failure recovery techniques discussed in Section 2.4 can be used to minimize, if not alleviate, these problems associated with forward error recovery.

For instance, consider the method of reinitialization of the state estimate and the estimation error covariance matrix associated with this estimate. Extending this approach to the domain of software fault tolerance, when a software error is detected in a software module, the program states are initialized by using current inputs into the module according to the proce-

dure employed at the start of the execution of the module. If there is a covariance associated with the computed state, this covariance can be set to its initial value. To illustrate the point consider the estimation problem discussed in Section 4.3. Suppose that this algorithm is initialized via:

$$\hat{x}(0) = Hy(0) \tag{5.1}$$

$$K(1) = f(P(0)) \tag{5.2}$$

where H is an appropriately dimensioned matrix relating the measurements to the program states, and the gain K(1) is a function of the initial uncertainty (covariance) of the state. If, for example, a software error is detected at the k'th instant, then the method above requires the implementation of:

$$\hat{x}(k) = Hy(k) \tag{5.3}$$

$$K(k) = f(P(0)) \tag{5.4}$$

Clearly, this forward error recovery procedure does not depend on an exact assessment and prediction of the damage. Hence, it is an appropriate means of recovery from unanticipated software faults.

Systems-based error recovery techniques can also yield procedures depending on only a partial assessment of the detected fault. Consider, for example, the application of the conditional covariance technique, discussed in Section 2.4, to the development of a forward error recovery procedure for software faults. In applying this method to software faults, the program state of the software block would not be changed after the detection of a

software error. However, the covariance of the program state would be increased by an amount depending on the type and level of the failure detected. Hence, only the covariance for those states deemed to be corrupted by the fault would be changed. Computational states would thus be gradually compensated through the algorithm dynamics. This approach is especially useful when a sudden change in a program state is not desired, especially in closed-loop control applications. Summarizing, the conditional covariance technique does not depend on determining the exact time of failure and level, but does depend on a precise determination of the type of failure, through the detection logic.

Finally, the techniques described above are applicable to algorithms where there is not an explicit covariance associated with the program state. In these cases, however, a measure of uncertainty needs to be developed for the implemented algorithm.

5.2 Systems-Based Recovery Block Initialization

When an acceptance test on a primary block declares that module faulty, then the program states in the alternate block need to be initialized (see Figure 5.1). Since software fault tolerance techniques hinge on the use of diverse algorithms, the internal program variables in the two modules will be different (in number, in physical meaning, etc.). Therefore, there needs to be procedure for converting the program states of the primary block into the equivalent alternate block states. This problem can be treated conveniently as an estimation problem, in the

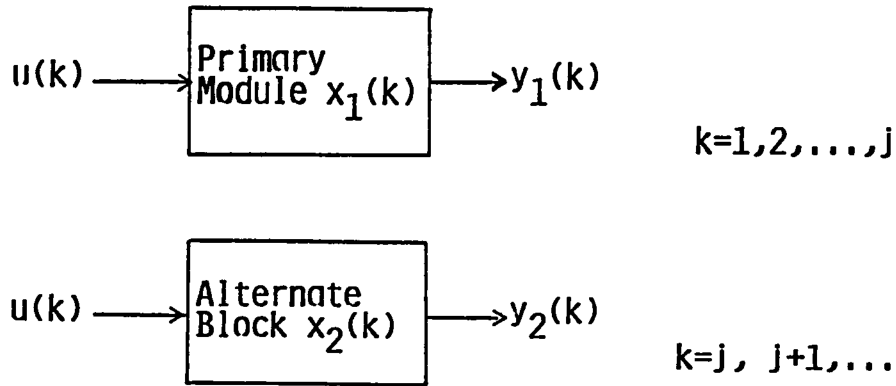

Figure 5.1: Recovery Block State Initialization Problem

context of system theory. We now formally define this problem, and give a solution for the case when the recovery block algorithm is represented by a linear dynamic system model.

Referring to Figure 2.2 and Section 3.2, consider the state space model for the primary block software, given by (3.5) and (3.6):

$$x_1(k) = f_1(x_1(k-1),u(k-1),w_1(k-1),k) \tag{5.5}$$

$$y_1(k) = h_1(x_1(k),u(k),w_1(k),k) \tag{5.6}$$

$$k = 1,2,...$$

where u(k) is the input sequence into the primary block, $x_1(k)$ is the primary block state, $w_1(k)$ is the random sequence modelling

the inaccuracies of the primary block, and $y_1(k)$ is the output sequence from the primary block. Note that a state-space rather than an ARMA representation is used, since the problem is with internal state initialization. Now, consider the alternate block representation:

$$x_2(k) = f_2(x_2(k-1), u(k-1), w_2(k-1), k) \qquad (5.7)$$

$$y_2(k) = h_2(x_2(k), u(k), w_2(k), k) \qquad (5.8)$$

where $x_2(k)$, $w_2(k)$, and $y_2(k)$ are the alternate block's state, random noise state, and output sequences, respectively. The transformations f_1, h_1, f_2, and h_2 are naturally different, reflecting the diversity of the two algorithms. Note that the primary block implementing (5.5) and (5.6) would be executed until a fault is detected, and the recovery block implementing (5.7) and (5.8) would be executed after the detection of that failure. Suppose the damage assessment for this failure requires the switching from primary to alternate block after the k'th instant. The problem is to determine the value of the state $x_2(k)$ which ensures that the output of the alternate block, after implementation, closely approximates the output that would have been generated by the primary block, had switching from primary to alternate not occurred. Stated formally, find an estimate of the alternate block internal states, $\hat{x}_2(k)$, as a function of the primary block inputs, outputs, and states, at time k, such that the following cost function

$$E \sum_{j=1}^{k} \left\| y_1(j) - y_2(j) \right\| \qquad (5.9)$$

is minimized for all input sequences, $u(k)$. Here $\|\cdot\|$ is a suitable vector norm, and E is the expectation operator. Note that the dimensions of the output vectors will be the same since the two blocks must be equivalent from an input/output point of view. If the algorithms are deterministic, we can equivalently state the problem as: Find $\hat{x}_2(k)$ such that

$$y_1(i) = y_2(i) \text{ for } i = 1,2,..,k \qquad (5.10)$$

and for all sequences, $u(k)$.

In general, it is not possible to compute the recovery block state estimate, $\hat{x}_2(k)$, from only the current primary module input, $u(k)$, state, $x_1(k)$, and output, $y_1(k)$ corresponding to a single computational frame. The next question is whether the alternate block estimate, $\hat{x}_2(k)$, is computable from a subset of the past primary block variables, or whether all of the past primary block variables are needed for this computation. This question is closely related to the observability problem for dynamic systems. In fact, we now show that the alternate block state estimate, $\hat{x}_2(k)$, can be determined using only the n previous inputs, $\{u(j), j=k-n,..,k-1\}$, and outputs, $\{y_1(j), j=k-n+1,...,k\}$, of the primary block, if the alternate block dynamic system is linear and observable in the system theoretic sense. That is, there is no need to start the execu-

tion of the alternate module from the beginning of the input data sequence. Note also that there are no restrictions placed on the primary block.

