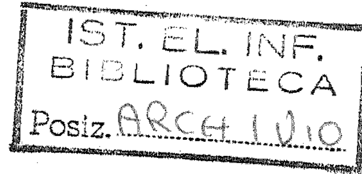
B4-71 (1990)

PREDICTABLY DEPENDABLE COMPUTING SYSTEMS
TECHNICAL REPORT PROJECT 3092 PDCS

PDCS No. 23 26 July 1990

SOFTWARE FAULT TOLERANCE

L. Strigini
IEI - CNR, Pisa


IST. EL. INF.
BIBLIOTECA
Posiz. ARCH I V.10 B4-71

Abstract

Software design faults are a cause of major concern, and their relative importance is growing as techniques for tolerating hardware faults gain wider acceptance. The application of fault-tolerance to design faults is both increasing, in particular in some life-critical applications, and controversial, due to the imperfect state of knowledge about it. This paper surveys the existing applications and research results, to help the reader form an initial picture of the existing possibilities, and discusses in an orderly fashion the design options available for using software fault-tolerance in a design. The decision to employ software fault tolerance, and in which form, cannot be based on a rigourous reliability evaluation, since both experimental data and proven models are lacking (this situation is common to other popular engineering techniques). On the other hand, some software fault tolerance techniques are intuitively attractive as a means for improving reliability, and experimental evidence, albeit limited, supports this idea. The specific combination of techniques and design options in any individual software system must be dictated by their suitability to the characteristics and requirements of the different parts of the system. The discussion provided here offers a general guide for these choices, with extensive references to the appropriate literature.

Release number	1
Release status	Final
Availability status	Public
Available from	Felicita Di Giandomenico, IEI-CNR, Via S. Maria 46, 56126 Pisa, Italy

This report was prepared as a deliverable for the first year of the PDCS project, May 1990. It is partially based on a previous survey prepared for the Esprit project "Delta-4", available as I.E.I. Report No. B4-53 1988, "State of the art in software fault tolerance and use of software fault-tolerance in Delta-4: preliminary report", by Lorenzo Strigini

CONTENTS

Introduction 1
1- MOTIVATIONS FOR SOFTWARE FAULT TOLERANCE, AND A BRIEF HISTORY 3
1.1- The problem of software dependability 3
1.2- The concept of software fault tolerance 3
1.3- A brief history - research 4
1.4- Industrial use 6
1.4.1- Fully redundant software in critical application 6
1.4.2- Other software fault tolerance techniques 7
1.5- Effectiveness of software fault tolerance 8
1.5.1- Single project experiments 8
1.5.2- Statistically oriented experiments 9
1.5.3- Analytical estimation 10
1.5.3.1 Generalities 10
1.5.3.2 Failure models 11
2- THE UTILIZATION OF SOFTWARE FAULT TOLERANCE 13
2.1- Fields of application for software fault tolerance. 13
2.2- Validity of software fault tolerance approach. 13
2.3- Applicable research results 14
2.4- Transparency of fault tolerance provisions 15
2.5- Assessment of reliability improvements 16
2.5.1- Limitations 16
2.5.2- Advantages of software fault tolerance for reliability estimation 16
3- DESIGN ISSUES FOR SOFTWARE FAULT TOLERANCE 17
3.1- Software fault tolerance versus hardware fault tolerance 17
3.2- General implications of the possibility of design faults. 18
3.3- 'Safety net' approaches 19
3.3.1- The approach in hardware fault tolerance 19
3.3.2- Applicability in software fault tolerance 19
3.3.3- Utility and limitations 19
3.4- The Tandem example 20
3.5- Structured software fault tolerance 20
3.5.1- Generalities 20
3.5.2- Error detection 21
3.5.3- Generalized consistency checks 22
3.5.4- Error correction 22
3.5.5- Distribution of executions 23
3.5.6- Execution of variants 23
3.5.7- Granularity of Fault Units 23
3.6- Hybrid software fault tolerance 24
3.7- Conceptual problems in structured software fault tolerance 25
3.7.1- The problem of inexact voting 25
3.7.2- The problem of state recovery of variants 26
CONCLUSIONS 27
ACKNOWLEDGMENTS 28
REFERENCES 28

SOFTWARE FAULT TOLERANCE

L. Strigini
IEI - CNR
Via Santa Maria, 46 -56126 Pisa, Italy

Abstract

Software design faults are a cause of major concern, and their relative importance is growing as techniques for tolerating hardware faults gain wider acceptance. The application of fault-tolerance to design faults is both increasing, in particular in some life-critical applications, and controversial, due to the imperfect state of knowledge about it. This paper surveys the existing applications and research results, to help the reader form an initial picture of the existing possibilities, and discusses in an orderly fashion the design options available for using software fault-tolerance in a design. The decision to employ software fault tolerance, and in which form, cannot be based on a rigourous reliability evaluation, since both experimental data and proven models are lacking (this situation is common to other popular engineering techniques). On the other hand, some software fault tolerance techniques are intuitively attractive as a means for improving reliability, and experimental evidence, albeit limited, supports this idea. The specific combination of techniques and design options in any individual software system must be dictated by their suitability to the characteristics and requirements of the different parts of the system. The discussion provided here offers a general guide for these choices, with extensive references to the appropriate literature.

INTRODUCTION

This paper attempts a survey of the state of the art in the techniques for tolerating the effects of design defects in software products.

Software design defects (or "design faults") are obviously an important current problem. Techniques for improving the reliability of products in general can be classified into fault tolerance and fault avoidance. Fault avoidance techniques aim at a product that is as much as possible free, and likely to remain free, from internal defects (faults). Fault tolerance techniques aim at "tolerating", by redundancy, the effects of faults, so as to reduce or eliminate the disruption caused in the service provided by the product.

Software fault tolerance can be considered as just a new name given to "defensive programming". Indeed, what characterizes studies and techniques known under the label "software fault tolerance" is not so much the basic concepts used (which can all be reduced to the principle of double checking one's results, a principle an early enunciation of which has been found in Charles Babbage's work - [Babbage 1837], quoted in [Randell 87]), but the emphasis on structuring and systematicity, to make these concepts applicable in practice without their complexity getting out of hand and so becoming counter-productive, or their impact becoming impossible to evaluate.

Since the inception of research on software fault tolerance, much progress has been made, through both academic research and practical applications, in clarifying the possibilities of such methodologies and the problems their application entails. At the same time, the advantages to be

obtained from individual software fault tolerance schemes are not clearly measurable, a problem that is common to most engineering methodologies, and especially software engineering methodologies.

For those not familiar with software fault tolerance, there may be some perplexity in accepting its very concept. The application of fault tolerance to design faults may appear as a low-quality solution, as compared to getting the design right in the first place. After all, fault tolerance may be made necessary in hardware by the unavoidability of the physical decay of components; in software, the only reason for failures is design error, something that we are accustomed to track and eliminate.

So, software fault tolerance may look like a sloppy solution unbefitting "real engineers" (as opposed to software "engineers"). This criticism is actually applicable to the whole area of research on software reliability: real engineers do not seek probabilistic estimates of the chance that their bridges contain a fatal design mistake, they follow design procedures that avoid such mistakes.

We think that this point of view is quite unrealistic; since it seems to surface even in the production of standards for critical software ([Barnes 89] summarizes the debate about the proposed British military standard, DEF-STAN-0055), it is worth discussing briefly.

Two main considerations apply. First, software reliability is a problem here and now. All means that can be brought to bear to reduce the severity of the problem in the short term are important. In the long term, it may well be that one design methodology will emerge as the clear solution to all problems, but in the meantime critical systems will be built (for the very simple reason that society already depends on them, e.g. in defense, air traffic control, banking,...) and they should be as good as possible.

Besides, the distinction between design faults and physical faults is less clear-cut than it would appear. Environmental effects and manufacturing tolerances are the two factors that blur the distinction. "Real" engineers, including computer hardware engineers, take into account uncertainties regarding the actual operating stresses, and the actual characteristics of manufactured components, by introducing safety margins in design and by building structures that are insensitive to slight deviations of the components from their ideal behaviour (consider for instance the use of feed-back to compensate the variations in the gain of active components in amplifiers). When component failures are caused by unusual (meaning "difficult to anticipate during the specification phase") combinations of operating conditions and/or component characteristics that are at the boundary of allowed tolerances, it may become difficult to state whether the failure is a "normal" physical failure or was caused by some oversight in design.

Likewise, software design faults typically cause errors (in mature systems) as a consequence of rare or unforeseen combinations of circumstances. Software reliability is affected by the environment where a product is used just as hardware reliability is. A faulty piece of software, when delivered, typically exhibits a behaviour that is "mostly" correct, or, in other words, differs only slightly from what was specified. It should therefore seem natural for software engineers to try and neutralize the dangers coming from these manufacturing imperfections and oversights like other engineers do.

Section 1 of this paper discusses the importance of the problem and summarizes the history of research in, and applications of, software fault tolerance. Section 2 discusses the relevance of software fault tolerance toward the solution of software reliability problems, drawing to the extent that is possible on the research results known. Open research topics and work in progress are also described. Section 3 is dedicated to the more technical aspects of software fault tolerance. It discusses the design choices available to the builder of a software fault-tolerant system, considering the constraints imposed by the characteristics of the applications.

1- MOTIVATIONS FOR SOFTWARE FAULT TOLERANCE, AND A BRIEF HISTORY

1.1- The problem of software dependability

The need for increasing reliability, and in general dependability, of computing systems is commonplace, as is the fact that software dependability (i.e., design faults) now constitutes a major problem. The knowledge that software was not going to be a neat, obviously well-functioning mathematical object reportedly appeared when the first subroutine in the history of computing was unexpectedly born with a bug [Hamlet 87].

The awareness of the problem has since grown steadily, producing the term "software crisis", the discipline called "software engineering", and much progress in software production methodologies. Nonetheless, the problem itself is still growing, because the demands on software are growing: software systems are larger and larger, and the responsibilities placed on software are increasing. So, software errors are a major cause of unreliability in computer systems. According to [Gray 86], for instance, once disk storage is duplicated, software errors become by far the most common cause of errors (except operator errors) in the Tandem transaction-oriented computer system. According to [Giloth 83], about one fourth of the down time in the 4ESS electronic switching system was due to software problems.

A history of interesting computing mishaps can be obtained, for instance, from the publications [ACM SIGSOFT SEN, RISKS]. It will be noticed that those mishaps are often due to software, and their consequences may be very serious.

The discipline of software engineering has offered many improvements in the way software is produced, but no radical solution is in sight [Brooks 87]: formal proofs of correctness are still unfeasible for many real-world products, and suffer from some inherent limitations (with respect to specification errors, for instance); testing suffers from basic limitations, best stated as the fact that it can discover errors, but not prove absence of errors (sometimes not even upper bounds on the probability of errors) [Miller 86, Hamlet 87]). Development methodologies and tools have improved, but none are known to avoid all software errors[1].

1.2- The concept of software fault tolerance

In the situation described above, the idea of software fault tolerance becomes attractive: since creating perfect software components (fault avoidance) seems impossible anyway, one assumes that errors will happen, and builds into the software additional "checks and balances" so that errors are detected in time and their effects corrected as much as is possible.

This implies, of course, that software must include much code that is redundant: it is only made necessary by the expectation of some error in the rest of the code; the latter could, by itself, be sufficient to perform the specified computation, if written correctly.

What advantages can one expect from software fault tolerance? It should obviously improve the dependability of any given program to which it is applied. From a management point of view, effective software fault tolerance could:

- improve the reliability of a product, for a given cost;
- for any given level of required reliability, decrease the cost of testing and debugging, because on-line error detection reduces the need for ad-hoc evaluation of test results, and in general the cost of verification and validation (regardless of the reliability of the product);

---

[1] In the meantime, the problem of errors in hardware design is growing in importance as well, due to the increasing scale of integration and complexity of systems; though some examples of such design faults have received wide publicity (a recent example is a problem with the ALU of the Intel 386), their contribution to system unreliability is still far less important than that of software bugs.

- for a given software system, accelerate (by more effective detection of the remaining problems) the operational testing phase between the first time the system is fully operational and the time it can be confidently delivered to customers, thus giving its producer market advantages;
- make it possible to obtain reliability levels that would be impossible by other means, regardless of the accepted cost. This is relevant for applications where the cost of failure is very high, the advantage of using software, instead of other technologies, is sizable, and the cost of software is, anyway, a small part of the total cost. According to this point of view, structured fault tolerance methods provide a chance to go beyond a hypothetical upper bound on the reliability obtainable in non fault-tolerant software. This upper bound exists because of the fallibility of human developers and the fact that increasing the resources spent on a software project beyond some threshold tends to make results worse, through increased overhead and human communication problems. Intuitively, if five people can build a given module, to some reasonable reliability, in a given time, it will be easier to manage two five-persons teams building two redundant modules than to manage ten people trying to build the same module in the same time with much higher reliability.

On the cost side, there are the development and run-time costs of redundant code, and the need for additional development tools and run-time support with reliability comparable to that of the redundant application.

