UNLIMITED

2

AD-A205 180

RSRE

MEMORANDUM No. 4237

ROYAL SIGNALS & RADAR ESTABLISHMENT

SOFTWARE FAULT TOLERANCE

Author: L N Simcox

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

89 3 06 176

UNLIMITED

RSRE MEMORANDUM No. 4237

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4237

SOFTWARE FAULT TOLERANCE

June 1988

Author: L N Simcox

SUMMARY

As part of its software engineering research for the Civil Aviation Authority (CAA) the ATC systems research division at RSRE has been studying software fault tolerance. This memorandum sets out the main concepts involved, discusses the current research and describes some of the aerospace applications. The cost-effectiveness of the technique is addressed briefly and the applicability to ATC systems is considered.

CARDPB	Ref 2.4.1.1	
RSRE	Task 900	
CAA Sponsor:	Director of Projects and Engineering DPE(R&DP)	
CAA Liaison:	S C Willmott	Hd DP1a
	A.Price	DP1a
	C.Metcalfe	RD1

Contents

1 INTRODUCTION

Modern Air Traffic Control (ATC) systems are dependent on highly reliable computer based systems. This dependency is increasing with the growing volume of air traffic, making the control task after a system failure more difficult and prone to safety problems. Somewhat paradoxically, the addition of extra functionality to help air traffic Controllers with increased traffic implies even more reliance on the ATC computer system and, in turn, creates the need for even higher reliability and availability on new and evolving systems.

The Civil Aviation Authority (CAA) is aware of these problems and to help it keep abreast of current techniques and methods, and assess their relevance to current and future projects and plans, it sponsors a research and development programme in various software engineering topics. Part of this programme is carried out in the ATC Systems Research Division at RSRE Malvern, and one item of the Division's work has been an investigation into software fault tolerance as a means of achieving high reliability.

Fault tolerance is about making use of component or information redundancy and is one of two approaches to achieving reliability in systems. The other approach is fault prevention which is concerned with using methods, techniques and technologies that aim to avoid introducing faults into the implementation, including removal of faults found during testing. Specifically software fault tolerance is about using software redundancy to minimise the effects of software faults on the operational system, however the techniques used may, in some instances, provide some protection against hardware faults.

The RSRE research programme on software fault tolerance was mapped out to follow two lines of research:-

1. Study actual applications and existing research,
2. Study the applicability to Air Traffic Control (ATC) systems.

This report provides a summary of the studies into the research and application of software fault tolerance and considers the implication for ATC systems. The results of a study contract, initiated as part of the programme on the applicability of a specific software fault tolerant design technique to an ATC system, are given in references [24,29].

1.1 Structure of Report

Chapter 2 provides an introduction to software fault tolerance, setting out the main principles and terminology. The two basic schemes introduced in this chapter, N-version programming (NVP) and recovery blocks (RB), are described in more depth in chapter 3. A description of the recent research and the results of several of the leading research projects are presented in chapter 4 .This is followed by an account of some important applications in chapter 5, and a brief discussion on the important subject of cost-effectiveness in chapter 6 . The final chapter gives a quick summary of the status of software fault tolerance, and highlights some of the key issues relevant to ATC systems.

2 OVERVIEW OF SOFTWARE FAULT TOLERANCE

2.1 Introduction

Fault tolerance and fault prevention are two complementary approaches to achieving reliability in systems:

Fault prevention is concerned with using methods, techniques and technologies that aim to avoid introducing faults into the implementation. It involves two aspects referred to as fault avoidance and fault removal. Fault avoidance tries to exclude faults by use of the appropriate design and construction methods; examples of its use are the selection of reliable components and the adoption of good design methods. Fault removal attempts to locate and remove as many faults as possible by extensive testing, validation and verification.

Fault prevention may not provide the required reliability and it may be necessary to construct systems, usually involving redundant components, that tolerate faults and hence prevent system failure. Fault tolerance schemes require a combination of some or all of the activities of error detection, damage confinement, error recovery, and fault removal to reach the required reliability. Such schemes are well established in hardware systems where the physical nature and failure statistics of transients and component ageing are sufficiently understood that reliability predictions can usually be made with some confidence. In contrast, a software fault is either present or not present. Thus software does not have transients or wear out in the hardware sense; however, software faults may manifest themselves as errors in a computer system in such a way that the errors have characteristics similar to hardware failures (for example [12] describes an ageing characteristic due to maintenance). Thus software fault tolerance is about using design redundancy to minimise the effects of software faults.

The remainder of this chapter gives a brief description of the principal features of fault tolerance and the two main schemes for software fault tolerance: Chapter 3 give more details on on these two schemes.

2.2 Principles

Further discussion of software fault tolerance needs to be based on a coherent model of computer system failure. The model commonly used [24,26,22,1] is one in which a computer system is considered to consist of a set of hardware and software components interacting under the control of a design. A component of a system is also a system , so that a hierarchical decomposition of a computer system can be performed until all components are 'atomic', i.e. the internal structure of the component is no longer of interest. and can be ignored. In operation a computer system will move through a sequence of internal states as represented by the values of such items as bus-voltage levels or data base variables. During normal operation a physical component fault may occur, or a residual hardware or software fault may be encountered. This may lead to an error in the system state which in turn can result in a system deviating from its specification which causes a system failure. This chain of causality is depicted in the diagram below.

```
Physical          Hardware          Software
component          design            design
  fault             fault             fault
     \                |                 /
      \               |                /
       \              |               /
           error in system state
                      |
                      |
              failure in system
```

SYSTEM FAILURE MODEL

The model considers all software errors to be design faults. The reasoning, reference [1]pp57 behind this is that the failure of a component is attributable either to a design fault in the component or to a failure of a sub-component. The same reasoning can be applied to sub-component failure until eventually the component failure is traced either to a design fault, or to the selection of a faulty sub-component which itself is considered a design fault. The model also begs the issue of obtaining an authoritative specification by which a deficiency can be judged to be a system error or not, and for this report the existence of such a specification is assumed.

If fault tolerance is to be effective it will be necessary to construct the hardware and software system in such a way that a fault is prevented from causing a system failure. To do this, there will usually be some form of error detection, damage assessment and containment, error recovery and return to system service. It will also be desirable to attempt to identify the fault, remove it, and return the corrected component to the operational system.