To illustrate, consider now the recovery block representation defined by eqs. (5.7) and (5.8) for a linear dynamic system defined by:

$$x_2(k) = A_2\, x_2(k-1) + B_2\, u(k-1) + w_2(k-1) \tag{5.11}$$

$$y_2(k) = C_2 x_2(k) + v_2(k) \tag{5.12}$$

The dynamic system above is the same one considered in Appendix A, with noise sequences $w_2(k)$ and $v_2(k)$. For simplicity of presentation, we consider the deterministic case where the noises are ignored. Now, the recovery block output at time k is related to the recovery block state at time k-n+1 via [23]:

$$y_2(k) = C_2 A_2^n\, x_2(k-n+1) + \sum_{i=1}^{n-1} C_2 A_2^{n-i}\, B_2 u(k-i) \tag{5.13}$$

Using the notation in the Appendix, current and n-1 previous outputs of the primary block are then given by

$$Y_2(k) = C_2(n)\, x_2(n-k+1) + B_2(n)\, U(k) \tag{5.14}$$

where the matrices C_2 and B_2 are defined by eqs. A.8-9 in the Appendix. As it happens, the matrix $C_2(n)$ is the <u>observability matrix</u> for the recovery block system dynamics. For recovery block initialization, we desire output sequence equality in accordance with (5.10) above, so that

$$Y_2(k) = Y_1(k) \tag{5.15}$$

The recovery block state estimate at time k-n+1 is then given by

$$\hat{x}_2(k-n+1) = C_2^{\#}(n)\ [Y_1(k) - B_2(n)U(k)] \tag{5.16}$$

where

$$C_2^{\#}(n) = [C_2'(n)\ C_2(n)]^{-1}\ C_2'(n) \tag{5.17}$$

The observability assumption guarantees the existence of the inverse in (5.17) so that the recovery block state at time k can then be obtained by propagating, $\hat{x}_2(k-n+1)$, though (5.11).

Summarizing, for arbitrary time k, the recovery block state <u>can</u> be initialized from the n primary block outputs $\{y_1(k), y_2(k-1) \ldots, y_1(k-n+1)\}$, and from the n-1 inputs, $\{u(k-1), u(k-2), \ldots, u(k-n+1)\}$ for software modules which have a linear dynamic representation.

Since the recovery block initialization outlined above is application dependent and fairly involved, it is of interest to find other equivalent implementations for the recovery block to simplify the initialization procedure. We give one example below.

5.3 ARMA vs. State Space Models for Recovery Blocks

The preceeding section underscores the importance of algorithm choice for recovery blocks. That is, if there are a number of functionally equivalent computational algorithms, then the choice of a recovery block implementation should be dictated by reinitialization considerations. We illustrate this point by using the example in the previous section. For this example, the

ARMA model form of the recovery block would, in accordance with the discussion of section 3.2, have the form:

$$y_2(k) = \sum_{i=1}^{n} A_2(i)\, y_2(k-i) + \sum_{i=1}^{n-1} B_2(i)\, u(k-i) \qquad (5.18)$$

where $A_2(i)$ and $B_2(i)$ are computed in terms of the state space description matrices A_2, B_2 and C_2 with the use of z-transforms [25].

To ensure that this recovery block ARMA model produces the identical outputs as those from the primary block state space model (for the same input sequence) requires a very simple initialization procedure: the recovery block ARMA model can be initialized from the primary block outputs and inputs simply by setting

$$y_2(k-n)=y_1(k-n) \quad \ldots. \quad y_2(k-1)=y_1(k-1) \qquad (5.19)$$

The simplicity of this ARMA model initialization has significant relevance in flight control software applications.

6. FAULT TOLERANT FLIGHT SOFTWARE

In this chapter, we discuss the issues involved in applying software fault tolerance techniques to the development of flight software. Our discussion covers the stability of flight systems with fault tolerant software, the preservation of functional performance, the use of generic flight software blocks amenable to the introduction of software fault tolerance and, finally, a comparative evaluation of N-Version and recovery block methods for use in generic flight software blocks.

6.1 Stability Issues in Fault Tolerant Flight Software

There are two main stability issues involved in the application of software fault tolerance techniques to flight software:

- o software module stability
- o total system stability

Software module stability refers to the stability properties of a given fault tolerant software module. Total system stability, in contrast, deals with the overall stability of the composite system, including both the physical system and the control software. We begin our discussion with software module stability.

6.1.1 Software Module Stability

Software module stability problems can arise both in N-version and recovery block applications. In N-version program-

ming, the versions that are consistently not selected by the voting logic can, for instance, go unstable. In recovery block applications, excessive switching between the primary and alternate blocks due to an improperly designed acceptance test can also introduce instabilities.

We illustrate the potential instability problems in converting standard single version flight software into a fault tolerant implementation, by considering a 3-version implementation of a feedback control module for the linear dynamic system defined in Appendix A (eqs. A.1-2). The single string version of the controller software takes in the measurements, y(k), and generates the input, u(k), which drives the physical system in accordance with:

$$z(k+1) = E\ z(k) + F\ y(k) \tag{6.1}$$

$$u(k+1) = G\ z(k) + D\ y(k) \tag{6.2}$$

where z(k) is the internal controller state contained in the software module. One of the requirements of such a controller design is that the total system, including both the physical system and controller dynamics, should be stable. This would imply that the system dynamics (ignoring the noise states), given by:

$$\begin{matrix} x(k+1) = \\ z(k+1) = \end{matrix} \begin{bmatrix} A + BCD & BG \\ FC & E \end{bmatrix} \begin{bmatrix} x(k) \\ z(k) \end{bmatrix} \tag{6.3}$$

should be stable (i.e. the eigenvalues of the state transition matrix of (6.3) are all within the unit circle). However, this

does not necessarily imply that the controller software module described by (6.1) and (6.2) is stable. In fact, there are a number of practical examples in which some eigenvalues of E would be outside or on the unit circle. For instance, a controller employing integral feedback is such an example. Now, consider a 3-version implementation of (6.1) and (6.2). In this case, we would have three recursive relations controllers, each implementing the following:

$$z_i(k+1) = E_i z_i(k) + F_1 y(k) \qquad (6.4)$$

$$u_i(k) = G_i z_1(k) + H_i y(k) \qquad i=1,2,3 \qquad (6.5)$$

Suppose that the voting logic consistently selects the first version. In this case, the overall closed-loop system would be stable since the first controller version is designed to stabilize the system. The second and third blocks, with control outputs $u_2(k)$ and $u_3(k)$, would not be used; hence, these versions would not have the loop closure provided by the physical plant dynamics. Now, if one of these unselected controllers were neutrally stable or unstable, then we could expect that controller to go unstable, without the stabilizing effects of plant feedback.

We now give a specific example illustrating the version instability problem described above. Consider the N-version implementation of an aircraft pitch axis PID controller depicted in Figure 6.1. PID (proportional-integral-derivative) controllers are commonly used in flight control applications. In Figure