1.3- A brief history - research

Starting in the early 70s, several proposals were made, and experiences reported, about dual development of software in two different languages, to allow back-to-back testing of the resulting product: the two variants of the program, built to the same specifications, are run on the same test cases, and disagreeing results are taken as a symptom of a software fault [Rault 73, Girard 73, Gilb 74, Shaw 76, Geiger 79]. This method, while it is not software fault tolerance proper (i.e., mechanisms that are left in place during normal, operational usage of the product), has some affinity to what was later called N-version programming.

Proposals for software fault tolerance proper began to appear in the mid 70s [Horning 74, Kopetz 74, Randell 75, Yau 75, Shaw 76, Avizienis 75]. By the late 70s, some methodological proposals had been defined. We will quote here:

1) general concepts about the proper structuring of error detection and treatment (independent of the origin of errors): executable assertions, exception handling, etc.. These are proposals about language constructs, and corresponding program-structuring criteria, that would make it easier to: i) state conditions that the state of the computation must satisfy at given points in the execution, and ii) describe the actions that must be taken if an assertion is not satisfied (see e.g. [Cristian 84] for a linguistic proposal, [Cristian 87] for a general description of the problem).

2) specific proposals for structuring the addition of redundant code in programs in a simple and coherent way:

   2a) recovery blocks, (here, the word "block" has the same sense as in "block-structured languages") [Horning 74, Randell 75], where a block of code contains, besides the "primary" routine for the specified computation, i) an acceptance test to be performed on its results and ii) back-up ("alternate") routines[1], functionally equivalent to the primary, to be invoked, on the state of the data prior to the execution of the primary, if the results of the primary fail the acceptance test; so, the recovery block concept is based on error-detection through executable assertions plus backward error recovery; only if no alternate produces an acceptable result (or if the acceptance test accepts an erroneous result) the block fails, and the resulting exception must be handled at a higher level.

---

[1]In the original proposals, the software variants were called "alternates" when speaking about recovery blocks, and "versions, for N-version programming. We shall use "variants", in both cases (since the requirements are the same), except in the phrase "N-version programming".

2b) N-version programming [Avizienis 77, Chen 78], or "multiple version programming", also known as "design diversity" (the latter is a kind of misnomer, as we shall show later, since all software fault tolerance is based on some kind of design diversity). Here, replicated, diverse, functionally equivalent software modules ("versions" or "variants")[2] are executed concurrently (e.g on the replicated processors of an N-modular redundant system), and their (final or intermediate) results are compared and voted to mask errors and possibly correct the internal states of the variants themselves: it is a form of masking redundancy, with forward error recovery.

Industrial application of software fault tolerance is also reported starting in the 70s, and is detailed in the next section. Typically, these applications employ multiple variants of the software with comparison of results but do not use voting for fault-masking (they are either dual, fail-safe configurations or implementations of "N self-checking programming" [Laprie 87]): correct results are not obtained by majority vote, but by detecting and discarding erroneous results.

During the late 70s and the 80s, research has progressed in different directions. The two most important research centres in the field have certainly been the Computing Laboratory at the University of Newcastle upon Tyne, and the Computing Science Department at the University of California, Los Angeles, respectively for research on recovery blocks and on N-version programming. Articles about the history of research in these two centres, with abundant references, can be found in the volume [Baden 87].

In general, the topics of research in the area may be divided as follows:

- Architecture of fault-tolerant software. A number of proposals have appeared, generally originating from one of the three concepts of recovery blocks, N-version programming, and assertion checking, and often combining aspects of these techniques [Hecht 76, Kim 76, Kim 84, Kant 87, Kim 88, Kim 89]; [Avizienis 84, Strigini 85, Anderson 86] attempt to show how the component elements of those techniques can be subsumed in a more global vision of design possibilities. A series of contributions from the Newcastle group (see [Randell 87]) concentrates on the structuring of complex systems for fault tolerance: nesting of recovery blocks, nesting of generic components and exception propagation, structuring fault tolerance in distributed systems.
- Run-time support mechanisms for running fault-tolerant software. Relatively little has been published on this issue, and it is mentioned in a later section.
- Evaluation of software fault tolerance strategies. This topic is discussed in greater detail in a subsequent section.
- Ad hoc techniques for tolerating special classes of faults. We shall quote the area of "robust data structures" [Black 81b]. These are data structures stored in a redundant form, that allows the original structure to be retrieved after being damaged by a fault/error; different from the well-known error correcting/detecting codes, which are usually meant to protect all the coded information from the kind of random alterations typical of hardware faults, these techniques aim at protecting only the "structural" information of stored data structures (e.g. the topology of a graph structure, regardless of the information located at the nodes), and can catch some logical errors, typical of software faults [1]. This area is now quite well developed, and a complete survey would be quite long [Black 80, Black 81a, Black 81b, Kuspert 84, Ravichandran 89, Sampaio 85, Seth 85, Taylor 80a, Taylor 80b, Taylor 82, Taylor 85, Taylor 87, Yoshihara 83, Kant 90].

---

[1] For instance, a wrong operation on a list, which leaves dangling or inconsistent pointers, might be caused by a hardware fault, e.g. a memory fault in the location that stores a pointer, and would then probably be detected if the memory implemented a proper redundant code; on the other hand, a program error will write into that location a perfectly legitimate code word, that happens to be the wrong value for that pointer. Such an error could still be corrected if the memory representation of the data structure were "robust".

- Exceptions and exception handling. These are a research area in itself, often considered not a part of software fault tolerance, but of the study of programming languages or methodologies. It is particularly interesting for its links to the area of formal specification and verification [Bolot 89, Bidoit 85].

1.4- Industrial use

1.4.1- Fully redundant software in critical application

The reported industrial applications of software fault tolerance are mostly based on parallel execution of diverse variants of the software. Due to its cost, this approach is currently limited to relatively few, highly critical applications. Nonetheless, there are a number of interesting documented applications, that we shall list briefly. Additional documentation and references can be found in [Voges 88].

In these applications, diversity is also often applied in hardware (implementing parallel computation lanes with different processors), to tolerate hardware design faults as well.

All these applications are in the the aerospace, transportation and atomic energy industries. In all three environments, products must be approved by regulatory agencies for what concerns their safety characteristics; a relevant difference among them is that in earth-bound activities the first requirement is normally fail-safeness (the computer system can stop operating, provided it leaves the controlled system in a safe state), sometimes with continuous fault-tolerant operation as a long-term objective, while in avionics the basic requirement is becoming fault-tolerant operation, as computers take on flight-critical tasks.

The involvement of the nuclear industries has been mostly experimental so far, and will be mentioned in the next section. Use of dual programming techniques for an actual implementation is mentioned in [Popovic 86].

In avionics applications, some widely known systems have been operational for several years. We can quote:

- the Airbus A-310 flap and slat control [Martin 82, Hills 83] uses diverse microprocessors and programs, in a fail-safe configuration (the availability of these control surfaces is not considered critical to flight safety);
- the autopilot flight director system for the Boeing 737/300 (by Sperry Co, now Honeywell-Sperry Commercial Flight System Division) [Yount 84, Voges 88] employs dual diverse processors and software. The Boeing 757/767 use a yaw damper system with dual version software;
- flight control systems by Rockwell Collins use dual diverse microprocessors and software [Turner 87];
- the Airbus A-320 fly-by-wire flight control system), where two types of computers and four software variants are used [Traverse 88a, Traverse 88b]. Here the purpose of redundancy is not just safety, but also continued operation in the presence of faults. The four executing variants of the software are organized as two self-checking pairs. "Degradation" of the control laws is possible (to continue operation with fewer resources: the control laws are further from optimality, but the computers still control the aircraft), and a mechanical backup is preserved for one vertical and one horizontal set of control surfaces, to allow minimal manoeuvrability in case of computer failure.

In space applications, there is the example of the Space Shuttle Flight Control System, produced by IBM and Rockwell for the NASA: here, a separately programmed Backup Flight System runs on a single spare computer as a backup for a single-variant, quad-hardware redundant primary system [Spector 84, Carlow 84]. This system got wide media coverage after a bug in the synchronization between the primary and backup systems caused a shuttle mission to be delayed. A detailed explanation is in [Garman 81], and points to the criticality of specifications and specification revision control. Application of software fault tolerance has also been studied for the European Space Shuttle, Hermes [Laprie 87].

There are examples of diverse-redundant systems in railway applications:

- Ericsson Signal systems AB, from Sweden, has used dual variants for interlock systems for the Swedish State Railways [Hagelin 88]. The configuration is fail-safe, in that a disagreement causes the signal lights to switch to red. The same paper mentions the use of diverse software in on-board Automatic Train Control systems as well.
- [Theuretzbacher 86] describes a fail-safe application in a railway interlocking system in Austria. The two redundant software suites that comprise the application are made diverse in both implementation and specification: one implements the interlocking control functions, while the other is a so-called "safety bag", which checks that the control part respects all safety constraints. This subdivision mimics the organization of non-software safety-critical control systems, where basic safety mechanisms are kept separate from the control apparatus to minimize common-mode faults. Interestingly, the "safety bag" was implemented as a rule-based system, to increase methodological diversity between the two subprojects.
- ESACONTROL, from Italy, [Mongardi 88] employed N-version programming in a triple-redundant, software-voted, "safety kernel" for a railway station control system.
- [Turner 87] reports that 2-version software is used in the U.S.A. by General Railway Signal Co., in an interlock system used by two railway companies in the U.S.A, and in another interlock system manufactured by Union Switch and Signal Co.
- Other applications in operational systems in Italy and Singapore, and experimental work in Germany and France, are reported in [Voges 88].

Last, we should mention that dual programming as a method for accelerated testing, detection of specification ambiguities and such is reported to be used with success by many sources. Lists of references can be found in [Gilb 74, Golovkin 86, Voges 88].

These experiences cannot be used to measure the effectiveness of software fault tolerance techniques, because little information is published, and the operational record of the software usually shows few or no serious errors: these software products are built to very high standards, and the reliability level that they try to attain is so high that it could hardly be measured experimentally with any reasonable degree of confidence. The reason why these techniques are used is the belief that they help, although in a non-quantifiable way, to reduce the probability of failures whose cost would be extremely high.

1.4.2- Other software fault tolerance techniques

With regard to the use of robust programming practices, whose boundary with software fault tolerance is only vaguely definable, it is seldom reported per se, so that it is difficult to state its popularity and success in operation.

Extreme care regarding the problem of software faults has been reported for the Electronic Switching Systems of the Bell System. These systems, which have both hard real-time requirements and the need to manage large amounts of long-term data, have employed recovery techniques aimed at recoverying the smallest module affected by an error, and proceeding to a higher-level, wider recovery action (all the way to full restart) if local recovery fails [Spencer 69, Davis 81]. A discussion of this experience is in [Giloth 83], which strongly suggests the employment of all defensive programming techniques. A detailed description of the techniques used in the 5ESS system to detect software errors is in [Haugk 85]: these include acceptance tests on intermediate results, robust data structures with both periodical and "directed" (to recover a data structure which has been found corrupted) audits, checks on resource availability and allocation, and hardware detectors of tight loops, processor inactivity, time-out on watchdog timers and such. Though specific figures for the effectiveness of these techniques are not known, the high reliability and availability demonstrated in the operation of these systems[1] is an indication of success.

---

[1] The AT&T system suffered a major outage in January 1990, which dramatically worsened the previous availability records for the switching equipment (down time of the order of a few

Use of similar techniques has also been reported by [Gray 86] for Tandem systems. This example will be further discussed later on.

1.5- Effectiveness of software fault tolerance

Research to determine the effectiveness of software fault tolerance techniques, as an improvement in the operational reliability or safety of software, has followed two main directions: experimental measurement and analytical estimation. Only very partial conclusions can be reached so far (which is not new in software engineering research). A workshop on the possibilities of evaluating the effectiveness of software fault tolerance was held in 1986 with the participation of most researchers in the area and produced an indication of open questions and research topics worth pursuing [Littlewood 87c]. Most of those questions are still open. Here we shall summarize the work published on the topic. Documentation, and additional references, about most of the experimental work and industrial applications can be found in [Voges 88].

Experimental evaluation of software fault tolerance methods has been conducted in two main ways, to which the next two sections are dedicated.

1.5.1- Single project experiments

These are experiments where a software project was developed, with procedures as close as possible to those that would be used if the proposed methodology were chosen for producing "real-world" software, and then tested extensively. Projects of this kind provide useful insight into the problems of implementing the methodology; its effectiveness can be estimated by comparing the behaviour of the software when the fault tolerance mechanisms are switched on and when they are switched off. Very little statistical value can be given to the data collected, since no attempt is made to validate the representativeness of the experimental sample.

In this class of experiments, we can list:

- those conducted on N-version programming in the nuclear industry. Experiments in dual programming (for back-to-back testing) were run for the Electric Power Research Institute by Babcock & Wilcox and the Univ. of California at Berkeley [Ramamoorthy 81]; and by the Norwegian Halden project jointly with the Technical Research Center of Finland [Dahll 79]. Though back-to-back testing was found useful, the probability of common-mode errors was not evaluated.