2.3 Conceptual model

At a conceptual level the basic structure for software fault tolerance is shown in the diagram below in which the modules are different implementations of the same specification. The objective is to produce modules that do not fail on the same set of input data through a fault common to the different modules. This failure independency is difficult to ensure and the use of diverse development, for example separate teams with different design methods and different languages, is adopted in an attempt to achieve it. The adjudicator makes decisions about the results obtained from execution of the modules.

```
                 ----------
                 |        |
              -->|(primary) |____
             /   | module 0 |    \
            /     ----------      \
           /                       \         ------------
          /            "            \        |          |
  -------->            "          -------------->| adjudicator|---->
          \            "            /        |          |
           \           "           /         ------------
            \     ----------      /
             \   |        |      /
              -->| module n |___/
                 |        |
                  ----------
```

SOFTWARE FAULT TOLERANCE SCHEMATIC

Probably the most obvious interpretation of this diagram is the one which is directly analogous to the hardware technique of N-modular redundancy (NMR) and is known as **N-version programming**. In this scheme the modules, usually referred to as versions, are executed concurrently and the adjudicator in its simplest form is a majority voter. As in the hardware analogue, the modules must be atomic in the sense they must not interact with one another. In terms of the failure model, error indication is provided by the voter, damage assessment is achieved by atomicity of the modules, error recovery by ignoring outputs from modules identified as faulty by the voter.

The other main interpretation is the **recovery block** scheme in which the modules, usually referred to as alternates, are executed serially, starting with the primary module, until a module passes the acceptance test that is embodied in the adjudicator. To ensure alternate modules can execute from the same consistent state as the primary module, the system state is stored on entry to a recovery block (i.e. before executing the primary module), and restored before an alternate module is executed. Error detection is provided by the acceptance test, damage assessment is achieved by backward recovery, error recovery is obtained by state restoration and execution of an alternate module.

A 'module' is frequently referred to as a 'version' or an 'alternate'.

2.3.1 Nesting and concurrent systems

Nesting of the basic fault-tolerance schemes can be applied to hierarchically decomposed sequential systems. However, in a system of concurrent interacting processes, recovery has to be coordinated because state restoration after a module failure must continue until a consistent state is available to all other processes that had been involved in interactions with the faulty module: in the limit this could be back to the initial system state.

3 IMPLEMENTATION ISSUES

The implementation issues for the two main software fault tolerance schemes are examined. The schemes are compared and other schemes are discussed briefly.

3.1 N-Version programming

Implementation of the N-version scheme requires an underlying machine which is responsible for:

1. invoking each of the versions with identical data sets,
2. ensuring the versions execute to completion or are timed-out,
3. invoking the adjudicator with the outputs from the versions,
4. responding to the results from the adjudicator and initiating the next cycle of fault-tolerant processing.

Independent execution of the versions may be ensured by using separate hardware processors for each version. This approach can result in a heavy demand for resources if N-version programs are nested within each other to any extent. It is also possible to perform pseudo parallel independent execution on a single processor. Versions can retain private data, such as local state variables, between successive calls and this in itself can increase diversity, although there is a danger of cumulative effects leading to versions being detected as faulty and hence decreasing the system reliability.

It is usually stated that the inputs to each version must be identical; however in some systems, for example one with sensors providing slowly changing data values, it may be possible to increase diversity by feeding data from adjacent time slots into the different program versions : careful design of the adjudicator will be needed.

In a real system it will be essential to provide a time-out mechanism to avoid having the adjudicator 'hung-up' for a version that is unable to complete in time, for example because it is in infinite loop. There must also be a means of informing the adjudicator when a version is unable to complete properly due to an exception internal to the version such as a divide by zero. The task of adjudicator is to identify a consensus on the outputs of the versions and the system designer must decide what actions are required if it is unable to do this. For some systems where input data is not critical and is refreshed frequently it may be possible to ignore the occasional consensus failure. Safety critical systems may prefer to revert to some form of manual control.

3.2 Recovery blocks

A single recovery block consists of a primary module (0th alternate), standby alternates (1 to n), and an acceptance test. Execution of a recovery block requires the following steps to be carried out:

1. establish recovery point - save state,
2. execute the primary module,
3. pass the outputs from the module to the adjudicator,
4. if acceptance test of adjudicator succeeds, discard recovery point and exit,
5. if acceptance test fails and a next alternate module is available then restore system state to that at the recovery point, execute alternate module and repeat from step 3.
6. no more alternates, fail.

Establishing a recovery point can be thought of as a note to inform the underlying operating system that there may be a requirement in the future that will need the system state to be restored to that which exists at this recovery point. Since the primary module is the only module that is always executed, the standby alternates should not retain data between calls of the recovery block.

Because the primary module is always executed, it is usually the module with the most desirable characteristics. For example in surveillance radar tracking it might be appropriate to use a full Kalman filter for the primary module, a simple alpha-beta filter for the first alternate and a dead reckoning algorithm for the third alternate. Alternate modules can also be independently designed versions produced from the same specification as occurs in N-version programming. Indeed it should be possible to design an adjudicator to effect the N-version fault tolerance scheme. Another, potentially very important, way of using recovery blocks is to let the alternate modules be earlier versions of a program and the primary module the newly released enhancement. This approach could enable automatic reversion to an older trusted version if the new release fails the acceptance test.

It is desirable to have timeout and exception mechanisms in operation so that the adjudicator can be invoked when a module cannot execute in its allotted time or cannot handle an exception internally. In such cases the action of the recovery block would be to execute the next alternate, if available, after restoring the state to the recovery point. As originally conceived, the acceptance test, as its name implies, was not intended to guarantee complete correctness, but was meant to check on the acceptability of the results produced by a module. However it is a critical component of a recovery block and must be extremely reliable if it is not to have a significant effect on the overall reliability of the recovery block scheme. Consequently consideration is being given to adopting a correctness test as one form of adjudicator. In view of the high reliability required for acceptance tests it is sensible to aim for programs that are short and simple, and that are smaller than the modules that are intended to be checked.

3.2.1 System structure

In sequential programs nesting of recovery blocks presents no difficulty: each alternate can itself be a recovery block if desired (example structures are given in

[26]). In a system of concurrent interacting processes, recovery presents a problem because recovery of a process will normally include recovery of its interprocess communication data which in turn may force processes that have used or are using this data to recover. This backward recovery will need to continue until a consistent state is reached, and unless recovery is coordinated properly, a "domino" effect is likely and extensive restoration, even back to the initial system state, may occur. Particularly in a real time system such uncoordinated recovery is unacceptable.

The general solution to the domino effect is to establish suitable sets of recovery points so that if one of the interacting process raises an exception, backward error recovery can be provided by state restoration back to the appropriate consistent set of recovery points. The time sequence diagram below illustrates the concept for three processes, P1, P2 and P3. Intercommunication is shown by vertical lines and recovery points are labelled with unprimed lower case letters. Priming is used to indicate discarding of recovery points.

```
          e                    a            a'
----------+--------------------+------------+----------> P1
              |      |               |  |
              |      |               |  |
        d     |      |            b  |  |   b'
--------+-------------------------+---------+----------> P2
                            |
                            |
               c            |
---------------+---------------------------------------> P3


                          time  -->                    T
```

RECOVERY LINES and RESTORABLE ATOMIC ACTIONS

At time T a failure of process P1 or P2 will require restoration to the recovery points (c,d,e): note recovery points (a,b) have been discarded. Recovery of process P3 is required in the restoration because there is an interaction between P2 and P3 since the last available recovery point. The line joining points (c,d,e) is called a recovery line.