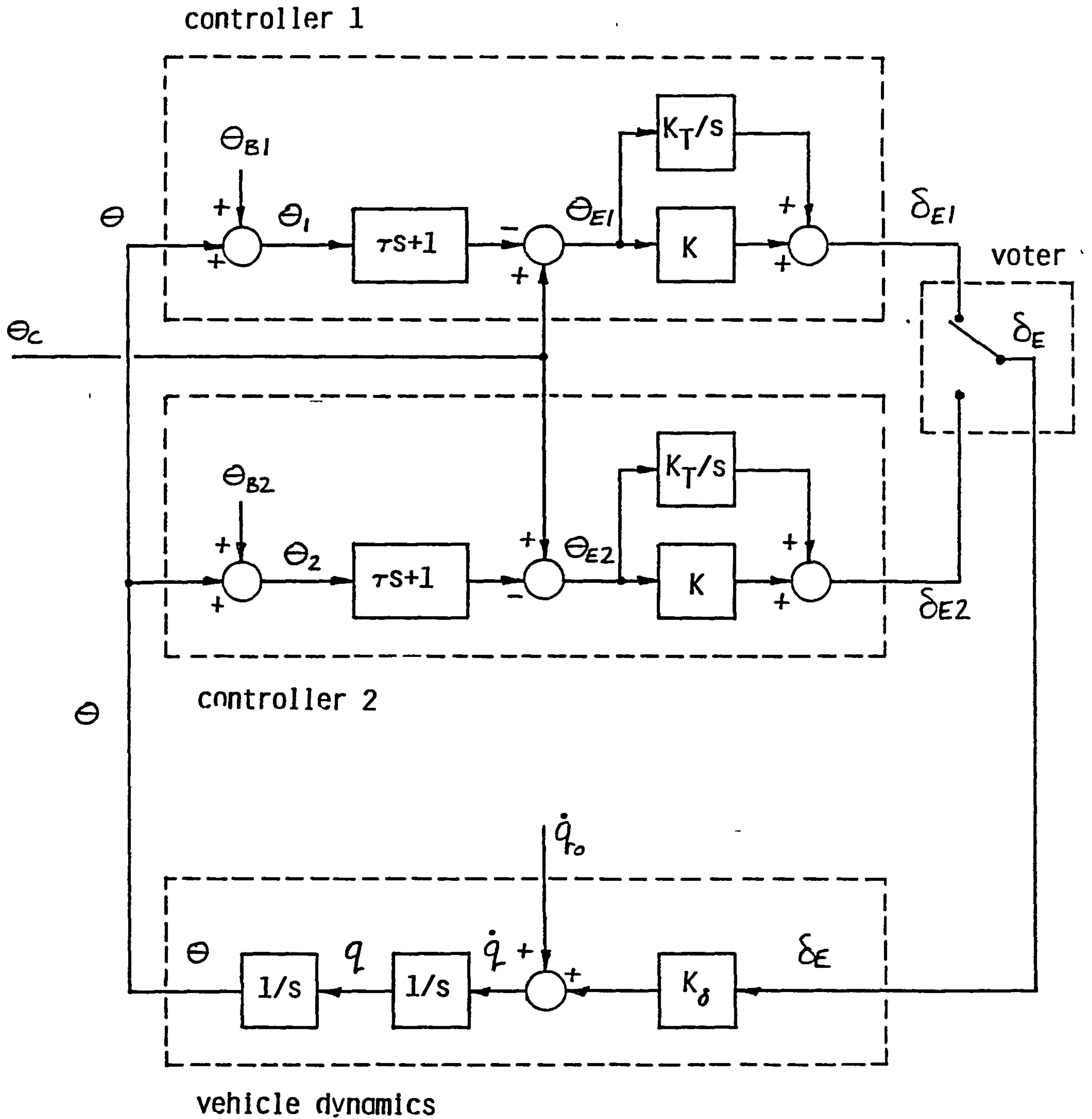

Figure 6.1: Version Instability in an N-Version PID Controller

6.1, the lower block represents the simplified aircraft pitch axis dynamics, while the top two blocks show the two versions (other versions not shown for clarity) of flight controller software blocks. The objective of the controller is to compute the elevator input δ_E in order to pitch the aircraft to the commanded attitude, θ_C. The controller achieves this by employing proportional feedback through the K block, integral feedback through the K_T/s block, and derivative feedback through the (τs + 1) block. The variables θ_{B1} and θ_{B2} in the controller software blocks may represent the different sensor biases driving each controller or other input differences such as those due to time delays.

Assume that the voter continually selects the first controller output. In this case, the pitch rate acceleration, $\dot{q}$, into the aircraft should be zero in steady-state. Hence, the elevator input, δ_E, and the first controller output value would be $-\dot{q}_O/K$ where $\dot{q}_O$ is an unknown torque into the system. In the first controller block, the integrator output should then be $-\dot{q}_O/K$ with an input, θ_{E1}, value of zero. This would imply that θ_1 is equal to the commanded attitude θ_C which, in turn, dictate that the steady state aircraft pitch attitude would be equal to $\theta_C - \theta_{B1}$.

Now, consider the second controller block, θ_2 would then be equal to $\theta_C - \theta_{B1} + \theta_{B2}$. This would imply that the input into the integrator would be given by $\theta_{B2} - \theta_{B1}$. If the sensor biases are not identical, then clearly the output of the second controller

would then be a ramp which would nullify the designed fault tolerance since the second block would not get selected by the voter even in the case of a software fault in the first controller. If θ_{B1} and θ_{B2} represented two independent measurement noises, then the second controller output would be a random walk process, again nullifying the designed fault tolerance.

In contrast, consider a 3-version implementation of a state estimator for the same physical plant. In this case, each filter would compute the state estimate $\hat{x}(k)$, implementing

$$\hat{x}_i(k+1) = \hat{x}_i(k+1/k) + K_i[y(k+1) - C\hat{x}_i(k+1/k)] \quad (6.6)$$

$$\hat{x}_i(k+1/k) = A\hat{x}_i(k)+B\,u(k) \quad (6.7)$$

In this scenario, even if a version is consistently selected out, that version would not go unstable since by design each filter version would be stable. That is, the filter dynamics described by

$$\hat{x}_i(k+1) = [A-K_iCA]\,\hat{x}_i(k) + [B-K_iCB]u(k)+K_iy(k+1) \quad (6.8)$$

would be stable since the eigenvalues of $[A-K_iCA]$ would be inside the unit circle.

Hence, from software module stability considerations, N-version programming should only be introduced around software blocks that are stable from an input/output point of view. For unstable software modules, N-version programming should be avoided.

6.1.2 System Stability with Fault Tolerant Software

We now turn our consideration from individual module stability to overall system stability. Most flight software blocks are designed to ensure stability of the overall closed-loop controller/vehicle configuration. For instance, a flight control software stability augmentation system should, when combined with the open-loop aircraft dynamics, yield a stable set of augmented vehicle dynamics, and produce a stable system. If N-version programming is introduced around the flight control software block, the resulting system should still remain stable. Now, overall system stability, in an N-version software environment, is closely coupled to the specific voting logic employed. Under normal operation, the voting logic will generate an output sequence comprised of "pieces" of the output sequences of the N separate versions. The number of "pieces" comprising a given length output sequence depends on the frequency of selecting between versions, which, in turn, can depend on the values of the inputs into the N versions, the relative accuracy of the implemented algorithms in the N modules, the noise level in the physical system, and the specifics of the voting logic.