  With regard to N-version programming proper, an experiment was run by the Kernforschungszentrum in Karlsruhe [Gmeiner 79, Voges 88] and another (PODS: Project On Diverse Software) by a 3-nation (UK, Finland and Norway) consortium [Bishop 86, Bishop 88a].

  In both cases, the target was a nuclear reactor safety (emergency shut-down) system (for which safety requirements are more stringent than for reactor control systems), and three variants were developed. These were quite small programs, with very high inherent reliability. The reported results were that multiple development seemed to improve reliability, and in particular to help the detection of errors due to specification ambiguities (or, in one case, of a specification error in a domain where the programmer was able to decide what the proper specification ought to have been). Errors common to two variants were reported in project PODS, but none that was common to all three variants. If used in actual operation, the three variants would be used in a safety-oriented configuration (stop on disagreement). Interesting results regarding coincident failures will be described in the section on "Failure modelling". A common problem of these experiments is that most errors reported were found during the test phase. Assuming that the residual bugs left in the operational software

---

hours in 40 years). At the time this is written, we are not aware of any updated availability figures. However, the occurrence of a single, catastrophic failure does not change the picture of a usually very successful fault-tolerant design policy.

would behave in a similar, generally benign, way, though plausible, is not necessarily correct.

- experiments on the effectiveness of dual programming for improving testing and detection of specification problems, which also tend to increase confidence in N-version programming: a recent case is reported in [Bloomfield 86]; for a recent discussion of back-to-back testing see [Vouk 88].
- the experiment run by the University of Newcastle upon Tyne together with MARI [Anderson 85].

  This was a fair-sized naval Command and Control system (8000 high-level language lines). software fault tolerance was applied in the form of recovery blocks, with synchronization of recovery via "conversations" [Randell 75]. Of course, the overall reliability of the final product was lower than for the small programs in the other experiments. A sizable improvement due to the use of software fault tolerance was reported, although it is difficult to assign it a single numeric value (see the paper quoted above for a discussion). However, the software fault tolerance mechanisms certainly avoided about 70 % of the system failures that would be caused by bugs in the application software. These results appear to be very encouraging, in particular considering that the fault tolerance techniques were novel to the programmers, and no pre-existing, mature development tool was available for them.

1.5.2- Statistically oriented experiments

In these experiments, a number of variants of the same software were developed, and then tested in combinations of 2, 3 or more, to obtain statistical data. Of course, the scope of applicability of such results is still limited, since the programs chosen, and the development environment, cannot be considered representative of software in general (or even of a particular class of software). Some experiments, with student programmers, used programs with a probability of failure higher than 10 % per execution: the experimental results may well be dominated by the effects of bugs with high probability of causing a failure, which would not exist in real-life commercial software. Besides, only very narrowly defined hypotheses can be rigourously proven or disproven statistically, which is somewhat unsatisfactory in an unknown field, where there is a lack of hypotheses. Indeed, experiment e) in the following list was the only one to be constructed for the rigorous statistical verification of a hypothesis. The other experiments in the list have more the character of "exercises", i.e. collection of experience to be later used in practice or in formulating theories. Difficulties also arise from fault detection: the main method was back to back testing in experiments a), b), c) and f), and comparison with a "gold version" (assumed to be correct) in experiments e), g). Both methods can obviously fail to detect residual faults: the final reliability of the N-version programming product cannot be measured. All the experiments concur to show that good specifications are of paramount importance.

These experiments were:

a) First experiment at UCLA [Chen 78]. It was mainly a demonstration of the feasibility of N-version programming: a small program for the numerical solution of partial differential equations was implemented by 18 programmers using 3 different algorithms. A limited test campaign gave encouraging results.

b) Second experiment at UCLA [Kelly 83, Avizienis 84]. This was oriented to evaluate the effect of different specification languages on the degree of diversity obtainable. 18 PL-1 programs were obtained from three specifications (in English, in OBJ and in PDL) of a textbook-level database application, with sizes of several hundred lines. The published conclusions point to a sizable increase in reliability in the 2- and 3-version executions, even for moderately reliable versions.

c) North Carolina State University and other institutions [Scott 84]. The purpose was to validate the models developed by the authors for the reliability of different software fault tolerance schemes. Sixteen programs, written as a class assignment (size unknown), were instrumented in different ways and tested on 50 test cases each. The results corroborated the simple models used by the authors, and showed a gain from the use of recovery blocks. N-

version programming could not be applied because the specification did not imply unique correct results.

d) NASA experiment with 4 U.S. universities [Kelly 86]. An English language specification of an avionic application (integration of the readings of a redundant set of accelerometers to obtain a corrected acceleration vector: a heavily mathematical application) was supplied to 20 teams of two programmers (five teams per university). The variants were written in Pascal and were 1000-2000 lines long. Problems with the specification complicated the development; some results are published in [Tso 87, Kelly 88].

e) Universities of California (Irvine) and Virginia [Knight 85a and 85b, Knight 86a and 86b]. 27 program variants were obtained from the same specifications used in a previous experiment on software reliability models by NASA and the Research Triangle Institute (the program examines radar information to determine the presence of a target). The variants were tested extensively (one million test cases) to determine whether their failures could be considered statistically independent: the result was that independence could be excluded with a very high degree of confidence. Interestingly, the same experiment indicated a very high coverage of the residual errors in the variants, so corroborating the idea that N-version programming can significantly boost the reliability of inherently good variants (the produced variants exhibited a quite high reliability, compared to all the other experiments in this list: success on more than 99.9 % of the test cases for most programs, and on the whole set of one million cases for a few).

f) An experiment conducted jointly by UCLA and Honeywell-Sperry Commercial Flight Systems Division. It consisted in the development and evaluation of 6 variants, in 6 different languages (Pascal, Modula-2, Ada, C, Lisp, Prolog), of a flight control program. [Avizienis 88b] gives preliminary results (few faults common to more than one variant) and discusses the effect of development rules on the likelihood of common faults.

g) An experiment comparing the effectiveness of back-to-back testing with other fault-detection strategies, at University of California, Irvine [Shimeall 88]. It used 8 variants of a combat simulator program, with low mean reliability: 0.74 probability of correct results, measured on a sample of 10,000 test cases. As a reliability-enhancement method, back-to-back execution yielded an increase of the fraction of correct results to only 0.81. As a testing method, it did detect some faults that were not detected by any other method, but provided an overall error coverage of only 0.89 on average. Other testing methods outperformed back-to-back testing (in number of bugs found: there was no measurements of the effect on program reliability). The experiment seems to show that each testing method is best at finding some specific category of bugs.

h) Lastly, results of a small scale experiment with robust data structures are described in [Black 81b].

1.5.3- Analytical estimation

1.5.3.1 Generalities

Analytical estimations of effectiveness (and sometimes of performance) of fault-tolerant software have been published in a number of papers [Hecht 76, Grnarov 80, Laprie 84, Eckhardt 85, Hecht 86, Saglietti 86, Cha 86, Laprie 87, Littlewood 87a and 87b, Arlat 88, Csenski 89, Arlat 90].

They differ in the models and analytical tools used and in the assumptions made. The main problem in using these models is the difficulty of estimating the values of the parameters, in particular the probabilities of errors common to redundant components (replicated modules, or the producer of a result and its verifier). Some discussion of the problem of estimating such probabilities is in [Saglietti 87] This information must be obtained experimentally, but we are still far from being able to determine it with any confidence. Although much experimental work is aimed at selecting development rules that make that probability as small as possible, the comparison between alternative rules is done mostly by qualitative observations on the peculiar anecdotal history of each development. This situation, while not very promising for the precise

between different variants (i.e., the probability masses associated with the intersections between the failure domains of the variants), for a realistic distribution of inputs and states.

The simplistic assumption of statistically independent failures in the redundant software modules has sometimes been used. Although convenient for mathematical analysis, this hypothesis represents neither reality, according to experiments, nor an ideal goal (an ideal goal would be zero probability of common error, i.e. negative correlation). Some experiments (experiment at Universities of California, Irvine, and Virginia - henceforth called UCI-UVA for brevity, and PODS experiment) have given relevant results. The UCI-UVA experiment has disproved (for one particular sample of variants, of course) the independence hypothesis. The latter experiment has suggested that it may hold in some cases, and specifically for small software modules.

More precisely, [Bishop 88b], analysing the results from the PODS experiment, concludes that coincident failures in two variants were not normally due to similar programming errors (faults). The coincidence is instead explained by a fault-masking effect in Boolean decision logic, a well-known phenomenon in the study of combinatorial circuits. In both variants some intermediate result was erroneous over a large subset (different between the two variants) of the input space, but this error was normally masked (for instance because the wrong result was a Boolean value, and was fed to a Boolean OR function together with a correct, TRUE value), and the small region where it was not masked (i.e., in our example, all the other inputs to the OR were FALSE) coincided for the two variants. Different faults (bugs) appeared to produce failures with independent probabilities.

These results seem to suggest that the positive correlation between observed failures can be reduced by proper design of the program variants, oriented to detect errors before they are masked; of course, the masking effect actually reduces the failure rate of the software, so it must be avoided only when error detection is considered important (e.g. during testing). [Bishop 89] discusses these development-related issues and how this theory could be applied to explain the high failure correlation observed in the UCI-UVA experiment, since the program used in the latter computed a combinatorial function of Boolean values (each obtained through processing of numerical inputs). [Brilliant 90] analyzes the behaviour of individual faults in the UCI-UVA experiment, and concurs on the fact that some concurrent failures where caused by different faults (they were not caused by faults that were "either the same logical flaw, or similar logical flaws located in parts of the program that compute the same part of the application"). Unfortunately, the errors in the individual Boolean intermediate variables (all defined as failures in the analysis) were not examined separately, so that nothing is concluded about masking effects.

Another important point is the probability that the results from two variants, if incorrect, are equal. So long as the erroneous results are different, they cannot outvote the correct results: even with a majority of failed variants, the software may still produce a correct result or an exception signal, rather than an erroneous, potentially catastrophic output. This probability is obviously influenced by the number of possible results: it is 1 (worst case) if the result is a binary value. But it is usually impossible to quantify it convincingly, and the typical simplifying hypotheses, that it is the inverse of the number of results possible, is obviously optimistic.

Some interesting contributions [Eckhardt 85, Littlewood 87a and 87b] give a probabilistic analysis of the consequences of different kinds of assumptions about the independence (subtly different statements are possible for such assumptions) or correlation of failures of different variants on the statistics of common failures. In particular, it has been shown that programs developed "independently" (in the sense that they are extracted randomly and independently from a population of possible programs) do fail independently on individual inputs, but may exhibit failure correlation, on a population of inputs, if some inputs are more likely to induce failures than others. Also, there is evidence that forcing methodological diversity in development is generally better or no worse than not forcing it. These contributions provide an elegant description of the intuitively expected behaviour of such software, but for the software developer they only restate the need to obtain variants with diverse failure behaviours.

2- THE UTILIZATION OF SOFTWARE FAULT TOLERANCE

2.1- Fields of application for software fault tolerance.

Where can software fault tolerance be applied beneficially?

There is a common misconception that software fault tolerance is worth applying only for life-critical applications, where the supposedly very high costs of software fault tolerance are acceptable. It is due mostly to the identification of software fault tolerance with N-version programming, its most visible and most expensive form. Instead,

- software fault tolerance can be applied in different degrees depending on the applications, so as to match the cost of the redundancy to the expected benefits;
- the criticality, or risk associated with, computer applications varies in a continuum[1]. In many applications that are not deemed life-critical, large amounts of money are staked on the dependability of the software. In some cases, the software can directly cause damage to valuable items, either information (databases, banking systems) or physical objects (control systems); in others, the software may cause unavailability of a computer system, which in turn causes the financial loss. In this light, many cases can be found where even expensive forms of software fault tolerance are justified.

2.2- Validity of software fault tolerance approach.

Having established that software errors can cause large financial losses (or missed gains), we have to determine whether software fault tolerance is a valid defense against such occurrences.

All experiments to date have shown software fault tolerance to effectively improve the reliability of software to which it was applied; what is really necessary is to assess its cost-effectiveness in each individual case, and compare it to that of other methods. A rational way to decide about its application would be to estimate the cost of software errors, and their likelihood as a function of the effort in fault avoidance. Estimating then the cost of introducing tolerance for each class of faults, and the effectiveness of it, one could decide how much software fault tolerance to apply and where.

This is, of course, an ideal planning process based on unknown probabilities. In practice, the freedom of the designer is limited, and choices simplified, by other factors besides the lack of information:

- the effort in fault avoidance will often be determined by a standard software development procedure that cannot be altered much, and whose resulting reliability can be approximately estimated;
- the allowable expense per expected increment in dependability will be determined by a standard policy of the customer, more or less conservative depending on his market situation (attitude of the public, of regulatory authorities and of the competition);
- in some applications, there is a well-defined set of catastrophic failure modes which will invite the use of software fault tolerance protections as an obvious and economical precaution; in others, obvious audit and cross-checking procedures will play the same role; in others, the complexity of the system may make it desirable to apply a "blanket" approach, such as global replication, to reduce the residual failure probability.