To minimise rollback, it is apparent that a recovery line should contain as few recovery points as possible, and these should have been established as recently as possible. In order to determine a recovery line which at least approximates to these conditions, two approaches are possible - planned and unplanned. The unplanned approach requires the recovery mechanisms to monitor interprocess communications and maintain a record of information flow so that a search for a recovery line can be made when a process raises an exception. A planned approach requires the system designer to use his knowledge about the particular system to place recovery lines.

The attraction of the unplanned approach is that no restrictions are placed on the processes. However, a complex recovery mechanism is required and there is a risk of severe loss of system facilities during recovery which is beyond the control of the designer. For these reasons the planned approach is preferred.

Of the possible strategies [1] for planned recovery those based on atomic actions are appropriate for structuring system activity. In this context a restorable (atomic) action is defined as providing a recovery line for a set of processes which are communicating only among themselves. Processes leave the restorable action simultaneously and the recovery line is discarded. An example of a restorable action containing processes P1 and P2 is depicted by a closed line through the points (a,b,a',b').

When the restorable action contains a recovery block spanning two or more mutually interacting processes it is called a 'conversation' [26].

The concepts of backward recovery in a system of intercommunicating processes have close analagies with the recovery of transactions in data systems [11]. A scheme for providing coordinated recovery in a MASCOT [20] based system has been developed [2,25] in which a common recovery point is provided for predefined subsets of concurrent processes (activities) and their intercommunicating data areas (IDA). These predefined subsets of activities and IDAs are called 'dialogues' and are essentially a MASCOT specific version of the generalized 'conversation' principle proposed in [26]. Thus activities executing within a dialogue are constrained to access only those IDAs which are associated with that particular dialogue.

In order to support the software fault tolerance scheme described above, the MASCOT (virtual) machine must be extended to provide recovery blocks and MASCOT construction and run-time facilities for dialogues [25].

Forward recovery is a means of providing a consistent state without state restoration. In particular, forward recovery mechanisms will be needed at the interface between recoverable and non-recoverable subsystems [1,2]. The form of these mechanisms is very much dependent on the application and can be difficult to design. Often the interfaces involved are essentially message passing systems with only a single activity accessing each side of the interface channel, and in such cases, simply clearing the channel may be sufficient for forward recovery. Some research work on forward recovery of failed versions in N-version programming is reported in [33].

3.3 Comparison of NVP AND RB

N-version programming is generally the preferred scheme in situations where replicated hardware can be used for concurrent execution of versions and voting schemes can be easily implemented. Voting can be problematic where real numbers, derived from different algorithms, are to be compared, and may be impossible in the cases where comparison of multiple correct solutions would be needed. Examples of such difficulties are given in chapter 7. N-version programming is also appropriate where versions need to retain data between calls. Where hardware is limited and voting checks are inappropriate then the recovery block scheme is indicated. In particular, the recovery block approach seems more suitable where alternate versions represent decreasing complexity and degraded functionality: n-version programming schemes tend to assume identical or near identical functionality of the versions in order to simplify voting. On performance grounds the recovery block scheme seems to be at a disadvantage because of the overheads that would be involved in using normal computer techniques to implement the backward recovery mechanisms. However, as pointed out in 4.4, there are practical hardware based solutions that appear to overcome the problem. Another less obvious problem with recovery blocks is that suitable acceptance tests are difficult to derive.

3.4 Other Schemes

Laprie proposes a scheme which he calls N **self-checking programming** [17]. The scheme is about executing 'hot' spares in parallel. Each spare is a self-checking component, and each component is either a version with an acceptance test or two versions with a comparator test. Fault tolerance for the failed service-providing component is obtained by switching to a component that has passed its test. Such a scheme corresponds closely to those adopted in the Airbus and Boeing flight control computers discussed later.

A mixture of the N-version and the recovery blocks ideas is proposed in [28] called **consensus recovery**. The consensus recovery scheme can be considered to be N-version programming which is followed by a recovery block if there is no agreement in the N-version voter. More precisely, all versions execute and submit their outputs to a voting procedure. If there is agreement the output is accepted; if there is no agreement then the outputs of the versions are examined in turn by an acceptance test until one output is accepted. Note that although there is no requirement for state recovery since all versions execute in parallel, there is a requirement to store all outputs.

3.5 Development of Independent Versions

The success of software fault tolerance depends upon the ability to provide versions of a program which are functionally equivalent but do not contain common design faults. More precisely, the different versions should fail independently, and the occurence of correlated failures should be sufficiently low that the reliability gain offered by software fault tolerance is not significantly impaired. The accepted methods for producing independent versions are outlined in this section, and the extent to which this can be achieved is discussed further in chapters 4 and 6.

The production of independent versions starts with a Customer requirement specification. Independent development teams, preferably using different development methods and tools, and aiming possibly for different target computers, work from this common specification. A monitoring authority is desirable to add impartiality, and to check and encourage the use of different development techniques; for example to ensure the use of different algorithms where possible.

The software produced by each team must be tested before release. Various testing techniques are possible [31]. Those relying on a knowledge of the internal structure of the software are referred to as "white-box" testing, and include generation and application of test cases, static analysis, manual inspection such as structured walkthroughs, and mathematical proof of correctness. Testing without internal knowledge of the internal structure is referred to as "black-box" testing, and includes the use of randomly generated inputs, and inputs simulated as representative of the intended user environment.

Use of test teams, separate from the development teams, is often advocated since they provide a different view of the problem and are more likely to detect faults. On the other hand, there is an argument that the original developer knows more about the design and structure of the software and is in a better position to design effective tests. No clear cut advantage of either approach has been demonstrated.

Once an an acceptable level of debugging has been achieved, the versions can be submitted to a customer acceptance test. Versions passing this test are compared in an N-version programming scheme with a comparator which is looking for any difference in the outputs for each set of common input data. This procedure compares each version against every other version and is commonly called "back-to-back" testing. Any discrepancies must be investigated and, if necessary, returned for repair and re-submitted to the testing cycle.

4 RESEARCH

4.1 Introduction

Research relating to software fault tolerance is carried out in several universities including Newcastle in the UK, and the Universities of California (UCLA), Illinois, North Carolina State and Virginia in the USA. NASA has supported much of the research. A number of industrial companies have also been active and more recently the electric and nuclear establishments have been making contributions. For convenience the research is described under the headings of reliability modelling, empirical evaluations of diverse software, and system simulations. While it is clearly not possible to discuss all the research work it is believed the main results and trends are represented.

4.2 Reliability Modelling

Important yardsticks for the credibility of software fault tolerance techniques are the predicted and measured increase in reliability. These quantities have been studied using statistical modelling techniques supported with some limited experimental verification. The application of the models require some means of assigning probabilities of failures of the various software components used in the particular fault tolerant scheme under consideration.