We will illustrate the potential system instability problem with fault tolerant software by considering the example depicted in Figure 6.2. This example involves the design of a controller for a fifth order dynamic system. The eigenvalues for the open-loop system without feedback (zeroes of the polynominal in the

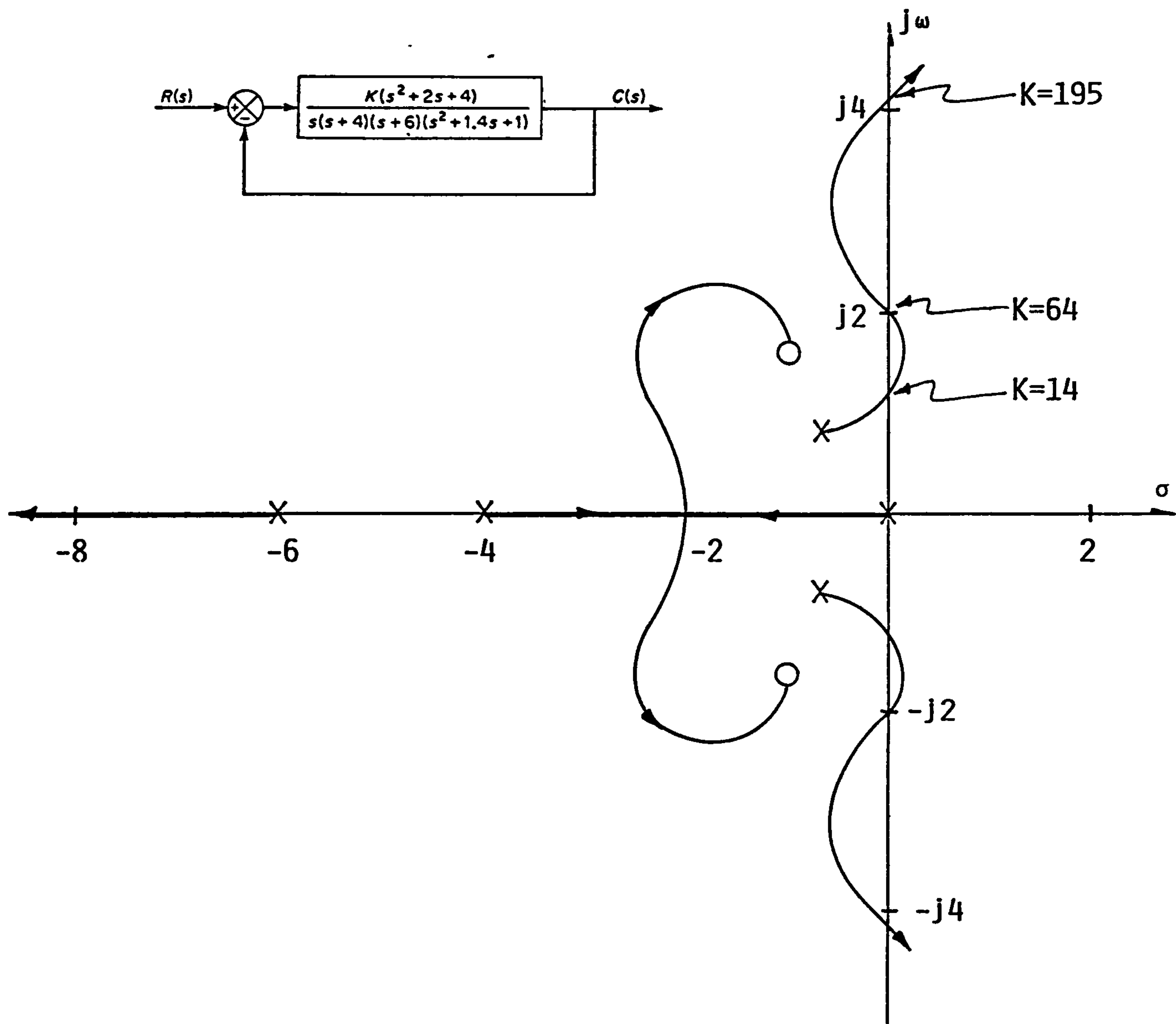

Figure 6.2: Example of Closed-Loop Instability with FTSW

denominator of the transfer function) are shown by "x", and the zeroes of the transfer function (zeroes of the polynominal in the numerator) are shown by "o" in the complex plane depicted in Figure 6.2. This figures also shows the variations in the closed-loop eigenvalues as the value of gain K is increased from zero. Note that for stability, the gain should be between either (0,14) or (64,195). Consider a 2-version implementation of this controller. Suppose for purposes of illustration that the first team selects a gain value of 12, whereas the second team selects a gain value of 66 based on the stability and performance specification for the controller software. Consider now a 2-version controller which uses the average of the two controller outputs when each passes an acceptance test. Clearly, this case would imply an effective controller gain of 39 which would result in an unstable closed-loop system.

This simple example illustrates the point that closed-loop stability with each separate version of an N-version program does not guarantee closed-loop stability of the system with the N-version software. As this example illustrates, N-version programming can introduce stability problems when the versions implement functionally dissimilar algorithms satisfying the same specification. Hence, to avoid potential closed-loop stability N-version programming should be restricted to the N different software implementations of functionally equivalent algorithms.

6.2 Preservation of Functional Performance

Preservation of functional performance refers to the requirement that the introduction of fault tolerance around a specific software block should be transparent to the rest of the software. This requirement can be addressed once it is determined that the software fault tolerance introduced does not impair stability, as just discussed. In N-version programming, the issue of functional performance preservation is more important than it is for recovery blocks, due to the normal switching expected between the N versions. In contrast, a properly designed recovery block application would not introduce excessive switching between the primary and alternate programs, which, if present could cause a significant impact on functional performance.

To illustrate, we refer back to the estimator problem considered in Section 6.1.1. Suppose that each module computes the state estimate with a steady-state bias error, i.e., as k gets large we have:

$$E\ [x(k) - \hat{x}_i(k)] = m_i \qquad (6.16)$$

where m_i is the constant bias in the state estimate $\hat{x}_i$ of the i'th software block. From a control point of view, a constant bias in the estimates are likely to be acceptable. However, switching between the versions would produce a sequence of bias jumps (due to the different m_i values generated by the selected version), and thus the state estimate provided by the N-version

programmed estimator block would contain errors whose frequency spectrum is determined by the distribution of switching frequencies. In effect, N-version switching can transform a simple bias, which may be compensated adequately by the control system, into wide spectrum noise, which could lead to a control performance degradation of the control system.

The issue of preserving functional performance in N-version programming can be analyzed by computing the statistical properties of the difference sequence between the version outputs. For instance, in the example just considered, consider the first version to be the "reference" version. Version differences can then be computed in accordance with:

$$\tilde{x}_{21}(k) = \hat{x}_2(k) - \hat{x}_1(k) \tag{6.9}$$

$$\tilde{x}_{31}(k) = \hat{x}_3(k) - \hat{x}_1(k) \tag{6.10}$$

Ideally, these sequences should be a zero mean uncorrelated sequence of random vectors for functionally equivalent dissimilar algorithms. Next, the effects of switching on the error sequence would be analyzed. For a given voting logic, the empirical statistics for the error sequence would be computed. For a given instant, when the first version is selected, the error would be zero. On the other hand, if the second version is selected, the error term for that instant would be given by (6.9).

Another approach to the assessment of functional performance preservation in fault tolerant software is to compute the performance deviation from the software specification. This approach

is, in essence, equivalent to finding a system model for the composite fault tolerant software block and the associated voting logic or acceptance test. In the example considered above, this approach would require the computation of the statistics of the sequence constructed from

$$\tilde{x}_1(k) = x(k) - \hat{x}_1(k) \tag{6.11}$$

$$\tilde{x}_2(k) = x(k) - \hat{x}_2(k) \tag{6.12}$$

$$\tilde{x}_3(k) = x(k) - \hat{x}_3(k) \tag{6.13}$$

according to the decisions dictated by the voting logic.