---

[1] Even the boundary between life-critical applications and other applications is somewhat arbitrary: it is useful to make sure that we pay attention to obviously dangerous situations, but there is no such sharp demarcation in reality. An error by a computerized billing system may deprive a person of telephone service or electricity, which in some situations may cause loss of life. Since all the direst consequences may be caused by a computer error, given proper accompanying circumstances, a judgment of the "hazard", or possibility of causing damage, in a system, can never be disjoint from some judgment about the likelihood of such damage.

2.3- Applicable research results

Software fault tolerance has been studied for a reasonably long time, but there is as yet no such thing as a set of accepted rules or (possibly diverse and alternative) established design paradigms for practical applications. The designer of a new system would need therefore to move from a set of scattered research results and proposals, and scarcely documented instances of practical applications, to select both a set of meaningful requirements and the rules and philosophy for their implementation.

Questions related to the application of software fault tolerance in a project include:

- system design: which techniques are appropriate for each given application, and what problems and constraints they imply;
- development method: which discipline helps best in developing software modules with the characteristics required (most important, diversity between the errors made by different modules in a redundant set);
- support: which libraries, development tools and operating system facilities are needed to support the application programmer;
- evaluation: how much better the system will become as a result of applying software fault tolerance.

Issues of **system design** are treated in many published papers, and are covered in the last chapter of this survey. The nature of work on this topic is mostly speculative, but we can say that many problems are now well understood.

Development methods have been studied mainly in conjunction with experiments, and a few "do"s and "don't"s are reported e.g. in [Kelly 83, Avizienis 88b]. Specification was recognized as a very important issue as far back as [Kelly 83]. The study of specification methods is an important research issue in itself; an interesting possibility is to introduce diversity directly into methods for systematic development of software from formal specifications.

Issues of **system support** for software fault tolerance are quite important, in that no technique can become economically viable for normal usage if the application programmers are required to take care of many low-level details.

To make the addition of fault tolerance to software a standard, and as much as possible programmer-transparent, activity (just as adding double-precision arithmetic), [Strigini 85] suggests the use of a toolset, to be provided with a computer system (hardware, operating system and development environment), including:

- language support for structured exception handling;
- low level run-time support for multicast message sending, multiplexed message reception, state saving and restoring, possibly voting (voting at the application level may be required because of the need for inexact voting, discussed in a later section, but may be very expensive);
- libraries for recovery blocks, cross-check functions, replicated procedure calls, atomic transactions;
- system configuration tools that allow (software) redundancy to be varied independently of other system parameters.

So far, most reported "real-life" applications have dealt with very small software systems, e.g. in avionics. Here, the application-level software running on each piece of hardware is fixed in advance, and of limited complexity: there is no need for a general-purpose operating system, offering support mechanisms for a wide spectrum of possible applications. The problem of operating system support is instead relevant for general-purpose computers, which must support different application-level packages with a standard native operating system. Several papers about operating systems mention the applicability of specific proposed or existing

mechanisms to supporting software fault tolerance [Cooper 84, Banino 85, Powell 88]. A linguistic proposal is in [Shrivastava 79]. Among the projects that have dealt with such issues in a more complete way:

- the Newcastle project [Anderson 85] developed limited operating system mechanisms to support recovery blocks and conversations;
- Dedix [Avizienis 85a, Avizienis 88a] was an attempt to work out, by building a supervisor program for research use with N-version programming, some of the requirements for operating system support to N-version programming. During its development, care was given to specifying a basic set of comparison algorithms among diversely produced results, to the conceptual problems of synchronising diverse replicas of software components, and of dealing with "failed" variants (state recovery, or shut-down of a variant and reconfiguration), and to specify a programming language interface that was reasonably easy to use and to implement.

  But Dedix had the following limits: i) no provision was made for concurrency: so the issues of inter-process communication and distributed recovery were not addressed; ii) it was implemented as an application-level Unix package, and therefore had to use the standard mechanisms of Unix: issues of efficiency, or in general of good implementation in an operating system or operating system kernel, were not addressed.
- [Purtilo 89] reports a fairly complete system for supporting N-version programming or recovery blocks in specific modules of an application, complete with configuration tools and binding facilities for modules written in different languages.
- [Ancona 87a, Ancona 87b] describe a way for structuring the necessary run-time support mechanisms in a layered way; [Ozaki 88] deals with the details of implementing such a scheme, limited to N-version programming and recovery blocks, in an architecture with ring-structured protection (Intel 286).

2.4- Transparency of fault tolerance provisions

An important feature for any reliability-increasing technique is the possibility of applying it without changing the representation of the system (at some level of abstraction). For instance, hardware can be implemented via N-modular redundancy in such a way that application programs are written as though they were to run on a non-redundant machine. This allows the application programmer to concentrate on the functional specifications of the product, and another designer to add redundancy without complicating the program.

By contrast, many of the possible ways of providing software fault tolerance require the application developer to think up tests based on the semantics of each individual module, to worry about whether the use of diverse modules might cause problems, etc.: in short, the application of redundancy complicates the development process.

Two considerations apply here:

- this lack of transparency is a necessary consequence of the willingness to check for deviations from the specifications of the individual applications, as opposed to the specifications of the machine supporting the applications; methods that are fully transparent are useful, but only for errors that violate the specifications of lower level interpreters and for non-application specific fault-treatment;
- it is both possible and desirable, and the choice of a proper structured model of software fault tolerance additions make it more feasible, that at least the individual application programmers be shielded from the need to take care of software fault tolerance at the same time as they program the functional parts of the application. The division of the software into work jobs assigned to different programmers should take into account the desirability of diversity, and configuration tools should make the task of assembling the redundant software parts relatively easy and error free. For instance, it should be possible for a configuration manager to vary the number of software variants in a redundant module without interfering with the individual variants.

2.5- Assessment of reliability improvements

The evaluation or forecast of the dependability of a system is of course of great importance in the acceptance of a design. The proposers of a new technique can be required to provide an estimation of the advantages it will procure.

2.5.1- Limitations

The state of the art, concerning the forecast of the reliability of fault-tolerant software, as we described it, is not very advanced.

Software fault tolerance may take many different, application-specific forms, and general estimates of its effectiveness are currently impossible. This situation is not exceptional in engineering, and indeed the same problem will appear in evaluating some hardware fault tolerance aspects of an architecture. Fault tolerance techniques are useful nonetheless, and can be shown to be.

There are in fact two main ways that lead to introducing redundancy:

- Engineering improvement proceeds by a "weakest link" approach. Important causes of failure are experimentally recognised; then, means to prevent those failures are applied, which appear intuitively useful for the purpose. If they are indeed useful, a new area of the design, or a sub-area of the previous one, becomes the "weakest link" to be dealt with. No global reliability forecast is needed to guide this kind of progress. Models for predicting the reliability of a system become mature after a design methodology has been in use for some time.
- General-purpose "safety nets" are thrown into the design, when their cost appears reasonable, as a form of added insurance: no one would claim that divide-by-0 detection should not be implemented because its actual contribution to operational system reliability is unknown.

Both these ways can be pursued without a precise forecast of their advantages. Indeed, if we were unable to improve designs without a precise forecast of the behaviour of each new design, innovation would be much less successful than it normally is. When a new generation of systems is built, they often behave in ways that violate the forecasts obtained through the use of the models that applied to the previous generation, because the new design increases the importance of some aspect of the system that could be neglected in the old generation. Nonetheless, engineers build novel designs that improve on previous designs, not by simple trial and error but by systematic, though imprecise, reasoning.

The use of software fault tolerance (as most other techniques for increasing the confidence in our systems) is as additional insurance. Its cost must be compared with the risks involved in the use of the system, and its application must be guided by reasonable criteria as to where it can be expected to be most effective in improving system dependability.

2.5.2- Advantages of software fault tolerance for reliability estimation

But another consideration can be made about reliability forecasts. Structured software fault tolerance methodologies (as N-version programming and recovery blocks) could allow one to build, out of unreliable components, a system that is not only more reliable but also more verifiably reliable.

This is because structural reliability evaluation models may be used on the complete software product, like the ones normally used for hardware. As a trivial example, let us consider a system built out of a main module M and a checking module C, such that the system will only fail catastrophically when both M and C fail. Let Ps, Pm and Pc be the failure probabilities of the system, M and C respectively, and assume, for simplicity (the argument also holds without this assumption), that the structure of the two modules reasonably implies statistical quasi-independency of failures in M and C, so that Ps = Pm Pc.

The required Ps might well be so low that no feasible amount of functional system testing would give sufficient confidence that it has been attained. But individual tests of M and C could

give reasonable confidence in values of Pm and Pc low enough for their product to satisfy the requirement on Ps. In other words, proper redundant design helps reduce (and keep reasonable) the demands made on testing. On the other hand, this requires that the redundant components interact only as planned: when we are concerned with design errors, this implies a simple, well structured interconnection scheme that we can trust, without experimental validation, to have very high reliability. Structural models for combined hardware and software reliability are studied for instance in [Laprie 84, Hecht 86, Laprie 89]. The general warning again applies that the behaviour of these models may be very sensitive to the values of parameters (like probabilities of coincident software failures) that are difficult to evaluate in practice.

3- DESIGN ISSUES FOR SOFTWARE FAULT TOLERANCE

3.1- Software fault tolerance versus hardware fault tolerance

Techniques for software fault tolerance only differ from those for hardware fault tolerance in the physical nature of the faults to be tolerated; the goals pursued and the available strategies are otherwise quite similar.

The purpose may be to avoid all consequences of the occurrence of an error on the progress of the computation, as seen from outside the fault-tolerant component (masking); to correct an erroneous system state, so as to allow the subsequent computation to proceed without propagating the effects of the error (forward recovery); to undo all the computation done after the error, in order to redo it without repeating the error (backward error recovery); or, last but not least, to visibly interrupt the computation so that the error does not dangerously propagate, and to leave the system in an acceptable state (safe shut-down). (Note: these philosophies are not mutually exclusive; they can coexist in different subsystems, or at different levels of abstraction or system decomposition.)

In particular, software fault tolerance often requires but slight modifications to mechanisms already in place for hardware fault tolerance, while putting in place sound mechanisms intended for tolerating software faults usually provides, "for free", good capabilities against some hardware faults.

The principle of all fault tolerance is redundancy; in the case of hardware faults, redundant components must be rendered physically independent, so that causes of faults affecting one of them are not likely to affect the others as well. In the case of design faults, this translates into logical independence, or *diversity*: the components used to check results, or to correct them, must be different from those that produced those results in the first place. The concept of diversity is fuzzy: it is not measurable (in a way that correlates to independence in erroneous behaviour), and its importance is only demonstrated by intuition and experience (an attempt to quantify some aspects of diversity is in [Saglietti 90]. This drawback should not be considered peculiar to our field. It is common to all design rules (e.g., safety margins in structural engineering) that take into account the likelihood of errors in the design itself, which are by definition unknown to the designer: the operational errors caused by such faults are unpredictable, and therefore a proof that the rule copes successfully with that class of faults is impossible.

The peculiarities of design (in particular, software) faults that require modifications in hardware fault tolerance mechanisms are most noticeable in the areas of:

- error detection
- error correction for forward recovery (producing a correct result instead of the one found to be erroneous)
- redundancy management (to take into account the notion of software redundancy)

Little change appears to be necessary for:

- backward error recovery, since it consists in using information that has been stored away so as to be protected from errors

- forward error recovery (different from direct correction of a result) which is largely a matter of ad hoc techniques at the application level, no matter what kinds of faults are to be tolerated.

3.2- General implications of the possibility of design faults.

As stated above, a number of mechanisms that are very useful in making software applications tolerant of hardware faults, such as system support for atomic transactions, may by themselves help in tolerating design faults, even without the use of structured software fault tolerance techniques in the application software: a great deal of tolerance to design faults can be obtained in a hardware fault tolerance system. A case in point is the Tandem system, which will be discussed shortly. But a necessary condition is that the possibility of design faults be actually taken into account.

An example is provided by redundancy management. In any complex computing system, it is normal to convey exception signals from all sources to some management entity, whose task is to decide when individual units are in need of recovery, disconnection, substitution or repair. Since the detected errors are just symptoms of the actual faults, the action of the management entity is based on some assumed diagnostic model of how faults cause errors. Typically, repeated errors (detected by hardware) in the same hardware unit are taken as a symptom of a hardware fault in that unit. But, if design faults are taken into account, an attempt must be made to discriminate between hardware and software faults, by observing the patterns of occurrence of errors in relation to the execution of software modules. Error reports to the management software have to contain not only the hardware data, but some information about executing programs as well (as a minimum, this information provides important information for software maintenance). Likewise, error-detection features in the software are a useful supplement for those in the hardware, but the designer must take into account the possibility that these features be exercised by hardware faults. There is thus a need to integrate software and hardware fault tolerance techniques.