A naive model for 3-version programming system with a majority voter demonstrates the principles. Here an execution of the system can be considered to fail when one of the three error conditions hold:

1. all three outputs disagree
2. two outputs agree but they are incorrect
3. the voter is in error.

Assume for simplicity that the voter is never in error and simultaneous incorrect outputs are never equal, then the probability of the latter two error conditions occuring is zero. The overall probability is then the probability of at least two versions containing an error, and assuming the versions fail independently this is

$$p1.p2.p3 + (1 - p1).p2.p3 + p1.(1 - p2).p3 + p1.p2.(1 - p3)$$

where p1,p2,p3 are the individual failure probabilities of the three versions. If it is further assumed that individual probabilities are equal to p then the expression for the 3-version error probability reduces to one familiar in hardware triple modular redundancy calculations

$$3.p^2 - 2.p^3$$

It is instructive to compare the reliability of a single execution of the 3-version system with that of a single component, i.e.

$$1 - (3.p^2 - 2.p^3)$$

with

$$1 - p$$

.

For unreliable components with values of p greater than 0.5 the reliability of a single component is better than the 3-version programming system. This is a somewhat unfair comparison since all errors would be flagged by the voter in the 3-version programming, and in effect any results passed by the voter are correct: the fault tolerance has a 100% coverage.

Recovery block modelling tends to be more complicated than that for n-version programming because the acceptance test will be more complex than a straight voter and because the recovery mechanism may not be immune to errors. A simple reliability model for a two versions (a primary and one alternate module) recovery block system involves the following error possibilities:

1. the acceptance test accepts incorrect results from the primary
2. the acceptance test rejects the correct results from the alternate
3. unsuccessful recovery
4. the acceptance test accepts incorrect results from the alternate.
5. the acceptance test correctly rejects incorrect results from the alternate.

As in the preceeding example it is assumed that versions are independent and have a probability 'p' of incorrect execution. Further simplifications are possible by assuming recovery is perfect and that the probability of accepting an incorrect result is the same as the probability of rejecting a correct result, namely 'q'. The overall reliability can be shown [28] to be:

$$1 - (p + q)(p + q - 3pq) - pq(1 + 2pq)$$

.

For a perfect acceptance test (q=0), the reliability is $1 - p^2$ which is better than the simple 3-version result given above for all values of p. However for a poor acceptance test, say q=0.5, then reliability is much worse than the 3-version results (except for very unreliable versions with p greater than about 0.7). For totally correct versions (p=0), the reliability is $1 - q^2$ which shows the importance of having reliable acceptance tests (q small). If highly reliable recovery block software is required then the criteria of 'acceptability' rather than 'correctness' of test, as emphasised in early research work, is in doubt. Similar concern about the quality of acceptance tests has been expressed in a more detailed analysis involving nested recovery blocks [22].

The simple reliability models described can be extended to include more realism, particularly in regard to removing the assumption about independency between versions. Scott [28] gives reliability formulae for models of differing complexity not only for recovery blocks and N-version programming but for the mixed scheme called consensus recovery block (the name is not really appropriate since no state recovery is involved). The reference also reports an experiment to validate the models. In the experiment, the reliability of 16 versions of a parcel delivery management system, independently programmed in Pascal by computer science undergraduates, was assessed using 50 test inputs. The programs were then arranged randomly into 50 groups, with three different versions in each group. Each group was considered as a fault tolerant system whose reliability could be predicted using the assessed reliability of the individual programs. The predicted reliabilities were compared with the observed reliabilities obtained by applying a different set of 50 tests inputs to each fault tolerant group.

The resulting statistical analysis validated those recovery block and consensus recovery block models that allowed for failure dependence of versions. The dependence appeared to be connected with complexity - "if one program found a particular test case difficult and gave the wrong answer, then the probability was a nonzero that other programs also had trouble finding a correct answer, even though the programs used different algorithms". None of the models for N-version programming were satisfactory because it was possible to have different correct answers to a given set of inputs. In these situations simple majority voting is not appropriate since the reliability of the fault tolerant system is likely to be less than that of a single version.

Dependence in program versions leads to common mode errors and reduces the reliability that a fault tolerant system would otherwise have.

4.3 Empirical evaluations

Independent failure of program versions is crucial if software fault tolerance is to be effective. There have been a few serious attempts to develop and measure failure probabilities of diverse software.

4.3.1 Multi-version Launch Interceptor Program

The best known experiments [15] are probably those based on a Launch Interceptor Program (LIP). This program receives radar data and decides whether an object is a threat and hence requests for an interceptor launch. The program originated in TRW company study, passed through the Boeing company to the Research Triangle Institute (RTI), reference [8], and then to the Universities of Virginia and California. The specification from RTI was rewritten to remove ambiguities and other problems that had been previously encountered. Twenty seven versions (averaging about 1k lines of code) were developed by students working independently using one of two Pascal compilers. Each program was tested by the students and subjected to independent acceptance tests to ensure high reliability before being used in the N-version experiment.

The failure probability of each program was measured by executing each program with one million randomly generated test cases and counting the number of executions where the outputs were different from those of a "gold" version of the interceptor program that had been subjected to extensive analysis and testing. While the number of errors found in each version varied widely from 0 to 9656, i.e. error probabilities from zero to 0.009656, the errors in all versions could be traced to only 45 faults. The average error probability was 0.000698.

Independency of failure was checked by counting, for each test case, how many versions failed. This enabled the measured probability of more than one version failing to be compared with that predicted on the basis of independent failure (using the measured failure probabilities of the individual versions). A statistical test clearly rejected the hypothesis that (these) programs fail independently.

To see what effect this failure dependency would have on N-version systems, further trials were set up to measure the failure probabilities of 2-version and 3-version programs using the 27 versions already developed [15]. In the 2-version experiment all 351 combinations of 2 different versions were each subjected to a million tests and a failure was counted if the output of both versions in a combination did not agree with the output of the "gold" version. The average failure probability of all 351 systems was 0.001384 compared with the average of 0.000698 for individual programs. This has to be interpreted with care because in an actual use, those error situations where the outputs from the two versions differ would be detected whereas of course there is no error detection as such in executing a single version. In this experiment it was found more 99% of errors were detected so the average probability of an error, ignoring the detected errors, for a 2-version system is about 0.000014.

A similar experiment was carried out for 3-version programming using all 2925 combinations of 3 different versions with failures being counted if 2 or three versions disagreed with the "gold" standard. The average failure probability was found to be 0.0000367 which is three orders of magnitude greater than would have been expected had the individual programs failed independently. Error detection was relatively poor in that only 65% were detected, 35% having 2 or more versions agreeing on the wrong result.