6.3 Generic Flight Software Blocks for Software Fault Tolerance

Modern flight software is comprised of functionally separate blocks, resident on different computers connected through a network which supports data and control communication between modules. These blocks can be categorized according to their functions as follows:

- navigation
- guidance
- state estimation
- flight control
- system monitoring
- housekeeping

Navigation software computes the aircraft's position, velocity, and attitude with respect to an external reference frame by using a subset of the aircraft's sensor suite. These

computed estimates are used by the guidance software to fly the aircraft from its current position to some defined position in the future, either along a prescribed path, or in some manner which specifies given trajectory objectives. Guidance software accomplishes this task by sending the required commands to the flight control software. The flight control software, in turn, computes the commands to drive the aircraft control surfaces and engines in order to perform maneuvers or maintain flight equilitrum as commanded by the guidance logic. Flight control software uses the state estimator block outputs which are the aircraft states estimates computed from a suitable subset of the aircraft sensor complement. Engine control software computes the engine controls needed to satisfy the flight control engine commands. System monitoring performs the redundancy management for the various redundant components by using FDI and BIT (built-in test) procedures. Finally, housekeeping software takes care of maintenance procedures such as the driving of the required displays, maintaining cabin pressures, etc.

In the context of our earlier discussion of system models, most of these flight software blocks are in a feedback configuration, with one another and with the aircraft dynamics. State estimation and navigation software blocks are usually designed to be stable when considered individually, but on the other hand, it is not uncommon to find applications when the guidance and flight control software blocks by themselves are neutrally stable or

unstable (for instance, when integral feedback is used to alleviate steady-state errors). It is also to be expected that the majority of these software modules will have memory as well, because of the need to provide iterative control, smoothing of state estimates and other like functions which demand knowledge of past software commands and vehicle responses.

6.4 N-Version vs. Recovery Blocks in Flight Software

Here we present a qualitative discussion comparing the feasibility of using N-version and recovery blocks in generic flight software blocks.

It is helpful to begin this discussion with a consideration of the various possible interpretations of algorithm dissimilarity. Recall that dissimilar algorithms are desired in software fault tolerance to maximize the independence of software errors in different versions. In the context of fault tolerant software, there are three possible "types" of algorithm dissimilarity:

- Type I: functionally dissimilar algorithms which satisfy the same given software specification,
- Type II: functionally equivalent dissimilar computational algorithms,
- Type III: dissimilar software implementations of the same computational algorithm.

To illustrate the first type of algorithm dissimilarity (type I), consider the design of an altitude-hold control law to be implemented in a 3-version programming fault tolerant structure. The objective of the altitude-hold system software module in an aircraft is to compute the pitch commands necessary to maintain a constant altitude by using the sink rate measurements. If each version were to be developed by an independent design team, consisting of a control engineer and a software engineer, then we might expect each team to come up with a functionally different algorithm for satisfying the specified control objective. That is, given the same input measurements, each version would be expected to produce significantly different output pitch commands particular to the control strategy used in that version. Hence, from an input/output point of view, the three versions would be functionally dissimilar, although each implemented law would accomplish the specified objective.

The second type of algorithm dissimilarity (type II) is that employed in functionally-equivalent but dissimilar computational algorithms. An example would be three software blocks for computing the eigenvalues of real square matrices in which one version implements Gaussian elimination, one implements Gram-Schmidt orthogonalization, and one implements Householder's method. For a given input matrix, each version could be expected to compute approximately the same values for the eigenvalues according to the accuracy of each algorithm, and the precision of the computing hardware. From an input-output point of view, they are thus all functionally equivalent, although highly dissimilar.

The third type of algorithm dissimilarity (type III) arises because of dissimilarities in software implementation, for a given computational algorithm. Extending our example above, consider three different software implementations of the Gaussian elimination algorithm, each implemented by a different programmer. It is likely that each programmer would write a different sequence of instructions for implementing this computational algorithm. For instance, it is likely that each programmer would write a different sequence of elementary row and column operations for performing the pivoting function. Hence, each of the three different implementations may not produce identical eigenvalues, although the same algorithm lies at the base of the software module.

Each of these three types of dissimilarity can be used to advantage in flight software applications for both N-version and recovery block configurations. However, there are advantages and disadvantages to using a given approach, depending on the generic flight software block under consideration. It should be clear that these different types of dissimilarity are increasingly restrictive, so that, if a type I is feasible then a type II is also feasible, and if a type II is feasible, then a type III is feasible.

Table 6.1 summarizes the feasibility of using N-version programming in the various generic flight software blocks discussed in the previous section. As indicated, the use of

functionally dissimilar N-version algorithms (type I) is generally not feasible in flight control or guidance software blocks, mainly due to the potential system instability discussed in Section 6.1.2. Additionally, one must be cautious in using either type II or type III algorithm dissimilarity in unstable software blocks since this can yield version instability as shown by the example in Figure 6.1.

Table 6.2 summarizes the feasibility of using recovery blocks in generic flight software blocks. As shown in this table, the advantage of using recovery blocks for flight control and guidance is that it can accommodate functionally dissimilar algorithms, (type I) since the system stability problems would be minimal in a properly designed recovery block configuration. In contrast as indicated in Table 6.1, functionally-equivalent but dissimilar control algorithms (type II) are feasible for use in guidance and flight control software blocks in an N-version configuration, since the closed-loop stability issue can be handled in this setup.

The use of functionally-equivalent (but dissimilar) estimation and detection algorithms (type II) is feasible for use in navigation, state estimation, and system monitoring blocks for both N-version and recovery blocks. However, N-version programming offers an extra advantage in terms of performance improvement for these generic blocks. For instance, if the estimator outputs are averaged in a 3-version programming state estimator

Software Block	Advantages	Disadvantages
navigation	type II dissimilar navigation algorithms improved estimation performance	slow execution speed large program size
guidance	type III dissimilar SW implementations	dissimilar guidance algorithms infeasible; potential system and version instability; transparency
state estimator	type II dissimilar estimation algorithms improved estimation performance	slow execution speed large program size
flight control	type III dissimilar SW implementations	dissimilar control algorithms infeasible; potential system and version instability; transparency
system monitoring	type II dissimilar detection algorithms improved detection performance	slow execution speed large program size
housekeeping	type III dissimilar SW implementations improved uptime	complexity

Table 6.1: N-version Programming Feasibility in Flight Software

Software Block	Advantages	Disadvantages
navigation	type II dissimilar navigation algorithms efficient, fault tolerant navigation	complex error recovery
guidance	type I dissimilar guidance algorithms efficient	potential switching transients
state estimator	type II dissimilar estimation algorithms efficient	complex error recovery
flight control	type I dissimilar control algorithms efficient, self-repairing control	potential switching transients
system monitoring	type II dissimilar detection algorithms efficient	slow detection speed
housekeeping	type II dissimilar algorithms improved uptime	complexity

Table 6.2: Recovery Block Programming Feasibility in Flight Software

application, it may be possible to reduce estimation error performance by using the average stated estimate. Similarly, it may be possible improve the false alarm and detection performance of a system monitoring software block by using an N-version programming approach.

We will give an example illustrating the potential for functional performance improvements in flight software through the use of software fault tolerance.

6.5 Performance Improvement with Fault Tolerant Software

We illustrate the potential for functional performance improvement by considering a software module which performs the management of redundancy in a skewed sensor array. For this example, we assume that the sensor array is a set of accelerometers and rate gyros in a semioctahedral configuration depicted Figure 6.3. Software is required to compute acceleration and angular rates, in the instrument reference frame, from the redundant accelerometer and rate gyro sensors mounted on the faces of the semioctahedron, all in the presence of possible sensor failures. This type of redundancy management software is currently implemented in commercial and military strapped down inertial measurement units; moreover, skewed sensor arrays will be common in the next generation of high-reliability integrated avionics. The example is thus highly relevant in the state-of-the-art flight software.