Such management issues may become quite complicated: for instance, a process in one node of a distributed system might make other nodes look faulty, by requesting too many remote operations, or under too strict deadlines, so that the other processors appear to produce timing errors. Though covering the whole spectrum of possible failures is very difficult, attention to these problems is essential.

Any error must be taken as a sure indication of the need for correction of the error, and as a hint about the existence of a fault of either the software or the hardware. If repeated errors lead to suspect the failure of a hardware unit, reconfiguration of the system may be in order, to move or restrict applications to non-faulty hardware units. If a program is suspected as the cause of errors, similar actions may be taken, automatically or under human control, to make it harmless: e.g., revert to an older release or shut down that application.

The distinction between local, short term *error correction* and *fault treatment* (or even among several "levels" of treatment - correction of an invalid response by a software object, recovery from errors in the internal state of the object, re-initialization of the hardware processor running the object...) is important: in immediate error correction, the action taken is based only on the information available about the error itself; as one proceeds towards more global, longer term actions, more information becomes available and more lines of action can be considered.

The reaction to errors cannot be entirely delegated to a general-purpose "management" entity. Rather, a multi-level structuring scheme [Randell 83] seems desirable in any complex fault-tolerant system. The detection of an error must be reported to the object (either application or operating system software) immediately involved in the use of the malfunctioning component, and to a management entity; the immediate reaction to the error may be programmed in the user module, or in some higher-level module to which the exception notification is propagated.

The different approaches to the tolerance of software faults can be classified on a scale that has at one extreme "safety net" techniques, and at the other extreme "structured" approaches. The former are general-purpose, application-independent mechanisms, typically for error detection and confinement; the latter give just a structure for integrating detection and recovery, and

usually require a specific style in writing application software, and an awareness of the fault tolerance provisions by the application designer, who has to provide application-specific detection and recovery procedure.

3.3- 'Safety net' approaches

3.3.1- The approach in hardware fault tolerance

Error detection (and confinement, which is usually strictly intertwined with detection) is made more complicated for software faults than for hardware faults by the multiplicity of specialized software elements, compared to hardware. The same hardware parts, providing a few basic functions, are used for all programs (almost), which makes it comparatively cheap and cost-effective to build mechanisms that detect errors in those few functions.

3.3.2- Applicability in software fault tolerance

Software components perform a myriad of different functions, and in principle a different error detection technique is needed for each component, since the possible errors, i.e., deviations from the specified behaviour, vary with the distinct behaviours specified.

Of course, some errors can be described in a general manner, so that a general error detection module, or a general way of designing individual error detection modules, will cover a complete class. Examples are inconsistencies in: use of machine arithmetic (division by 0), usage of data structures (array overflow), state of data structures (inconsistent linked list), procedure calls (parameter type mismatch), etc. The corresponding general solutions can be built into hardware (divide-by-0 or segment overflow detection), into software (run-time audit program for data structure consistency), or into rules for adding ad-hoc error-checking components (array overflow checks in software), either by hand or by integrating the rules in a compiler.

These "safety net" techniques include all kinds of memory protection schemes [Saltzer 75], tagged memory [Levy 84], different kinds of external monitoring [Avizienis 81, Namjoo 82, Schmid 82, Mahmood 88], automatic consistency checking for parameter types, array bounds and such, watchdog timers, signatures on program memory [Eifert 84, Schuette 86], run-time checks of consistency between the installed releases of different software components [van der Beken 85], and so forth. Indeed, a kind of safety net approach is espoused as a principle of good design by e.g. [Denning 76, Linden 76].

3.3.3- Utility and limitations

All these mechanisms certainly improve dependability. For example, while automatic overflow checks increase the probability that any given program will "bomb out", they increase the effectiveness of debugging and provide better error confinement, which generally increases system dependability and improves the designer's confidence that an apparently well-functioning program is working as planned, rather than in some haphazardly fortunate way. These techniques must therefore be part of the armoury of the fault-tolerant system designer; but their effectiveness is limited, in principle, by their acting at a lower level of abstraction than checks based on the specification of the particular application, in the same sense in which, e.g., an error-correcting code on memory locations is low-level with respect to an array overflow detector.[1] On the other hand, application-specific checks, though more difficult to implement,

---

[1] In the hierarchy of virtual machines, or abstract data types, that comprises a well-structured description of the computer, the machine called "memory cell with code" has, among the rules defining it, the set of the legal code words that can be written into it; at a higher level, memory cells are used to build an array, whose definition includes rules about which indices may be used to write into the array. Tests based on the definition of the lower-level machine can detect errors at that level (non-code words), but not errors with respect to the higher-level abstraction (array overflow). Furthermore, if the separation of levels is built into the physical machine, i.e., the error detecting codes are implemented in hardware in the memory, errors with respect to the lower level cease to concern the higher-level designer (no machine instruction can by mistake write a non-code word into memory); on the other hand, checks at the higher level can

can detect behaviour that is erroneous with respect to the specification of the particular application, though it requires no illegal operations by lower-level components, and they also cover many of the same errors as safety-net techniques. The latter are mainly justified as a way to obtain a good coverage with low investment (standard implementation) and low run-time cost (efficient implementation, in hardware or low-level software).

3.4- The Tandem example

An interesting discussion of the tolerance of software faults in existing hardware fault-tolerant systems is given in [Gray 86].

The author points out that Tandem software has a very good MTBF, better than one would expect from an estimation of the number of bugs remaining in the software. Actually, most software errors that do happen are recovered by the hardware fault tolerance mechanisms of Tandem. He synthesizes the features that allow to tolerate software faults as follows.

- software modularity;
- fault containment through fail-fast software modules;
- process pairs;
- transaction mechanisms.

His explanation is that most remaining bugs in mature software are "Heisenbugs" (temporary) rather than "Bohrbugs" (solid; see Section 1.5.3.2). The Tandem hardware/software architecture has a good chance of tolerating such errors because:

- processes are made "fail-fast" by defensive programming, i.e. they run many consistency or reasonableness checks on their inputs, intermediate results and outputs, and abort themselves if an error is detected;
- the structuring of application software into processes that interact via messages reduces error propagation;
- the atomic transaction mechanisms establish natural checkpoints/rollback points, masking the actions of the aborted process to the rest of the world;
- the process pair mechanisms allow easy repetition of failed computations; for processes that cannot use the atomic transaction mechanism, automatic checkpointing is provided. This operational experience should be interpreted with some caution. The applications for which Tandem is specialized are characterized by a natural structuring in short-lived transactions, operating on a data base; the pre- and post-conditions on the state of the data base, that each transaction needs to satisfy, are well defined. It might be difficult to achieve the same results in different application, where acceptance tests were less easy to define, or the interaction among concurrent activities were tighter than through a shared database.

Also, the figures available deal mainly with the percentage of detected error that were actually recovered; the number of undetected errors is of course unknown.

The causes hypothesized for Heisenbugs imply that typical "safety net" mechanisms would greatly contribute to their detection, possibly reducing the need for checks in the application software.

3.5- Structured software fault tolerance

3.5.1- Generalities; design decision space

We shall call "structured software fault tolerance" those techniques where redundancy, both for detecting and correcting errors, is applied to the individual blocks of application software,

---

still by chance catch errors at the lower level (a non-detectable memory error can corrupt a pointer that, when used to access an array, will trigger the overflow detection mechanism).

based on the semantics of each block, and with the goal of masking errors internal to the block; each technique is characterized by a peculiar way of structuring the interactions among redundant parts, which gives a common syntax and a way of managing the complexity added by fault tolerance. These methods are primarily N-version programming and recovery blocks, with a number of intermediate or combined techniques. The possible techniques can be classified with respect to a number of characteristics. This "decision space" for the designer of software fault-tolerant systems has at least the following dimensions (i.e., design decisions that can be taken, and are somewhat independent of each other):

- error detection
- error correction (voting vs rollback vs other forward recovery)
- distribution of executions
- execution of variants to obtain a "spare" correct result (unconditional vs. on demand)
- granularity in specification of Fault Units; granularity for each different method applied

The following subsections describe these dimensions. A conclusion will be that there are no application-independent rules to guide these design decisions, implying that a system and a methodology that must support diverse applications should allow the application designer as much freedom as possible along all these dimensions.

3.5.2- Error detection

Errors in a computation can be detected essentially in one of two ways: by checking that its results have some properties required by its specification (*acceptance test* or *executable assertion*), or by comparing them with results obtained independently for the same specifications. The criteria for comparing the two methods are their cost and the error coverage they provide.

Both approaches have advantages and disadvantages. The difficulty of finding acceptance tests with satisfactory coverage varies very much with the application (more examples of ways to build acceptance tests are in [Hecht 76]):

- if the application has to compute a function that has a computationally cheap inverse function, the acceptance test is straightforward: for instance, a square root procedure can be checked at comparatively low cost by computing the square of its result and comparing it with the input (note that even in this simple case, allowance must be made for arithmetic error);
- sometimes, only a few required properties of the result can be checked at a reasonable cost; take for example a procedure that takes a list Li and sorts it producing an output list Lo: verifying that Lo is indeed sorted, and that it has as many elements as Li, is certainly less complex than checking that Li and Lo have exactly the same elements;
- for some applications, no reasonable-cost acceptance test can be found;
- sometimes, there is no apparent way of checking the results against the specification, because the specification itself is only defined operationally, as a statement of what the program has to do.

In all cases but the first, it may appear desirable to substitute the acceptance test with an independent recomputation of the result (that can then be performed in parallel, improving throughput), and a simple, cheap comparison. The problem in this case is that error coverage depends on the probability of redundant variants failing in the same way. Statistical data obtained so far only tell us that common errors do happen, and still comparison has usually detected many errors. What can be done is to pursue diversity in the development process, and in the structures of the variants, so as to improve our confidence that failure modes will also be diverse. One can try to use, in the different variants, diverse algorithms, programming languages and styles, compilers and linkers (all the stages of the development process must be considered, leading from the initial requirements to the loaded executable module).

So, error detection by comparison seems most effective for problems where the specification can be very abstract, and different algorithms exist that satisfy the specifications. There will certainly be applications where only one algorithm seems appropriate, development teams where all programmers have similar backgrounds, and other diversity-reducing circumstances. It must also be noted that many application problems, as described in high-level specifications, do not have a unique solution: to obtain meaningful comparisons, one has then to add to the specification a preference among different correct solutions, so reducing the diversity attainable.

3.5.3- Generalized consistency checks

The methods of comparison and of acceptance tests are not actually as precisely separated as they might appear to be. This can be shown by first considering the problem of "inexact voting", as encountered in research on N-version programming. It so happens that diverse implementations of the same functional specification often produce (slightly) different results even in the absence of errors. The typical cause is different rounding in floating-point arithmetic (which can also happen with identical software copies running on diverse hardware), or different numerical algorithms. Another kind of inexact voting that has been found to be useful in practice [Kelly 83] is one that ignores "cosmetic differences" (such as added blanks or slight misspellings) between strings of characters. The problem of inexact voting consists in defining a proper distance function over the output space of the redundant module: a pair of outputs are considered consistent if their distance is less than a predefined allowable error. Defining a proper distance function can be very much application-specific (just think of the many ways a distance between arrays of numbers can be defined)[1].

If we now start from the other end of the spectrum, i.e. acceptance tests, we can observe that a possible form of acceptance test for the output of a module is some kind of general agreement with the result of another module. The railway interlock system where a module does the functional computations, and another checks, based on the same physical inputs, that the decision obey safety rules, is an example.

A further example would be a system where a control program implementing a sophisticated control law, highly optimized but too complex to be completely trusted, is checked against a less sophisticated, more trusted program: only large disagreements would be taken to imply errors of the less trusted module (we might add that the disagreement is best measured through an integrator, so that the system would be implemented as three application-level modules, two functional modules and a third as an integrator/comparator).

As a third example, consider two variants that compute a value and then compare it with a threshold: an arbitrarily small discrepancy in the computed values can cause the comparisons to give opposite results: one again needs to generalize the notion of consistency between the two variants, and/or to force a consensus on the comparison results, to prevent divergent behaviours [Brilliant 89].

Summarizing, the distinction between comparisons and acceptance tests, that appears clear when dealing with procedure-size examples, tends to become blurred as we consider how to build redundancy into a complete system.

[Anderson 86] uses the term "adjudication" for the process of both error detection and correction in a structured software fault tolerance scheme, and offers an initial scheme of how this process may be decomposed. [Di Giandomenico 90] gives a survey of the literature and a classification of the adjudication methods that include the comparison of multiple redundant results.

3.5.4- Error correction

Erroneous outputs produced by a software module, and the consequent erroneous state of the system, can be corrected in several ways. These are: backward recovery followed by reexecution of the failed computation; voting; other kinds of forward recovery. These are the

---

[1] Inexact voting is discussed in greater detail in a later section of this survey.

same concepts used in hardware fault tolerance, and we shall not redescribe here the aspects of the different methods that are known from hardware fault tolerance.