Further experiments to investigate adding acceptance tests to 8 of 27 of the launch interceptor programs are reported in [7]. Eight groups (3 per group) of students, working independently, were given a week to add error detection software to one of the program versions. The types of tests added were mainly duplication of functionality using mainly different algorithms from the original code, structural checks such as verifying the correct use of data structures, reversal checks where the "inputs" were produced from the outputs by reversing the operations involved, and consistency checks such as range checking (chapter 5 of [1]). The first series of tests were based purely on the LIP specification. The students were then required to read the program code and embed test software; they were also asked to note faults detected during the reading. Application of 200 randomly generated test cases resulted in just over half of the 26 known faults being detected. 20% were found by the specification based tests, 40% by code reading and 40% by the code based test. Six of the detected faults had not been previously known. Great differences in individual ability to design effective test programs was noted.

4.3.2 The PODS experiment

The European nuclear industry have shown an interest in the potential of diverse software in a collaborative project [5] called PODS (Project on diverse software) to provide insight into questions concerning the cost effectiveness of diverse software. The participants in the project were the Safety and Reliability Directorate (SRD) and the Central Electricity Research Laboratory (CERL) in England, the Technical Research Centre of Finland (VTT), and the Halden Reactor Project (HRP) in Norway. SRD acted as the customer, producing a customer specification for a reactor over-power (trip) system. The other bodies took on the role of manufacturers, independently implementing their own version of software to meet the customer specification.

The CERL team worked informally on the specification, while other development teams translated the specification into a formal specification language (X-SPEC). The UK and Finnish teams coded in FORTRAN77 while the Norwegian used assembly.

All three versions were submitted to a common set of acceptance tests consisting of systematic and random tests. After acceptance testing, extensive back-to-back testing was used to locate residual faults. In this testing, the outputs of the programs were compared for some 600k test cases and any disagreements used to initiate fault diagnosis. Faults were removed and the offending program returned to the back-to-back testing provided the acceptance tests were passed.

Only seven faults were discovered during back-to-back testing; six were related to ambiguous or incorrect customer specification and one to the formal specification. Two of the faults were found to result in common mode errors in the HRP and VTT versions. No faults were traced to the implementation phase, apart from two faults introduced when making corrections for the specification related errors.

The cost of a single development was nearly 1000 hours compared with the threefold diversity cost of nearly 2300 - this excluded the back-to-back testing since that was considered to be part of the operational usage.

The three programs have been subjected to further analysis and test methods in a follow up project [6] called STEM (Software Test and Evaluation Methodologies). Of particular relevance was the analysis of failure independence of particular versions of the CERL and VTT programs. These were known to contain 15 and 13 faults respectively, i.e. 195 fault pairs to consider. It was found about 80% of the fault pairs gave rise to independent or nearly independent failures. Nearly 15% resulted in simultaneous failures, and 5% never gave rise to simultaneous failure.

4.4 System Simulation

The experimental work discussed above has been largely concerned with the failure rates of independently developed programs particularly in connection with N-version programming. A more system orientated project [2,3] was set up at Newcastle University in 1981 to evaluate the cost-effectiveness of the recovery blocks scheme by building a naval command and control simulator. The command and control function took its input from simulated radar, sonar and inertial navigation systems. An operator interacted with the generated labelled radar display which enabled him to control helicopter attacks on a hostile submarine. The command and control function consisted of about 8000 lines of Coral code, structured into 14 concurrent activities.

The recovery block structure used was the 'dialogue' form in a MASCOT environment. The MASCOT executive which supports and controls pseudo concurrent processes and their interactions was modified and extended to provide recovery block facilities. The recovery mechanisms utilized a specially designed hardware recovery cache to enable state storage and restoration to be be performed rapidly.

Three main phases of experiment took place with 60 runs in each phase. Each run was automatically monitored and observed by an operator and each time an error was detected, an attempt was made to identify the fault which caused the error. The run would continue until the scenario was completed or a failure prevented the run from continuing. Two problems arose in the experiment, namely the unrepeatability of the runs and the prototypical nature of the recovery mechanisms. By the very nature of the project little could done about the first problem, but recovery mechanisms were improved for phase two and further for phase three. Phase one seemed to be very much an exploratory phase, verifying the recovery mechanisms and providing initial failure measurements. Phase two used the same command and control software as phase one with corrected recovery software. In phase three some of alternates in the recovery blocks were replaced by versions written by inexperienced programmers.

The principal measure of effectiveness of software fault tolerance was failure coverage, i.e. the ratio of successful failure recoveries to potential failures (successful + unsuccessful) recoveries. This came out to be 0.68, 0.53 and 0.81 for the three phases, giving an average of about 67%. A slightly higher average of 72% is obtained if failures due to imperfect recovery are ignored.

The mean time between failures, is quoted as 0.74hr for the fault-tolerant system against the 0.31hr for the nonfault-tolerant system. However, as pointed out by the researchers, these figures are on the low side because the simulations were run in "fast" mode during periods of relative inactivity.

It was estimated that an extra 60% of application software was needed for the acceptance tests and alternate modules. System overheads were measured as 33% extra code, 35% extra memory, and 40% additional run-time.

The similarities of a naval command and control system to a computerized ATC system stimulated an interest in the possibility of using software fault tolerance techniques for ATC. This led to the ATC Systems Research Division at RSRE initiating a study [29] on the applicability of recovery blocks to ATC with former participants in the Newcastle simulation: a contract being formally placed by CAA with RMCS who subcontracted some work to MARI. This study [24] produced a high level MASCOT-based design of the tracking and flight plan processing parts of London Air Traffic Control Centre (LATCC), and a detailed design of the tracking system with the fault tolerant features of the dialogue recovery blocks. The increase in size of the application pseudo-code for the fault tolerance design was about 40% although, as with Newcastle work, this is very dependent on the number and sizes of the alternates and the acceptance tests. Some minor refinements to the dialogue scheme were made during the study.

It was not possible within the scope of the study to address cost-effectiveness and to help provide such information a proposal to build a large demonstrator with hardware support for recovery has been made to Esprit by a consortium being led by MARI.

4.5 Research facilities

A number of experimental facilities to support the evaluation of software fault tolerance have been developed. One is the NASA Langley Research Center's AIRLAB facility in which developing assessment and validation techniques for fault-tolerant systems is a major objective. Another is a testbed for software DEsign DIversity eXperiments, DEDIX, at UCLA [4,34]), support by FAA with a NASA contract. Reference [38] reports a FORTRAN 77 package that implements advanced reliability modelling techniques called HARP (Hybrid Automated Reliability Predictor). This is sponsored by NASA and is under development at Duke University. It is being used to analyse both hardware and software fault-tolerant computing systems [30].

5 APPLICATIONS in Operational Systems

The number of systems reported to be making use of diverse software for fault tolerance in a major way is rather small. The A310,A320 aircraft are the best known examples, with the Boeing 757/767 aircraft, the Titan III launcher ([12] refers to a backup module for attitude control being incorporated) and Space Shuttle also frequently referred to in the literature. The use of diverse software in the Swedish state railways and in a computerized reactor safety shut down system of a nuclear power generator currently under construction has been reported [34]. The aerospace applications are described below.