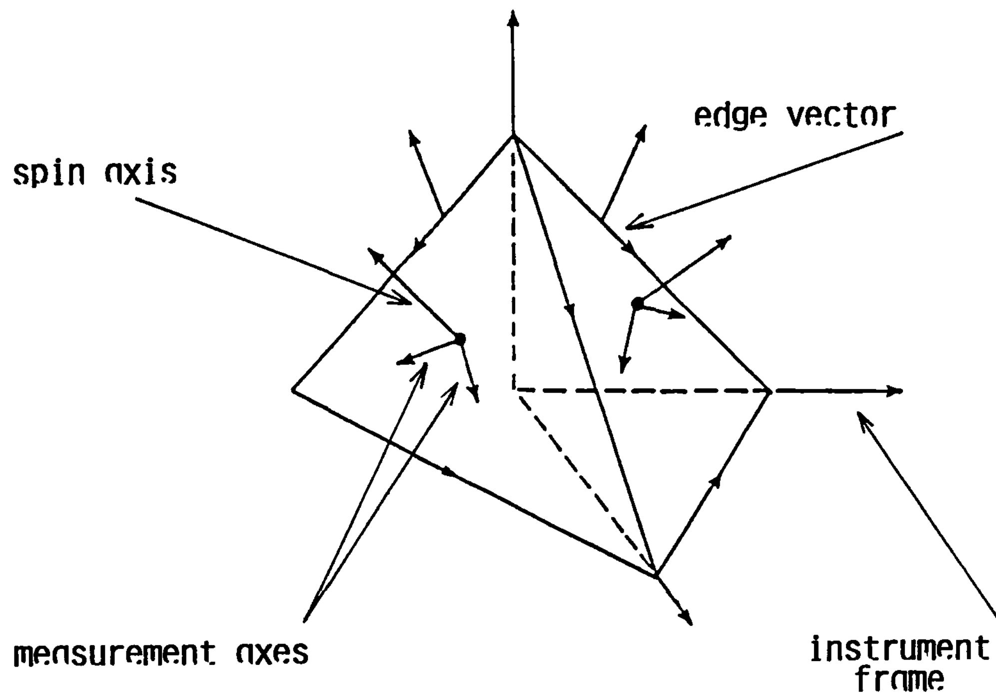

Figure 6.3: Semioctahedron Sensor Array Geometry

For this discussion, we assume that the skewed sensor array under consideration consists of two accelerometers and rate gyros mounted on each of the four different faces of a semioctahedron. There are five different coordinate systems of interest in this problem. One is the instrument reference frame, usually coincident with the body axes of the aircraft, and the other four are the individual sensor axes. Each of these sensor axes are customarily defined by the outward normal to the face (which makes the sensor spin axis) and by the two measurement axes which are symmetrically placed about the face centerline. Both accelerometer and rate gyro measurement axes, on a given face, coincide with each other.

Each sensor measures the physical variable of interest (acceleration or angular rate) along its two measurement axes. Geometrically, each sensor provides the projection of the three-dimensional acceleration (or rate) vector onto its two measurement axes which, in turn, determines the in-plane component of the acceleration (or rate) vector for that a specific face of the semioctahedron.

Software is required to compute the vehicle acceleration and angular rate in the instrument frame, from the redundant sensor measurements, using only healthy sensors. If, for example, a sensor is declared to be failed (by some means to be discussed below), then the computation of the physical variable is to be done without using the faulty instrument. These computations are

to be performed in the possible presence of up to two possible sensor failures of the same type.

There are at least three fundamentally different ways of designing the sensor failure detection and isolation software for this application:

- o edge vector test
- o parity test
- o generalized likelihood ratio test.

Each of these tests correspond to a different formulation of the hypothesis testing problem under consideration. For instance, the edge vector test (EVT) for the semioctahedral array involves resolving the outputs of each sensor along the edges of the semioctahedron for comparison across faces. If two sensors on neighboring faces are functioning properly, then their projections onto the common edge separating the faces should be approximately equal, within some threshold determined by the sensor noise characteristics.

In the parity test approach, eight measurement residuals are formed by subtracting the expected value for each measurement (generated from the estimated vehicle acceleration and body rate, and a knowledge of the semioctahedron geometry) from the actual measurements obtained from the sensor. Then, a minimal set of linearly independent relations are found from these eight residual relations, to test for postulated sensor failures.

In the generalized likelihood ratio (GLR) test approach, the measurement residuals are generated in the same fashion as in the

parity test, and are tested for zero mean under the various sensor failure hypotheses. This approach attempts to classify the sensor failures according to their effects on the measurement residuals, and thus identify the particular failed sensor.

Now, in a single-string software application, one of the sensor failure detection and isolation techniques would be implemented in software. Consider now a 3-version programming approach to this problem, where the first version performs the edge vector test, the second performs the parity test, and the third performs the generalized likelihood ratio test. We assume that a majority voting algorithm, which acts on the decision outputs of the three versions, has been implemented. Consider now the false alarm performance of the 3-version software, i.e., the percentage of cases in which the fault tolerant software voting logic will declare a sensor to be faulty although there is actually nothing wrong with that sensor. Ignoring the false alarms generated by the voter, this false alarm condition would occur when there is a false alarm for an identical sensor concurrently in either version 1 and 2, or version 1 and 3, or version 2 and 3. Depicting the false alarm event in the i'th version for the j'th sensor i by F_{ij}, the false alarm event in the fault tolerant software block for the i'th sensor will be given by

$$F_j = (F_{1j} \text{ and } F_{2j}) \text{ or } (F_{1j} \text{ and } F_{3j}) \text{ or } (F_{2j} \text{ and } F_{3j}) \quad (7.1)$$

Assuming that the false alarms are independent events,

$$F_j = (F_{1j} \text{ and } F_{2j}) \text{ or } (F_{1j} \text{ and } \bar{F}_{2j} \text{ and } F_{3j}) \text{ or } (\bar{F}_{1j} \text{ and } F_{2j} \text{ and } F_{3j}) \quad (7.2)$$

so that the probability of false alarm of the 3-version software block for the j'th sensor will be given by

$$P(F_j) = P_1P_2 + P_1(1-P_2)P_3 + (1-P_1)P_2P_3 \quad (7.3)$$

When $P_1=P_2=P_3=P$ and $P<<1$, then the false alarm for the overall fault tolerant software block for the j'th sensor would be approximately given by:

$$P(F_j) = 3P^2 \quad (7.4)$$

Hence, the false alarm rate of the fault tolerant software block would be substantially lower than that for a single version. If the assumption about the independence of false alarms in each version is violated, then the improvement in the false alarm rate would be less. This issue is similar to the problem of ensuring the independence of versions, so as to minimize correlated errors which act to reduce the desired reliability improvement. However, in the case of functional performance improvement, forced diversity through the choice of version algorithms can potentially minimize these correlated false alarms. In the next section, we will discuss the feasibility of introducing fault tolerant software into existing conventional software.

6.6 Adaptability of Fault Tolerant Software

We believe that fault tolerant software techniques can be introduced, with a minimal overhead, to conventional software with structural attributes resembling those for fault tolerant software. There are a number of such situations in flight software applications. For instance, consider a self-repairing

flight control system software in which there are multiple software control blocks each implementing a different control strategy. One of these blocks might be optimized to compute the optimal control strategy during normal operation under no failures. A second might be executed when it is determined that an aircraft actuator or surface has failed. The control strategy implemented in the second software module could be totally different from the first one. Naturally, the second strategy would not make use of the failed effectors in controlling the aircraft.