Both voting and backward recovery require diverse variants, and their states must be specified to be equivalent when they perform recovery: the variants must have the same state variables (at the beginning of the execution phase to be recovered, for backward recovery; at the end of it, and at the beginning of the next one, for voting).

For voting, a relevant problem is how to choose the consensus value to be forced on the failed variants; this problem is related to that of inexact voting discussed above. For instance, the Dedix system used the median of the results as a supposedly correct numerical value: this can be shown to be satisfactory, in that this value will most certainly be correct. But this line of reasoning is not always appropriate: in a control system, if diverse variants were continually forced to correct their states based on such a decision, this might well lead to instabilities in otherwise correct, or slightly incorrect, control laws.

Another consideration is that backward recovery completely reconstructs the state prior to the erroneous phase of the computation; for voting, which part of the state will be corrected is specified by the variables that are actually voted. A simple model is SIFT, where the whole application is made up of short-lived tasks which input and vote redundant inputs when started, and output results only when terminating: the whole system state is frequently and completely voted upon. In most applications, this is not economically feasible; besides, it would impose too strict a similarity among diverse variants. A compromise proposed in [Tso 87] considers two levels of recovery in an N-version programming system: at every cross-check point, the few variables that are voted can be corrected (the result of the vote is written back into the internal state of the variants). At some specified cross-check points, called *recovery points*, a variant that appears to be corrupted can correct its internal state by copying the state of a correct variant. To this aim, the specifications include a standard format for the transmission of the state of a variant, and each programmer has to write exception handlers to send/receive the full state, and translate between the transmission format and the internal representation of the state.

Forward recovery, different from voting, is usually application specific. Repairing a robust data structure that was corrupted, moving to a state that is safe for peripheral devices, and such can all be seen as forward error recovery. It is difficult to envision general solutions for the support of this wide class of policies; however, specifications for application level libraries could probably be obtained from recurring problems in any specific application area.

3.5.5- Distribution of executions

The redundant software modules can be executed in parallel or sequentially, on the same or on separate computers. In choosing separate computers, one obtains protection against hardware faults, at a cost. Parallel execution is then possible, improving performance.

It should be noticed that completely parallel execution is impossible for acceptance tests: a result can only be checked after it has been produced. This is critical for response times, but not for throughput, since the test on a result can be performed in parallel with the computation of the next result. As for comparison, it can only be completed after a number of variants (typically more than half of them) have computed their results, so that the response time is never so low as that of the fastest variant.

3.5.6- Execution of variants

The variants can be required for error detection, and thus executed unconditionally, or for correction only. In the latter case, their execution can obviously be conditioned on the detection of an error by the primary; on the other hand, executing them unconditionally, in parallel with the primary variant, guarantees fast recovery from errors, and hence good time response for time-critical applications. This is the case with many operational systems, implemented as "N self-checking programming" systems [Laprie 87].

3.5.7- Granularity of Fault Units

A fault unit is a piece of software that one chooses to consider as a black box, whose outputs are tested for correctness. A fault unit may be defined statically, as some piece of source code,

or dynamically, in the domain of execution time: for instance, an audit program can be run once per a defined time interval, or once every n transactions, irrespective of how many lines of code, and which procedures, where run during that interval.

In either case, the size of fault units is very important.

Making fault units smaller:

- increases the cost paid for error detection, for mechanisms with the same cost;
- decreases error latency, for a given error coverage provided by the detection mechanisms;
- reduces therefore error propagation, and allows backward recovery to undo a lesser amount of computation, and forward recovery to correct a smaller part of the system state (lower cost in case of error, in return for higher cost in the absence of errors);
- helps in diagnosing the bug responsible for the detected errors;
- last but not least, it implies the testing of more intermediate results, and hence, in general, it is based on lower level details in the specification. This decreases the dissimilarity allowed among diverse variants, which is the main guarantee both of good error coverage (if done through comparison), and of proper recovery (either through re-execution or through voting). Of course, lower diversity reduces the problem of inexact comparison.

Even taking into account all these implications of the size of fault units, it should be considered that applications do not usually lend themselves to partitioning into parts of arbitrary size. For any given application, only some modularization schemes are natural and yield clean enough interfaces and correspondence to the specification. The designer of fault-tolerant software has then to respect that natural modular structure, and use it as best as possible.

It is also possible to adopt different granularities for different techniques or for different parts of the fault tolerance policy. For instance, in Tandem the unit for recovery is a whole transaction (the natural run-time granule for the application) while acceptance tests are executed repeatedly during a transaction (finer granularity). It is possible to run comparisons between the results of large modules, which are themselves implemented as sequences of recovery blocks, or simply interspersed with executable assertions to decrease the probability of presenting an undetected incorrect result to the comparison.

3.6- Hybrid software fault tolerance

A frequent objection to software fault tolerance is that its costs are excessive with respect to the possible benefits. This objection is mainly motivated by the high visibility of research on N-version programming, which seems to advocate that every software component be written three times (at least), and consume at least three times the CPU cycles.

Actually, although N-version programming has an indisputable appeal for its elegance and apparent high coverage, it is not the only (nor, for that matter, the most widely used) software fault tolerance technique.

It is true that "massive" N-version programming will normally be used for applications that also use hardware fault tolerance by modular redundancy, and hence provide a replicated software structure, and where the cost of rewriting some software is acceptable (see the discussion of motivations in Section 1).

On the other hand, software fault tolerance, or in general techniques for software robustness, are also appealing for areas of computing where the degree of hardware fault tolerance required is low. For instance, in commercial general-purpose systems (office applications, accounting, scientific programming) dependability requirements are often limited to a minimum level of availability, plus the guarantee of data integrity. There are also embedded systems where hardware support for fault tolerance is limited to a watchdog timer or other low cost solutions, (because the risks do not warrant the use of more costly hardware, and because most faults will safely shut down the system) but it would be good to avoid prolonged unreasonable behaviour due to software error or transient faults. These kinds of faults present a worse problem than massive hardware faults due to their potential for very high latency. In such cases, fault

tolerance techniques based on massive replication present an unacceptable cost in terms of throughput (in absence of faults) and development effort, and specialised hardware configurations are often out of the question because of already consolidated designs.

For these applications, a number of effective software fault tolerance techniques exist, that entail a lower cost than N-version programming, and that can improve the robustness of software without adding much to the cost. Without offering 100 % masking of all errors, these techniques may greatly reduce the costs induced by errors.

- first of all, all techniques can be applied selectively to the areas with higher vulnerability and/or value (potential cost of errors);
- robust data structures and periodical auditing have proven extremely effective; their memory overhead is often minimal, and the overhead of auditing can be chosen as desired, by deciding a point of compromise between accepted fault latency and accepted execution cost;
- techniques based on checkpointing and backward error recovery also have an execution overhead that can be adjusted as desired;
- different techniques can be combined in different parts of a system or application, according to what is most cost-effective in each different part [Strigini 85];
- "reasonableness" tests can usually be run at much lower cost than full correctness checks or duplication of the computation; a set of "safety" tests can be built into the software to avoid those failures that are considered catastrophic, rather than guaranteeing absence of all failures [Leveson 83, Leveson 86];
- last but not least, proper care in designing exception-handling facilities and routines can substantially reduce the number of catastrophic failures and improve the availability of the system. It is commonly observed that many "crashes" in time-shared systems are due either to: i) a conservative attitude stating that the safest reaction to a perceived problem is to shut down the whole system: this sometimes protects data integrity, but certainly decreases availability and makes all users pay for a possibly very local problem; or ii) bugs in exception-handling procedures, originated by the idea that "they will never be executed anyway", causing them to receive less attention from designers and management alike.

It should also be underscored that low cost, software-based methods usually also cover a number of hardware faults, otherwise difficult to cope with, and even to observe and identify convincingly, such as transient memory faults.

3.7- Conceptual problems in structured software fault tolerance

This section discusses problems that must be considered whenever structured fault tolerance is employed, i.e., problems inherent in the combination of diverse software components (variants) satisfying the same specification.

3.7.1- The problem of inexact voting

When error detection must be based on comparison of results, the problem arises that diverse variants will often produce different results. For instance, diverse implementations of real-numbers arithmetic functions will usually produce results that differ in the least significant digits (we shall not consider here specifications that allow multiple, very different correct solutions).

For this reason, the possibility of inexact voting is a requirement for practical N-version programming systems. Otherwise, many N-version programming applications would always fail on a "no agreement" situation. The specification of an inexact voting function, or generalised decision function [Avizienis 84], or arbitrator [Anderson 85], or collator [Cooper 84], must tell: i) how to decide that different results are indeed in agreement, and possibly ii) how to choose a consensus value among such different results.

Specification i) can be seen as defining a distance function over the space of the possible results for that decision point, and giving rules to group results into consensus groups based on the

distances among pairs of results. Specification ii) would then define, once a majority set of agreeing results has been found, how to extract the consensus result.

Briefly, inexact voting is based on mathematics much more complicated than exact (bit-by-bit) voting.

This implies higher costs than the kind of voting applied in non-diverse redundancy. Besides, the definition of the distance function may be application-specific, making it more difficult to create a standard (software or hardware) voter component. One could attempt to define one or more standard non-exact voting procedures, that work satisfactorily for most applications (for instance a few alternative distance functions, between numbers and arrays thereof). There is not enough experience to tell whether this would in practice solve the problem.

However, there is a number of cases in which inexact voting is not needed, provided the possibility of design errors has been taken into account. There will be cases where all the implementations of a given specification will provide numerical results that are identical in all the significant digits[1]: specifying that the application programmer (or the voting modules) must truncate or round the results to that precision, before transmission, will take care of the problem. Of course, disagreements in the results of truncation/rounding may be caused by arbitrarily small differences in the original numbers: once again, this must be taken into account in diagnosing the system after a disagreement is detected.

One may expect the difference between the results of diverse variants to grow with time, during a long execution, as their internal states become more different. The assumptions about similarity of results are therefore different, for instance, in a cruise autopilot module and in an automatic landing module: for the latter, one might assume that the results of the variants will, for the duration of the execution, stay in a small range of each other; for the former, they have enough time to diverge greatly, unless some way of periodically accepting new consensus values is built into the application.

3.7.2- The problem of state recovery of variants

State recovery presents a separate problem. Once an error (or even a disagreement between correct variants) has been detected in some result, the first problem is how to mask the error and give the user (of the result itself) a correct result; once that is solved, a second problem is what to do with the failed (or disagreeing) variant[s]. The internal state of the variant is presumably corrupted, hence we may expect subsequent results to be erroneous as well, and thus not only useless but actually dangerous.

A first choice is to kill the erroneous variant altogether. For short-lived tasks (e.g., the task that manages a transaction) this seems advantageous: the module will be automatically "repaired" when its execution terminates, and at the next execution a correct redundant module will be instantiated, from the same code, taking its internal state from an uncorrupted global state (data base, sensor readings). The probability of enough errors to cause failure happening within the life-span of the task is low enough.

For modules with long execution life-spans, this is not appropriate: too many variants would be needed (the analogous hardware configuration is self-purging N-modular redundancy without repair). Then, there is a need to recover a variant that erred, so that it can catch up with the correct variants and carry on its work. Recovery brings with it an inherent problem that we discuss in the following text.

Recovery may be:

1. programmed into the same variant that errs (estimation of approximate correct values, for instance, or reset to safe values). This is the simpler choice, but is not widely applicable; in

---

[1] the definition of "significant digits" may vary. An example would be a message containing the input to a digital-to-analog converter: the preliminary calculations are likely to be carried out with much higher precision than needed for the input to the converter, so the final result will contain useless less significant bits.

particular, as noted in the previous subsection, perfectly healthy variants might drift apart from each other, with time, if no built-in way exists to recover them to consistent states. Or,

2. the recovery may use values produced by the other variants. The problem with this solution, i.e. correcting the internal state of a variant based on the states of the others, is that it requires either identical internal representations of those states (same internal variables laid out in the same way) or a way to translate between diverse representations (with much higher demands on the application software).

Once solution 2 is chosen, some of the decision points must be selected as recovery points.

In recovery blocks, the backward recovery strategy implies that all variants work on the same state variables (and only differ in their local, temporary variables, that disappear at the termination of the block): every decision point may be a recovery point.

With forward recovery (voting and inexact voting), the solutions are:

- to have the whole state of the terminating block passed as a result to the subsequent block, that has no memory of previous invocations; the normal error masking method makes sure that all the variants after the recovery point have the same state (this solution is viable for many cyclic control systems, and is assumed for instance in SIFT [Wensley 78], and in [Kim 84, Kim 86, Kim 88, Kim 89].
- to program an explicit recovery action, separate from the normal error masking procedure [Tso 86].

Summarising, the problem of state recovery arises from the fact that the state variables to be recovered must be external to (i.e., common to and independent from) the variants that use them. This is solvable by several means, as e.g.:

- short-lived variants, that only use long-lived information stored outside themselves;
- transforming the sequential blocks of a variant into communicating modules, and having all long-lived state information passed between them and checked by the error-masking mechanisms;
- programming recovery as explicit, consensual exception handling by the variants [Tso 86];
- setting recovery points on the boundaries provided in the application for the scope of variables (recovery blocks).