5.1 A310 Airbus

The A310 aircraft uses dissimilar software in the control of flaps and slats [14,21,36]. In this system the flaps and slats control consists of 2 computers and two sets of hydraulic motors. Each computer consists of two halves - one for flaps and one for slats. Each half-computer controls its own dedicated hydraulic motor which drives the flaps (or slats) on both wings. Inputs to the computer are from pilot's selector and from position sensors on the control surfaces. Within each half-computer, dual-lane dissimilar processing is employed to give the system very high integrity.

In order to make the two lanes as independent as possible, different microprocessors (an Intel 8085 and a Motorola 6800) are used with the software being produced by seperate design teams. The outputs from the two lanes are compared by a hardware AND function before being fed to the motor. The software in each lane also monitors the other lane and in case of failure or disagreement brakes are applied to freeze the surfaces in their current position.

The design aims for the high integrity requirement and the less stringent availability requirement are:

Inadvertent deployment of surfaces, Asymmetric deployment of surfaces, Slats or flaps no longer operating and no warning given to the crew.	$< 10^{-9}$ per hour
Surfaces not operable when commanded but failure indicated.	$< 10^{-5}$ per hour

Note that the aircraft can be flown and landed with the surfaces frozen; but runaway movement of the surfaces, failure with no indication to the pilot, or asymmetric deployment are dangerous.

The integrity depends on the independence of the software designs and considerable management effort was expended on ensuring this. The design teams were kept separate and even used different host computers. A central committee dealt with problems of interpretation of the (common) requirement specification, and monitored the development to ensure that the teams employed different solutions.

Claimed advantages of dissimilarity include clarification of the specification, ease of getting certification acceptance, and less need for module testing because the dual-lane real-time tests provided a stringent testbed. More software was produced and high order programming languages could be used without undue concern about compiler errors. Overall, therefore, it was thought there was no increase in development cost in adopting the dissimilar software approach.

The technique used does reduce the availability of the system. In this application this does not matter but in future applications where high availability is required then the full fault tolerant techniques will need to be used. Certification was achieved in March 1983 and since then there has been only one revision of the software (as of Jan 1988). No erroneous deployment of surfaces had been reported in 750,000 flying hours up to May 1985 [36].

5.2 A320 Airbus

The diverse software in the A310 is primarily aimed at fault detection with recovery being achieved by manual intervention, either to the standby channel, or to manual control. The design objective in the A320 is that the computer based control system should be sufficiently reliable that the reversion to the limited mechanical backup should be unecessary. The limited mechanical backup is intended to increase confidence and ease certification [27].

Five computers are used in the control of the roll axis and four of these are also used for control of the pitch axis. Each computer employs a duplex processor configuration with 2-version software and includes watchdog timers. Two types of computer are used, one based on 68000 microprocessors and the other on 80186 microprocessors. This gives rise to four different versions of control software.

The computers are arranged into two subsystems, one for the "pilot side" and one for "co-pilot side". For example, the pitch control of the pilot side consists of one 68000 and one 80186 based computer. At any one time, only one computer will be in control with the other in hot standby. It is claimed that the control system can tolerate complete loss of one side of the control system, and at least one software fault that leads to shut down of one type of computer.

In meeting the reliability requirement failure rate of 10^{-9} per hour for the computer control system, software failures are excluded from the calculations. An airworthiness certificate was obtained recently in France in February 1988.

5.3 Boeing 757/767 and 737 Aircraft

The Boeing 757/767 is equipped with yaw damping that makes use of 2-version software ([37] and page 91 of [34]). The Advanced autopilot flight director system computer architecture for Boeing 737-300 aircraft has also been reported to employ diverse software but the extent, if any, to which this used to detect design errors is not clear from the available reference [35].

5.4 First shuttle orbital flight delay

The flight of the shuttle in 1981 was stopped 20min before scheduled launch because of a synchronisation failure between the backup computer and the main computing system [10]. The main system consisted of 4 identical computers in hardware, and software during critical phases. Four were chosen since the failure of one would still enable effective majority voting with the remaining three. The backup system consisted of one computer, the same as those used in the main system, but with software developed by a separate team and structured as synchronous processes as opposed to the asynchronous approach used in the main system. When not in control the backup system listens to the input/output data of the main system not only for checking purposes but to keep itself in a state of "readiness". The software fault tolerance detection is embodied in the backup system, and in this instance can be said to have succeeded. The fault was traced to a problem in the main system software concerned with using different 'clocks' at initialisation, resulting in the backup system noting data 'fetches' at an unexpected time cycle.

6 COST-EFFECTIVENESS

The cost effectiveness of software fault tolerance has to some extent to be judged on the results of the experiments, modelling and applications discussed briefly above. Scott et al [28] have used the cost model

$$Cost = A.(1 - R)^{-B}$$

where, R is the required reliability $(0 < R < 1)$, B is a constant for given development process, and A is a constant relating to the complexity of the software, in the context of their experiments on software failure rates (4.2). The model is used to show for low programming costs it is more effective to spend more effort on increasing the reliability of a single version than to use multiple versions in a fault tolerance scheme. However as programming costs increase the model indicates fault tolerant schemes become cheaper: Consensus recovery being cheaper than recovery blocks which in turn is cheaper than N-version programming.

The recovery block scheme is more sensitive than the consensus scheme to the reliability of the acceptance tests.

Migneault [23] has put forward a model to relate the amount of testing and debugging of a software component in removing faults to meet given reliability requirements.This is then used to relate costs to the overall system reliability for N-version programming. For a single component the cost of increasing the reliability by a factor of 10 can result in a 100 times increase in the testing and debugging, thus indicating that a fault tolerance scheme might be a more cost effective solution.

The Newcastle recovery block simulations provide one of the few system measurements in that a 60% increase in application code was required to raise the MTBF from 0.31hr to 0.74 hr, an increase of only 2.4. Excluding failures due to imperfect recovery, a 9 times improvement in reliability would have been obtained. An increase in application code of about 30% was required in the RMCS study on applying the Newcastle based techniques to the London ATC system.

Of particular interest to the aircraft certification authorities is the proposed use of N-version software back-to-back testing in order to reduce the otherwise considerable costs that would be needed for structural testing ("white box") for Category III autopilot [18]. The reference has some severe criticisms particular concerned with the ability, or rather lack of it, for back-to-back testing in detecting common mode faults, noting that LIP experiments found common failures in up to eight independently developed programs. It also considers FAA document (RTCA/DO-178A) about the requirements for the development of airbourne-software is not rigorous enough for safety critical software.

7 APPLICATIONS in AIR TRAFFIC CONTROL SYSTEMS

The potential benefits of software fault tolerance to Air Traffic Control systems are increased operational reliability with the attendant improvements in integrity and safety. Some protection against the serious malfunctions associated with the introduction of new software releases is also possible.