This self-repairing flight control software has a structure closely related to that of a recovery block module. The first software control module could be considered to be the primary module whereas the second software module (executed after a hardware failure) could be considered to be an alternate module. The only difference is that the alternate module is executed when a hardware failure is detected rather than a software fault. Clearly, software fault tolerance can be introduced into this system by simply implementing an acceptance test acting on the outputs of the software control modules, so that the software would tolerate not only external hardware faults but also internal latent design faults in the software control modules.

7. CONCLUSIONS AND RECOMMENDATIONS

We have presented a unified analysis of software and software implemented hardware fault tolerance methodologies, and have shown that systems-based failure detection, isolation, and compensation methods can be extended to the domain of software fault tolerance, by developing system models for software modules. We have demonstrated this system-based approach in a number of areas. For example, we have demonstrated that systems-based failure detection techniques can be used to develop consistency checks that are easier to implement than acceptance tests based on software specifications. We have also shown that the generalization of system-based failure recovery techniques to software fault tolerance yields forward error recovery procedures which do not depend on an exact assessment of the software error damage. Finally, using a systems formulation, we have found a solution to the recovery block state initialization problem, for the case when the alternate block algorithm can be approximated by a linear dynamic system.

We have also identified the basic system issues involved in applying software fault tolerance to flight software. In particular, we have demonstrated that introduction of fault tolerant software can potentially induce both version and closed-loop instabilities in flight software. In addition, we have presented a comparative evaluation of N-version and recovery block program-

ming techniques, in the context of generic flight software blocks. Finally, we have demonstrated the potential for functional performance improvement in flight software using fault tolerant software.

In summary, this study has shown that systems-based failure detection, isolation, and compensation techniques can be used to resolve significant issues in software fault tolerance. Our recommendation is to apply the developed technology to a real flight software problem at the level of complexity of the skewed sensor array redundancy management software discussed in the last chapter. We believe that there is much to be learned from such a demonstration experiment since the majority of past and current fault tolerant software experiments have been restricted to problems of academic interest.

APPENDIX A

DYNAMIC CONSISTENCY RELATIONS FOR LINEAR SYSTEMS

In this appendix, we derive the consistency relations discussed in Section 4.2 for software modules described by linear systems of the form:

$$x(k) = Ax(k-1) + Bu(k-1) + w(k-1) \tag{A.1}$$

$$y(k) = Cx(k) + v(k) \tag{A.2}$$

where referring to Figure 2.2, $x(k) = [x_1(k), x_2(k) \ldots x_n(k)]'$ is the n-dimensional internal program state, $u(k) = [u_1(k), u_2(k) \ldots u_p(k)]'$ is the p dimensional input vector, and $y(k)=[y_1(k), y_2(k) \ldots y_m(k)]'$ is the m-dimensional output vector. The white zero-mean random sequences, $w(k)$ and $v(k)$, represent the algorithmic and finite precision arithmetic errors introduced by the module.

We now derive an expression relating the measurement sequence, $\{y(k), y(k-1), \ldots y(k-n+1)\}$, to the input sequence, $\{u(k-1), u(k-2), \ldots u(k-n)\}$. Defining the nm vectors $Y(k)$ and $V(k)$, the nn vector $W(k)$, and the np-vector $U(k)$ by:

$$Y(k) = [y(k-n+1) \ldots\ y(k-1)\ y(k)]' \tag{A.3}$$

$$V(k) = [v(k-n+1) \ldots\ v(k-1)\ v(k)]' \tag{A.4}$$

$$W(k) = [w(k-n+1) \ldots\ w(k-1) w(k)]' \tag{A.5}$$

$$U(k) = [u(k-n+1) \ldots\ u(k-1) u(k)]' \tag{A.6}$$

we obtain from (A.1) and (A.2):

$$Y(k) = C(n)x(k-n+1) + B(n)U(k) + E(n)W(k) + V(k) \tag{A.7}$$

where C(n) is the observability matrix of the linear dynamic system defined by:

$$C(n) = \begin{bmatrix} C \\ CA \\ \cdot \\ \cdot \\ \cdot \\ CA^{n-1} \end{bmatrix} \tag{A.8}$$

B(n) is the nmx np dimensional matrix defined by

$$B(n) = \begin{bmatrix} 0 & \cdots & 0 & 0 & 0 \\ CB & \cdots & 0 & 0 & 0 \\ \cdot & \cdots & \cdot & \cdot & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdot \\ CA^{n-3}B & \cdots & CB & 0 & 0 \\ CA^{n-2}B & \cdots & CAB & CB & 0 \end{bmatrix} \tag{A.9}$$

and E(n) is the nmx n^2 dimensional matrix given by

$$E(n) = \begin{bmatrix} 0 & \cdots & 0 & 0 & 0 \\ CA & \cdots & \cdot & \cdot & \cdot \\ \cdot & & \cdot & \cdot & \cdot \\ \cdot & & \cdot & \cdot & \cdot \\ CA^{n-3} & \cdots & CA & 0 & 0 \\ CA^{n-2} & \cdots & CA^2 & CA & 0 \end{bmatrix} \tag{A.10}$$

If the linear dynamic system is observable, then the observability matrix C(n) would have full rank n. We can estimate x(k-n+1) via

$$\hat{x}(k-n+1) = C^{\#}(n)[Y(k) - B(n)U(k)] \tag{A.11}$$

where $C^{\#}(n)$ is the pseudoinverse of the observability matrix C(n). Using the estimate for x(k-n+1), we now define the residual sequence as:

$$Y(k) - C(n)\hat{x}(k-n+1) = [I - C(n)C^{\#}(n)]\ Y(k) \tag{A.12}$$

Substituting the expression for Y(k) above and using the property of the pseudoinverse ($C_n = C_n C^{\#}_n C_n$) as before, we get the following:

$$E\{I - C(n)C^{\#}(n)][Y(k) - B(n)U(k)]\} = 0 \qquad (A.13)$$

Eq. (A.13) above defines nm dynamic relations. Any one of these relations can be tested for zero mean by using the noise parameter values associated with W(k) and V(k). It can be shown there can be at most nm-n linearly independent dynamic consistency relations for this example.

Dynamic consistency relations generate an open-loop measurement residual sequence, in contrast to the closed-loop residual sequences produced by other fault diagnosis techniques employing state estimators.

REFERENCES

[1] Anderson, T. and Lee, P.A., _Fault Tolerance Principles and Practice_, Prentice Hall International, 1981.

[2] Randell, B., "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975.

[3] Hecht, H., "Fault Tolerant Software for Real-Time Applications", ACM Computing Surveys, Vol. 8, No. 4, December 1976.

[4] Avizienis, A. and Chen, L., "On the Implementation of N-Version Programming for Software Fault Tolerance during Program Execution", Proc. of COMPSAC 77, Chicago, IL, November 1977.

[5] Sklaroff, J.R., "Redundancy Management Technique for Space Shuttle Computers", IBM Journal of Research and Development, Vol. 20, No. 1, January 1978.

[6] Wensley, J.H. et al, "SIFT: Flight Design and Analysis of a Fault Tolerant Computer for Aircraft Control", _The Theory and Practice of Reliable System Design_, D.P. Siewiorek and R.S. Swarz, ed., Digital Press, 1982.

[7] Hopkins, A.L., Smith, T.B. and Lala, J.H., "FTMP -- A Highly Reliable Fault Tolerant Multi-Processor for Aircraft", _The Theory and Practice of Reliable Systems Design_, D.P. Siewiorek and R.S. Swarz, ed., digital press, 1982.