CONCLUSIONS

We have surveyed the situation of research and applications of the different techniques known as software fault tolerance. This situation can be summarized as follows:

- there are good reasons to believe that software fault tolerance techniques can be helpful in obtaining dependable systems;
- there is industrial experience about the use of software fault tolerance; for "structured" techniques, this is limited to a few very specialized "shops" (in particular, in avionics, railroad control and telecommunication switching systems);
- no standard rules exist for applying these techniques to whole systems; only the more technical aspects of exception handling and robust data structures have received some kind of formal systematization;
- there is a general comprehension of the issues to be considered when designing software fault tolerance in a new system, both from the point of view of how to build the redundant application software and how to provide support mechanisms in the operating systems. These have been discussed at length in this survey;
- forecasting the effectiveness of software fault tolerance in a system prior to its implementation is currently impossible, though there is a general understanding of the issues involved. All the problems encountered with non redundant software are also present here.

Structured software fault tolerance methods facilitate the use of hardware-like, structural mathematical models, but obtaining the correct values for the parameters of these models is currently an almost impossible task.

We have tried to give a fairly complete account of the technical issues involved, and a guide to the existing literature, sufficient for designers to find their way in the wealth of existing knowledge about the topic.

Much controversy has arisen about the merits of software fault tolerance (in particular N-version programming), with worries by some researchers that its use may be an excuse for applying lower standards in the procurement of critical software [Leveson 87] and statements by others that it can be extremely valuable to alleviate software dependability problems [Avizienis 89]. A panel session on this topic was held at the 1989 IFIP World Congress, where widely different positions were expressed.

The controversy is made more heated by the fact that software fault tolerance is meant to be used in life-critical applications. Now, design diversity in general is an obvious way to pursue *additional* assurance against design weaknesses in complex, critical systems. Adding an analog control channel to a digital fly-by-wire system, or separate safety interlocks to a railway switching system, are decisions suggested by prudence and common sense, whenever affordable, regardless of our inability to tack reliability figures on them, and of the known cases in which such measures failed through common-mode failures. Since economic trade-offs are made in every project, this inability to quantify is bound to haunt all decisions about highly dependable designs. It should be noticed, though, that this problem is not limited to software, or to computer, projects. As the dependability requirements increase, we are less and less able to quantify the effect of design decisions in purchasing additional, increasingly small reliability gains towards the established goal. This is because of the uncertainty on the probabilities of rare events, embodied in the parameter values used in our models and in the aspects of the real world that we choose to neglect in the models. All design decisions become then a matter of "engineering judgment". The very decision of setting very high reliability requirements may be considered unwise [Miller 86].

We believe that both the extreme statements quoted above are true: software fault tolerance is obviously no "silver bullet" and can be used as a selling trick (just as "structured programming" or "artificial intelligence") rather than an engineering concept, with its advantages, drawbacks and uncertainties.

The purpose of this paper is limited to giving the reader a sufficiently (we hope) wide and accurate picture of the current state of the art, with an abundant bibliography. It is impossible to produce some simple "rule book" based on wide experience, which does not exist. We expect software fault tolerance, in one form or another, to be a valuable aid in obtaining dependable software for at least some current applications. In the absence of proven rules, the mix of methods to be applied to a given project, depending on the peculiarities of the application, the manpower and the corporate culture in the project environment can only be a matter of judgment within the individual projects. The lack of convincing quantitative evaluations is a quite real problem; the designer has to avoid both the danger of using ineffective methods, and the danger of refusing a useful method just because it is not sufficiently proven (a standard that, if applied with severity, would have delayed or impeded the application of most widely accepted tenets of software engineering).

ACKNOWLEDGMENTS

The author wishes to thank Jean Arlat, Mohamed Kaaniche, Jean-Claude Laprie and Brian Randell for their helpful comments on the earlier versions of this paper.

REFERENCES

[ACM SIGSOFT SEN] ACM SIGSOFT (Special Interest Group on Software) Software Engineering Notes.

[Amman 87] P. E. Amman and J.C. Knight, "Data Diversity: an Approach to Software Fault-Tolerance", Proc. 17th International Symposium on Fault-Tolerant Computing, Pittsburgh, Pennsylvania, USA, July 1987, pp. 122-126.

[Amman 88] P.E. Amman, J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 418-425.

[Ancona 87a], M. Ancona, A. Clematis, G. Dodero, E.B. Fernandez, V. Gianuzzi, "A System Architecture for Software Fault Tolerance", International Conference on Fault-Tolerant Computing Systems, Bremehaven, West Germany, Springer Verlag, Berlin 1987, pp. 273-283.

[Ancona 87b], M.A. Ancona et al., "Using Different Language Levels for Implementing Fault Tolerant Programs", Microprocessing and Microprogramming, Vol. 20, 1987, pp. 33-38.

[Anderson 85] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software Fault-Tolerance: An Evaluation", IEEE Transactions on Software Engineering, Vol. SE-11, No.12, December 1985; also in [Voges 88], pp. 109-128.

[Anderson 86] T. Anderson, "A Structured Decision Mechanism for Diverse Software", Proc. 5th Symposium on Reliability in Distributed Software and Data Base Systems, Los Angeles, 13-15 January 1986, pp. 125-129.

[Arlat 88] J. Arlat, K. Kanoun, J.-C. Laprie, "Dependability Evaluation of Software Fault-Tolerance", Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan, June 1988, pp. 142-147.

[Arlat 90] J. Arlat, K. Kanoun, J.C. Laprie "Dependability Modelling and Evaluation of Software Fault-Tolerant Systems", IEEE Transactions on Computers, Vol. 39, No. 4, April 1990, pp. 504-513.

[Avizienis 75] A. Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing", Proc. International Conference on Reliable Software, Los Angeles, California, April 1975, in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 458-464.

[Avizienis 77] A. Avizienis and L. Chen "On The Implementation of N-Version-Programming for Software Fault-Tolerance during Execution, "Proc. International Computer Software and Applications Conference, New York, 1977, pp. 149-155.

[Avizienis 81] A. Avizienis, "Fault-Tolerance by Means of External Monitoring of Computer Systems", AFIPS Conference Proceedings, Vol. 50, May 1981, pp. 27-40.

[Avizienis 84] A. Avizienis, J. P. J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, No. 8, August 1984, pp. 67-80.

[Avizienis 85a] 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 Multiple-Version Software", Proc. 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985, pp.126-134.

[Avizienis 85b] A. Avizienis, "The N-version Approach to Fault-Tolerant Software", IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 1491-1501.

[Avizienis 88a] A. Avizienis, M.R. Lyu, W. Schuetz, K.S. Tso, U. Voges, "DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA", in [Voges 88], pp. 129-168.

[Avizienis 88b] A. Avizienis, M.R. Lyu, W. Schuetz, "In search of effective diversity: A six-language study of fault-tolernat flight control software", Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan, June 1988, pp. 15-22.

[Avizienis 89] A. Avizienis, "Software Fault Tolerance", Proc. 1989 IFIP World Congress, San Francisco, California, August 1989, Elsevier 1989, pp. 491-498.

[Baden 87] A. Avizienis, H. Kopetz, J.C. Laprie (eds.), "The Evolution of Fault-Tolerant Computing", Springer-Verlag, Dependable computing and Fault-Tolerant Systems series, Vol. 1, 1987.

[Banino 85] J.S. Banino, J.C. Fabre, M. Guillemont, G. Morisset, M. Rozier, "Some Fault-Tolerant Aspects of the CHORUS Distributed System", Proc. 5th International Conference on Distributed Computing Systems, Denver, Colorado, 1985, pp. 430-437.

[Barnes 89] M. Barnes, "Dependable Computing in the UK", Preprints of the 1st International Working Conference on Dependable Computing in Critical Applications, Santa Barbara, California, August 1989, pp. 7-14.

[Bidoit 85] M. Bidoit, B. Biebow, M.-C. Gaudel, C. Gresse, G. Guiho, "Exception Handling: Formal specification and systematic program construction", IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985, pp. 242-252.

[Bishop 86] P. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, "PODS - A Project on Diverse Software", IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, September 1986, pp. 929-940.

[Bishop 88a] P.G. Bishop, "The PODS diversity experiment", in [Voges 88], pp. 51-84.

[Bishop 88b] P.G. Bishop, F.D. Pullen, "PODS Revisited - A Study of Software Failure Behaviour", Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan, June 1988, pp. 1-8.

[Bishop 89] P.G. Bishop, F.D. Pullen, "Error Masking: A Source of Failure Dependency in Multi-Version Programs", Preprints of the 1st International Working Conference on Dependable Computing in Critical Applications, Santa Barbara, California, August 1989, pp. 25-32.

[Black 80] J. P. Black, D. J. Taylor, and D. E. Morgan, "An Introduction to Robust Data Structure", Proc. 10th International Symposium on Fault-Tolerant Computing, Kyoto, Japan, October 1-3, 1980, pp 110-112.

[Black 81a] J. P. Black, D. J. Taylor, D. E. Morgan, "A Case Study of Fault-Tolerant Software", Software-Pratice and Experience, Vol.11, 1981, pp. 145-157.

[Black 81b] J. P. Black, D. J. Taylor, D. E. Morgan, "A Compendium of Robust Data Structures", Proc. 11th International Symposium on Fault-Tolerant Computing, June 24-26, 1981, pp. 129-131.

[Bloomfield 86] R.E. Bloomfield, P.K.D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, September 1986, pp. 988-993.

[Bolot 89] Jean Chrysostome Bolot, Pankaj Jalote, " Formal Verification of Programs with Exceptions", Proc. 19th International Symposium on Fault-Tolerant Computing, Chicago, Illinois, June 1989, pp. 283-290.

[Brilliant 89] Susan S. Brilliant, John C. Knight, Nancy G. Leveson, "The Consistent Comparison Problem in N-Version Software", IEEE Transactions on Software Engineering Vol. 15, No. 11, November 1989, pp. 1481-1485.

[Brilliant 90] Susan S. Brilliant, John C. Knight, Nancy G. Leveson, "Analysis of Faults in an N-Version Software Experiment", IEEE Transactions on Software Engineering Vol. 16, No. 2, February 1990, pp. 238-247.

[Brooks 87] F.P. Brooks, "No Silver Bullett -essence and accidents of software engineering", IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

[Carlow 84] G. D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System", Communications of the ACM, Vol. 27, No. 9, September 1984, pp. 926-937.

[Cha 86] D. S. Cha, "A Recovery Block Model and its Analysis", Proc. IFAC Workshop SAFECOMP'86, Sarlat, France, October 1986, pp. 21-26.

[Chen 78] L. Chen and A. Avizienis, "N-Version-Programming: A Fault-Tolerance Approach to Reliability of Software Operation", Proc. 8th International Symposium on Fault-Tolerant Computing, Toulouse, France, June 1978, pp. 3-9.

[Cooper 84] E.C. Cooper, "Circus: a Replicated Procedure Call Facility", Proc. 4th Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984, pp. 11-24.

[Cristian 87] F. Cristian, "Exception Handling", RJ5724 (57703) Research Report IBM Almaden Research Center, San Jose`, California , 7-9 1987.

[Csenski 89] Attila Csenski, "Recovery block reliability analysis with failure clustering", Preprints of the 1st International Working Conference on Dependable Computing in Critical Applications, Santa Barbara, California, August 1989, pp. 33-42.

[Dahll 79] G. Dahll, J. Lathi, "An investigation of methods for production and verification of highly reliable software", Proc. SAFECOMP '79, pp.89-94.

[Davis 81] E.A. Davis, P.K. Giloth, "No. 4 ESS: Performance Objectives and service experience", Bell System Technical Journal, Vol. 60, No. 6, August 1981, pp. 1203-1224.

[Denning 76] P.J. Denning, "Fault Tolerant Operating Systems", ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 359-390.

[Di Giandomenico 90] F. Di Giandomenico, L. Strigini, "Adjudicators for Diverse-Redundant Components: Survey and Optimal Adjudicator", IEI-CNR Report No B4-11, and PDCS Report, 1990.

[Eckhardt 85] D.E. Eckhardt, L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion software subject to coincident errors", IEEE Transactions on Software Engineering, Vol. SE-11, No 12, December 1985, pp. 1511-1517.

[Eifert 84] J.B. Eifert, J.P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams", Proc. 14th International Symposium on Fault-Tolerant Computing, Kissimmee, Florida, 1984, pp. 392-393.

[Garman 81] J.R. Garman, "The 'bug' heard round the world", ACM SIGSOFT Software Engineering Notes, Vol. 6, No. 5, October 1981, pp. 3-10.

[Geiger 79] W. Geiger et al., "Program Testing Techniques for Nuclear Reactor Protection Systems", Computer, August 1979, p. 16.

[Gilb 74] Tom Gilb, "Parallel Programming", Datamation, Oct. 1974, pp. 160-161.