Can such benefits be achieved in ATC systems and in a cost-effective manner? The answer to the first part of the question is a guarded "yes", in that experiments, such as the Naval command and control simulation described in the previous chapter, do indicate an an increased reliability. The answer to the second is "maybe". Certainly the airborne system such as that in A320 aircraft is presumably considered cost effective, but the cost of designing and building a commercial recovery blocks computer for software fault tolerance for a large ATC system is still an uncertainty and is a topic which the proposed ESPRIT work mentioned in chapter 8, if accepted, will address.

7.1 Recovery Blocks

Accepting that increased reliabilty is possible, it is important to have some indication of where in an ATC system software fault tolerance might be most effective. The study on the applicability of software fault tolerance to LATCC [24] used two guide lines in deciding where alternate software modules might be beneficial.

1. Provide alternate(s) for a module with complex processing.
2. Provide alternate(s) for processes involved in updating significantly large parts of the data base.

   Two further guidelines were used in determing what type of alternates should be employed.
3. Use full function (not degraded) alternates where high integrity data flow occurs.
4. Use degraded alternate where low integity data flow occurs.

Use of these guide lines in the LATCC study resulted in degraded alternates being provided for the complex tracking algorithms because radar target plots are continually being updated. Full function alternates were provided for updating the radar database because of the critical nature of the flight plan related parts. Although not investigated during the study, an analysis of the LATCC Flight Plan processing system using these guidelines would probably result in more full function alternates due high integrity required of the flight plan data.

As well as benefits to ATC system development, software fault tolerance could also aid software maintenance. For instance, the cost-effectiveness of software maintenance could be enhanced both by explicitly exploiting the mechanisms of the recovery block technique and implicitly through the strong structuring placed on the software design. New versions of software modules could be introduced as the primary alternates, with the older trusted versions acting as standby alternates. Error detection would result in data base recovery and the system would resume operation with the well tried software. The recovery mechanisms could also prove beneficial in protecting the operational system against design faults when adding new functions, for example by including watchdog timers and integrity checks on critical parts of the database in the acceptance tests.

The effectiveness of the recover blocks techniques in ATC system will be very dependent the designers ability to develop highly reliable acceptance tests used in the error detection preceding recovery. Such tests have been difficult to devise in both in the LATCC study and the Naval command and control simulation [3]. Many of the acceptance tests in the LATCC study were effectively data type checks and a language such as Ada with its exception handling would be of enormous benefit. However the conceptual simularity between a highly reliable acceptance test and the pre and post conditions of a formal specification suggests more fundamental approach would be through the use of the techniques employed in formal specification languages.

7.2 N-version

The discussion above has concentrated on the use of the recovery block technique. It is pertinent to ask about the use of other schemes such as n-version programming. For instance, it would seem possible to replace the the recovery blocks of the LATCC design with n-version programs, and a few preliminary thoughts are recorded here to indicate a few of the problems areas. Each recovery block could become an n-version program, with each alternate, including the primary, becoming a version and the acceptance test being replaced by a voter. Clearly, an additional version is likely to be required for each recovery block with only two alternates: majority voting on two versions is unsatisatisfactory since it provides only error detection and error recovery is needed to mask faults. Voting will frequently involve inexact comparisons; for example a target's position and velocity coordinates will be slightly different from one version of tracking to another. If one version employed a full Kalman filter and another relied on dead reckoning, as occurs in the recovery block LATCC design, then reconcilliation may be difficult. Even if reconcilliation is possible, the system database will only want one set of values. Will a straight averaging of the coordinates be sufficient, or will a weighted average be necessary, or will the resuts from one of the version be acceptable? A related problem is "Do versions maintain there own independent track tables?". Tnis covers but a few problem areas that could, with benefit, be addressed in a future study.

8 SUMMARY and CONCLUSIONS

Experiments have confirmed that software fault tolerance can significantly increase the reliability of software systems, but the gains will not be as large as those predicted on the assumption that the software components, developed independently to the same specification, fail independently. The research work suggests that an increase in reliability of least an order of magnitude should be attainable for perhaps less than double the software cost: a satisfactory relation between reliabilty and cost has yet to be established.

Software fault tolerance is considered to have little capability against requirement specification errors, yet interestingly, the residual faults detected in the back-to-back testing in the nuclear industry's PODS experiment [5] were all traced back to specification errors. It has also been argued that back-to-back testing reduces the debugging costs per software version sufficiently to make the use of dissimilar software in flight control systems a competitive solution. This argument has been followed up with one which seems to imply the use of diverse software should permit the safety-critical rating of software to be lowered [18]: this should be of great concern to the certification bodies.

While failure independence remains the major problem area in software fault tolerance there still remain difficulties with the voting system in N-version programming and the acceptance tests in the recovery block scheme. Voting systems are not always of the simple comparison type; for example they may have to compare for equality in data items whose values have only been calculated approximately, or they may have to handle multiple correct solutions such as in travelling salesman type problems. Acceptance tests in the recovery block systems present difficulties of concept since originally they were thought of as simple tests just to check that results were 'acceptable', but checking for 'correctness' and hence the functionality features in current thinking: many of the acceptance tests in the LATCC system design [24] were of the former kind which would be implicit in the type checking of an Ada-like language. One interesting route to devising 'correctness' tests is through the constructs of formal specification languages. It would be a valuable exercise to recast the LATCC recovery block design into one employing n-version programming, investigating the problems related to voting in an ATC application and assessing how n-version progamming should be structured in a large system.

Variations on the basic N-version programming have been applied successfully in operational systems, although the A320 is possibly the only system which will tolerate software faults without manual intervention. Application of recovery blocks awaits the development of a suitable reliable hardware supported recoverable machine. An ESPRIT proposal from MARI as prime contractor to build a recovery blocks demonstrator with ATC application has been made (April 88); the CAA would be one of the participants.

From a maintenance point of view, the recovery block scheme has the attractive feature that modified software could be introduced into an operational system with the knowledge that an acceptance test failure would result in recovery of the database and automatic resumption of service with an existing proven version. How such recovery operates in a distributed system needs investigating.

References

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

[2] T.Anderson, P.A.Barrett, D.N.Halliwell, and M.R.Moulding. *Software Fault Tolerance: An Evaluation.* IEEE Trans SE-11(12), Dec. 1985.

[3] T.Anderson, P.A.Barrett, D.N.Halliwell, and M.R.Moulding. *An evaluation of of software fault tolerance in a practical system.* Proc. 15th IEEE Symp. Fault-tolerant computing, pp.141-145, 1985.

[4] A.Avizienis, P.Gunningberg, J.P.J.Kelly, L.Strigini, P.J.Traverse, K.S.Tso, and U.Voges.*The UCLA DEDIX system: A distributed testbed for mutiple-version software.* Proc. 15th IEEE Symp. Fault-tolerant computing, pp.126-134, 1985.

[5] P.G.Bishop, D.G.Esp, M.Barnes, P.Humphreys, G.Dahll, J.Lahti. *PODS - An Experiment in Software Reliability.* IEEE Trans. Software Engineering, SE-12, No.9, September 1986.