[8] Slivinski, T. et al, "Study of Fault-Tolerant Software Technology", NASA CR-172385, Sept. 1984.

[9] Pau, L.F., _Failure Diagnosis and Performance Monitoring_, Marcel Dekker, Inc., New York, 1981.

[10] Willsky, A.S., "A Survey of Design Methods for Failure Detection in Linear Systems", Automatica, Vol. 12, pp. 601-611, November 1976.

[11] Mozgalevskii, A.V., "Technical Diagnostics: Continuous Systems", Automatika i Telemekhanika, Vol. 39, No. 1, January 1978.

[12] Willsky, A.S., "Failure Detection in Dynamic Systems", AGARD-LS-109, _Fault Tolerance Design and Redundancy Management Techniques_, September 1980.

[13] Basserville, M., "Changes in Statistical Models: Various Approaches in Automatic Control and Statistics", Rapports de Recherche I.R.I.S.A., No. 72, May 1981.

[14] Cunningham, T., Carlson, D., Hendrick, R., Shanar, D., Hartmann, G. and Stein, D., "Fault Tolerant Digital Flight Control with Analytical Redundancy", AFFDL-TR-77-25, AFFDL, Wright-Patterson AFB, Ohio, May 1977.

[15] Deckert, J.C., Desai, M.N., Deyst, J.J. and Willsky, A.S., "Reliable Dual-Redundant Sensor Failure Detection and Identification for the NASA F-8 DFBW Aircraft", NASA CR-2944, February 1978.

[16] Beattie, E.C., LaPrad, R.F., McGlone, M.E., Rock, S.M., and Akhter, M.M.,: "Sensor Failure Detection System Final Report", NASA CR-165515, August 1981.

[17] Morelle, F.R. and Russell, J.G., "Design of a Developmental Dual Fail Operational Redundant Strapped Down Inertial Measurement Unit", NAECON 1980.

[18] Caglayan, A.K. and Lancraft, R.E., "An Aircraft Sensor Fault Tolerant System", NASA CR-165876, April 1982.

[19] Lancraft, R.E. and Caglayan, A.K., "FINDS: A Fault Inferring Nonlinear Detection System: User's Guide", NASA-172199, September 1983.

[20] Caglayan, A.K. and Lancraft, R.E., "A Fault Tolerant System for an Integrated Avionics Sensor Configuration", NASA CR-3834, Sept. 1984.

[21] Potter, J.E. and Suman, M.C., "Thresholdless Redundancy Management with Arrays of Skewed Instruments", AGARDOGRAPH 224, Integrity in Electronics Flight Control Systems, 1977.

[22] Chow, E.Y. and Willsky, A.S., "Issues in the Development of a General Design Algorithm for Reliable Failure Detection", Proc. 1980 Conference on Decision and Control, Albuquerque, NM, December 1980.

[23] Beard, R.U., "Failure Accommodation in Linear Systems Through Self-Reorganization", NASA CR-118314, 1971.

[24] Montgomery, R.C. and Caglayan, A.K., "Failure Accommodation in Digital Flight Control Systems by Bayesian Decision Theory", J. of Aircraft, Vol. 13, No. 2, pp. 69-75, February 1976.

[25] Kuo, B. C., Discrete-Data Control Systems, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.

[26] Willsky, A.S., Deyst, J.J. and Crawford, B.S., "Adaptive Filtering and Self-Test Methods for Failure Detection and Compensation", Proc. of the 1974 JACC, Austin, TX, June 1974.

[27] Caglayan, A.K., and Lancraft, R.E., "Reinitialization Issues in Fault Tolerant Systems", Proc. 1983 ACC, San Francisco, CA, June 1983.

[28] McAulay, R.J. and Denlinger, E., "A Decision-Directed Adaptive Tracker", IEEE Trans. on Aerospace and Electronic Systems, Vol. AES-9, March 1973, pp. 229-236.

[29] Willsky, A.S. and Jones, H.L.,"A Generalized Likelihood Ratio Approach to the Detection and Estimation of Jumps in Linear Systems", IEEE Trans. on Auto. Contr., Vol. AC-21, pp. 108-112, February 1976.

[30] Caglayan, A.K., "Necessary and Sufficient Conditions for Detectability of Jumps in Linear Systems", IEEE Trans. on Automatic Control, Vol AC-25, No. 4, August 1980.

[31] Caglayan, A.K., "Simultaneous Failure Detection and Estimation in Linear Systems", Proc. of 1980 Conf. on Decision and Control, December 10-12, 1980, Albuquerque, NM.

[32] Mehra, R.K., "An Innovations Approach to Fault Detection and Diagnosis in Dynamic Systems", Automatica, Vol. 7, pp. 637-640, 1971.

[33] Segen, J. and Sanderson, A.C., "Detecting Change in a Time-Series", IEEE Trans. on Information Theory, Vol. IT-26, March 1980.

[34] Naylor, A. W. and Sell, G. R., Linear Operator Theory, Holt, Rinehart and Winston, Inc., New York, 1971.

[35] Golub, G. H., and Van Loan, C. F., Matrix Computations, The John Hopkins University Press, Baltimore, MD, 1983.

[36] Dongarra, J., Bunch, J. R., Moler, C. B., and Stewart, G. W., LINPACK Users Guide, SIAM Publications, Philadelphia, PA, 1978.

[37] Smith, B. T., Boyle, J. M., Ikebe, Y., Klema, V. C., and Moler, C. B., Matrix Eigensystem Routines: EISPACK Guide, Springer Verlag, New York, NY 1970.

1 Report No. NASA CR-172618	2. Government Accession No	3. Recipient's Catalog No
4 Title and Subtitle Study of Fault Tolerant Software Technology for Dynamic Systems		5 Report Date September 1985
		6 Performing Organization Code
7 Author(s) Alper K. Caglayan and Greg L. Zacharias		8 Performing Organization Report No Report No. R8503
		10 Work Unit No
9 Performing Organization Name and Address Charles River Analytics, Inc. 55 Wheeler Street Cambridge, MA 02138		11 Contract or Grant No NAS1-17705
		13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14 Sponsoring Agency Code 505-37-13-03

15 Supplementary Notes

Langley Technical Monitor: Dave E. Eckhardt, Jr.

Preliminary Formal Report for Phase I

16 Abstract

The major aim of this study is to investigate the feasibility of using systems-based failure detection isolation and compensation (FDIC) techniques in building fault-tolerant software and extending them, whenever possible, to the domain of software fault tolerance. First, it is shown that systems-based FDIC methods can be extended to develop software error detection techniques by using system models for software modules. In particular, it is demonstrated that systems-based FDIC techniques can yield consistency checks that are easier to implement than acceptance tests based on software specifications.

Next, it is shown that systems-based failure compensation techniques can be generalized to the domain of software fault tolerance in developing software error recovery procedures. In particular, the systems approach yields forward error recovery procedures which do not depend on an exact assessment of the software damage.

Finally, the feasibility of using fault-tolerant software in flight software is investigated. In particular, possible system and version instabilities, and functional performance degradation that may occur in N-Version programming applications to flight software are illustrated. Finally, a comparative analysis of N-Version and recovery block techniques in the context of generic blocks in flight software is presented.

17. Key Words (Suggested by Author(s))	18 Distribution Statement
Software fault tolerance Fault tolerant systems Flight software	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price
Unclassified	Unclassified	92	

N-305 For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document