[6] P.G.Bishop, D.G.Esp, F.D.Pullen, M.Barnes, P.Humphreys, G.Dahll, B.Bjarland, J.Lahti, and H.Valisuo. *STEM - A project on software test and evaluation methods.* pp 100-117 in 'Achieving safety and reliability with computer systems' the proceedings of the 1987 Safety and Reliability Symposium edited by B.K.Daniels, Elsevier Applied Science, 1987.

[7] S.D.Cha, N.G.Leveson, T.J.Shimwell and J.C.Knight. *An empirical study of software error detection using self-checks.* Proc. 17th IEEE Fault-tolerant computing, 1987, pp.156-161.

[8] J.R.Dunham. *Software errors in experimental systems having ultra-reliabilty requirements.* Proc. 16th IEEE Symp. Fault-tolerant computing, pp.158-164, 1986.

[9] D.E.Eckhardt and L.D.Lee. *A theoretical basis for the analysis of multi-version software subject to coincident errors.* IEEE Trans. Software Engineering, Vol SE-11, No.12, pp.1511-1517, 1985.

[10] J.R.Garman. *The "Bug" heard 'round the world.* ACM Software Engineering Notes, October 1981, Vol.6, No 5, pp3-10.

[11] J.Gray. *Notes on Data Base Operating Systems.* Lecture Notes in Computer Science, Vol.60 Operating Systems, Springer-Verlag, 1978.

[12] H.Hecht and M.Hecht. *Software reliability in the system context.* IEEE Trans. Software Engineering, SE-12, No.1, January 1986.

[13] H.Hecht. *Fault-tolerant software for real-time applications.* Computing Surveys, Dec. 1976, Vol.8, No.4, pp391-407.

[14] A.D.Hills. *Digital Fly by Wire.* AGARD Lecture Series No.143, 1985.

[15] J.C.Knight and N.G.Leveson. *An empirical study of failure probabilities in multi-version software.* Proc. 16th IEEE Symp. Fault-tolerant computing, pp.165-170, 1986.

[16] J.C.Knight, N.G.Leveson, and L.D.St.Jean. *A large scale experiment in N-version programming.* Proc. 15th IEEE Symp. Fault-tolerant computing, pp.135-139, 1985.

[17] J.Laprie, J.Arlat, C.Beones, K.Kanoun, and C.Hourtolle. *Hardware and Software Fault Tolerance: Definition and Analysis of Architectural solutions.* Proc. IEEE Fault-tolerant computing, 1987, pp.116-126.

[18] N.Leveson. *A scary tale - Sperry avionics module testing bites the dust?.* ACM Sigsoft, Software Engineering Notes, April 1987, Vol.12, No.2, pp23-25.

[19] B.Littlewood and D.R.Miller. *A conceptual model of a multi-version software.* Proc. 17th IEEE Fault-tolerant computing, pp.150-155, 1987.

[20] MASCOT Suppliers Association. *The Official Handbook of MASCOT.* RSRE, Malvern, UK, 1980.

[21] D.J.Martin. *Dissimilar Software in High Integrity Applications in Flight Controls.* Software for Avionics (AGARD Conf. Proc. 330), Jan. 1983, pp.36-1, 36-13.

[22] P.M.Melliar-Smith. *Development of Software Fault Tolerance Techniques.* NASA Contractor Report 172122, March 1983.

[23] G.E.Migneault. *The cost of software fault tolerance.* Software for Avionics (AGARD Conf. Proc. 330), Jan. 1983, pp 37-1 to 37-8.

[24] M.R.Moulding. *An Investigation into the Application of Software Fault Tolerance to Air Traffic Control Systems.* Project Final Report, RMCS Reference 1049/TD.6, July 1987.

[25] M.R.Moulding. *An Architecture to Support Software Fault Tolerance and an Evaluation of its Performance in a Command and Control Application.* Digest IEE Colloquium on Performance Measurement and Prediction, Feb. 1986.

[26] B.Randell. *System Structuring for Software Fault Tolerance.* IEEE Trans. SE-1 (2), pp. 220-232, 1975.

[27] J.C.Rouquet and P.J.Traverse. *Safe and Reliable Computing on Board the Airbus and ATR Aircraft.* Proc. 5th. Int. Workshop on Safety of Computer Systems (SAFECOMP), pp.93-97, 1986.

[28] R.K.Scott, J.W.Gault, and D.F.McAllister. *Fault-tolerant software reliability Modeling.* IEEE Trans. Software Engineering, Vol. SE-13, No.5, pp.582-592, 1987.

[29] L.N.Simcox. *The Application of Software Fault Tolerance to Air Traffic Control: Study Contract Overview.* RSRE Memorandum 4114, June 1988.

[30] T.B.Smith and J.H.Lala. *Development and evaluation of a fault-tolerant (FTMP) computer.* volume IV, FTMP executive summary, NASA CR-172286.

[31] I.Sommerville *Software Engineering.* 2nd edition, Addison-Wesley,1985.

[32] P.Traverse. *AIRBUS and ATR, System Architecture and Specification.* pp.95-104 of reference [34].

[33] K.S.Tso and A.Aviziens. *Community error recovery in N-version software : A design study with experimentation.* Proc. IEEE Fault-tolerant computing, 1987, pp.127-133.

[34] U.Voges (ed.). *Software diversity in computerized control systems.* Dependable computing and fault-tolerant systems Vol.2, Springer-Verlac Wien.

[35] J.F.Williams et al. *Advanced autopilot-flight director system computer architecture for Boeing 737-300 aircract.* in 5th Digital avionics systems conference, 1983.

[36] N.Wright. Alvey Software Reliability and Metrics Club meeting on Software Fault Tolerance, May 1985.

[37] L.J.Yount. *Generic fault-tolerance techniques for critical avionics systems.* In Proc. AIAA guidance and control conference, June 1985.

[38] *Special Issue on Fault-Tolerant Computing, IEEE Trans. Reliability.* June 1987, Vol.R-36, No.2. Although largely about hardware, the papers on reliability modelling methods and tools have some relevence to software fault tolerance.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ......UNCLASSIFIED...

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMO 4237	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE WR14 3PS		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location UK CIVIL AVIATION AUTHORITY		

7. Title SOFTWARE FAULT TOLERANCE
7a. Title in Foreign Language (in the case of translations)
7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials SIMCOX, L N	9(a) Author 2	9(b) Authors 3,4...	10. Date 1988.06	pp. ref. 29
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement
UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The main principles of software fault tolerance are presented and various schemes for achieving it are described, with emphasis being placed on the N-version programming and the recovery blocks schemes. The areas of research discussed include reliability modeling, version independence, and system simulation. A brief description of the use of software fault tolerance in aerospace applications is given and the important subject of cost-effectiveness is addressed. The potential benefits to Air Traffic Control systems are identified together with areas where further study and research are desirable.

S80/48