[Giloth 83] F.K. Giloth, K.D. Prantzen, "Can the reliability of digital telecommunication switching systems be predicted and measured?", Proc. 13th International Symposium on Fault-Tolerant Computing, Milano, Italy, 1983, pp. 392-397.

[Girard 73] B. Girard, J.-C. Rault, "A Programming Technique for Software reliability", 1st IEEE Symposium on Computer Software Reliability, New York 1973.

[Gmeiner 79] L. Gmeiner and U. Voges, "Software Diversity in Reactor protection System: An Experiment", Proc. IFAC Workshop, SAFECOMP'79, Stuttgart, 16-18 May 1979, pp.73-79.

[Golovkin 86] B.A. Golovkin, "Multiversion programming and its applications", Automation and Remote Control, Vol. 47, No. 7, July 1986, pp. 877-903.

[Gray 86] J. Gray, "Why do computers stop and what can be done about it?", Proc. 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, California, January 1985, pp. 3-12.

[Grnarov 80] A. Grnarov, J. Arlat, and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies", Proc. 10th International Symposium on Fault-Tolerant Computing, Kyoto, Japan, October 1-3, 1980, pp 251-253.

[Hamlet 87] R. Hamlet, "Testing for Trustworthiness", Proc. DIAC 87, Symposium on Directions and Implications of Advanced Computing, (sponsored by Computer Professionals for Social Responsibility), Seattle, Washington, July 1987, pp. 87-93.

[Hagelin 88] G. Hagelin, "ERICSSON Safety Systems for Railway Control", in [Voges 88], pp. 11-21.

[Haugk 85] G. Haugk, F.M. Lax, R.D. Rover, J.R. Williams, "The 5 ESS Switching System: Maintenance Capabilities", AT&T Technical Journal, Vol. 64, No. 6, July-August 1985, pp. 1385-1416.

[Hecht 76] H. Hecht, "Fault-Tolerant Software for Real-Time Applications", ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 391-407.

[Hecht 86] H. Hecht and M. Hecht, "Software Reliability in the System Context", IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 51-58.

[Hills 83] A. D. Hills, "A-310 Slat and Flap Control System Management and Experience", Proc. 5th Digital Avionics Systems Conference, 1983, pp. 6.7.1-6.7.8.

[Horning 74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, B. Randell, "A Program Structure for Error Detection and Recovery", in "Operating Systems", Proceedings of an International Symposium held at Rocquenfourt, April 1974, G. Goos and J. Hartmanis (Eds.), Springer-Verlag, Lecture Notes in Computer Science, Vol. 16, 1974, pp. 172-187.

[Kant 87] K. Kant, "Software Fault Tolerance in Real-Time Systems", Information Sciences, Vol. 42, 1987, pp. 255-282.

[Kant 90] K. Kant, A. Ravichandran, "Synthesizing Robust Data Structures - An Introduction", IEEE Transactions on Computers Vol. 39, No. 2, February 1990, pp. 161-173.

[Kelly 83] J. P. J. Kelly and A. Avizienis, "A Specification-Oriented Multi-Version Software Experiment", Proc. 13th International Symposium on Fault-Tolerant Computing, Milano, Italy, 1983, pp. 120-126.

[Kelly 86] J.P.J. Kelly, A. Avizienis, B.T. Ulery, B.J. Swain, R.-T. Lyu, A. Tai, K.-S. Tso, "Multi-Version Software Development", Proc. IFAC Workshop SAFECOMP 86, Sarlat, France, 1986, pp. 43-49.

[Kelly 88] J.P.J. Kelly, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and early Results", Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan, June 1988, pp. 9-14.

[Kim 76] K. H. Kim and C. V. Ramamoorthy, "Failure-Tolerant Parallel Programming and its Supporting System Architecture", Proc. 4th International Conference Distributed Computing System, San Francisco, May 14-18, 1976, pp. 413-423.

[Kim 84] Kim, K.H., "Distributed Execution of Recovery Blocks: an Approach to uniform treatment of Hardware and software faults", Proc. 4th Int.l Conference on Distributed Computing Systems, May 1984, pp. 526-532.

[Kim 86] K.H. Kim, J.H. You, A. Abouelnaga, "A scheme for coordinated Execution of Independently Designed Recoverable Distributed processes", Proc. 16th International Symposium on Fault-Tolerant Computing, Vienna, Austria, July 1986, pp. 130-135.

[Kim 88] K.H. Kim, J.C. Yoon, "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan, July 1988, pp. 50-55.

[Kim 89] K.H. Kim, H.O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Transactions on Computers, Vol. 38, No 5, May 1989, pp. 626-636.

[Knight 85a] J. C. Knight, N. G. Leveson, "Correlated Failures in Multi-Version Software", Proc. IFAC Workshop SAFECOMP '85, Como, Italy, October 1985, pp. 159-175.

[Knight 85b] J. C. Knight, N. G. Leveson, L. D. St.Jean, "A Large Scale Experiment in N-version Programmig", Proc. 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, USA, June, 1985.

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

[Knight 86b] J. C. Knight and N. G. Leveson, "An empirical study of failure probabilities in multi-version software", Proc. 16th International Symposium on Fault-Tolerant Computing, Vienna, June 1986, pp. 165-190.

[Kopetz 74] H. Kopetz, "Software Redundancy in Real-Time Systems", Proc. IFIP Conference Information Processing 74, Stockholm, Sweden, August 1974, North-Holland 1974, pp. 182-186.

[Kuspert 84] K. Kuspert, "Efficient error detection techniques for hash tables in database systems", Proc. 14th International Symposium on Fault-Tolerant Computing, 1984, pp. 198-203.

[Laprie 84] J. C. Laprie, "Dependability Evaluation of Software Systems in Operation", IEEE Transactions on Software Engineering, Vol. SE-10, No.6, November 1984, pp. 711-714.

[Laprie 87] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, C. Hourtolle, "Hardware -and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions", Proc. 17th International Symposium on Fault-Tolerant Computing, Pittsburgh, Pennsylvania, USA, July 6-8 1987.

[Laprie 89] J.-C. Laprie, "Hardware-and-software dependability evaluation", Proc. 11th IFIP World Congress, San Francisco, 28 August - 1 September 1989, pp. 109-114.

[Leveson 83] N. G. Leveson, "Safety Assertions for Process-Control Systems", Proc. 13th International Symposium on Fault-Tolerant Computing, Milano, Italy, 1983, pp. 232-240.

[Leveson 86] N. G. Leveson, "Software safety: Why, what, and how", ACM Computing Surveys, Vol. 18, No. 2, June 1986, pp. 165-170.

[Leveson 87] Nancy Leveson, "A scary tale: Sperry avionics module-testing bites the dust?", ACM SIGSOFT Software Engineering Notes, Vol. 12, No. 2, April 1987, pp. 23-25.

[Levy 84] H.M. Levy, "Capability-Based Computer Systems", Digital Press, Bedford (Mass.) 1984.

[Linden 76] T. A. Linden, "Operating Systems Structures to Support Security and Reliable Software", ACM Computer Surveys, Vol. 8, No. 4, Dec. 1976, pp. 409-444.

[Littlewood 87a] B. Littlewood, D.R. Miller, "A conceptual model of multi-version software", Proc. 17th International Symposium on Fault-Tolerant Computing, Pittsbugh, July 1987, pp. 150-155.

[Littlewood 87b] B. Littlewood, D.R. Miller, "A conceptual model of the effect of diverse methodologies on coincident failures in multi-version software", Centre for Software Reliability Technical Report, May 1987, and Proc. 3rd International Fault-Tolerant Computer Systems Conference, Bremerhaven, Germany, September 1987.

[Littlewood 87c] B. Littlewood, T. Anderson, "Reliability modelling for fault-tolerant software", report of a Workshop held in Badgastein, Austria, in July, 1986, published by the Centre for Software Reliability, The City University, London, 1987; also in [Voges 88].

[Mahmood 88] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", IEEE Transactions on Computers, Vol. 37, No. 2, Feb. 1988, pp. 160-174.

[Martin 82] D. J. Martin, "Dissimilar Software in High Integrity Applications in Flight Controls", Proc. AGARD-CPP-330, September 1982, pp. 36.1-36.13.

[Miller 86] D.R. Miller, "Making statistical inferences about software reliability", Software Reliability and Metrics Newsletter, Issue 4, December 1986, Center for Software Reliability - Alvey Club, p. 3.

[Mongardi 88] G. Mongardi, presentation at the First Workshop of the Working Group on Computer Dependability of the Associazione Italiana di Calcolo Automatico (AICA), Milano, January 1988.

[Namjoo 82] M. Namjoo, E.J. McCluskey, "Watchdog Processors and Capability Checking", Proc. 12th International Symposium on Fault-Tolerant Computing, Santa Monica, California, 1982, pp. 245-248.

[Ozaki 88] Brenda M. Ozaki, Eduardo B. Fernandez, Ehud Gudes, "Software Fault Tolerance in Architectures with Hierarchical Protection Levels", IEEE Micro, Vol. 8, No. 4, August 1988, pp. 39-43.

[Popovic 86] J.R. Popovic, D.C. Chan, D.B. Butjorjee, B.K. Patterson, "Computer Control in Candu Plants", in Symposium on Advanced Nuclear Service, CAN/CNS International Nuclear Conference, Toronto, June, 1986, quoted in [Voges 88].

[Powell 88] D. Powell (ed.), "Delta-4 Overall System Specification", Issue 2, published by the Delta-4 Project Consortium, Echirolles, France, 1988.

[Purtilo 89] James A. Purtilo, Pankaj Jalote, "A system for supporting multi-language versions for software fault-tolerance", Proc. 19th International Symposium on Fault-Tolerant Computing, Chicago, Illinois, June 1989, pp. 268-274.

[RISKS] Forum on Risks to the Public in Computer Systems, ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator (computer-distributed newsletter).

[Ramamoorthy 81] C. V. Ramamoorthy, Y.R. Mok, F.B. Bastani, G. Chiu, and K. Suzuky, "Application of a Methodology for the Development and Validation of Reliable Process Control Software", IEEE Transactions on Software Engineering, Vol. SE-7, No. 6, November 1981, pp. 537-555.

[Randell 75] B. Randell, "System Structure for Software Fault-Tolerance", Proc. International Conference on Reliable Software, Los Angeles, California, April 1975, in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 437-449.

[Randell 83] B. Randell, "Fault Tolerance and System Structuring", University of Newcastle upon Tyne, Computing Laboratory, Technical Report No. 189, and Proc. 4th Jerusalem Conference on Information Technology, Jerusalem, May 1984, pp. 182-191.

[Randell 87] B. Randell, "Design Fault Tolerance", in A. Avizienis, H. Kopetz, J.C. Laprie (eds.), "The Evolution of Fault-Tolerant Computing", Springer-Verlag, Dependable computing and Fault-Tolerant Systems series, Vol. 1, 1987, pp. 251-270.

[Rault 73] J.-C. Rault, "Extension of Hardware Fault-Detection Methods to the Verification of Software", p. 255-282, in Hetzel (ed.), "Program Test Methods", Prentice-Hall, 1973.

[Ravichandran 89] A. Ravichandran, K. Kant, "Fault Identification in Robust Data Structures", Proc. 19th International Symposium on Fault-Tolerant Computing, Chicago, Illinois, June 1989, pp. 275-282.

[Saglietti 87] F. Saglietti, M. Kersken, "Quantitative Assessment of Fault-Tolerant Software Architecture", Proc. 3rd International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems", Bremerhaven, 9-11 September 1987, Springer-Verlag, F. Belli and W. Goerke (Eds.), pp. 284-297.

[Saglietti 90] F. Saglietti, "Software diversity metrics quantifying dissimilarity in the input partition", Software Engineering Journal, Vol. 5, No. 1, January 1990, pp. 59-64.

[Saltzer 75] J.H. Saltzer, M.D. Schroeder, "The Protection of Information in Computer Systems", Proceedings of the IEEE, Vol. 63, No. 9, Sept. 1975, pp. 1278-1308.

[Sampaio 85] M.C. Sampaio, J.P. Sauve`, "Robust Trees", Proc. 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985, pp. 23-27.

[Schmid 82] M.E. Schmid, R.L. Trapp, A.E. Davidoff, G.M. Masson, "Upset Exposure by Means of Abstraction Verification", Proc. 12th International Symposium on Fault-Tolerant Computing, Santa Monica, 1982, pp. 237-244.

[Schuette 86] M.A. Schuette, J.P. Shen, D.P. Siewiorek, Y.X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes", Proc. 16th International Symposium on Fault-Tolerant Computing, Vienna, 1986, pp. 138-143.

[Scott 84] R. K. Scott, J. W. Gault, D. F. McAllister, J. Wiggs "Experimental Validation of Six Fault-Tolerant Software Reliability, Models", Proc. 14th International Symposium on Fault-Tolerant Computing, 1984, pp. 102-107.

[Seth 85] S.C. Seth, R. Muralidhar, "Analysis and Design of Robust Data Structures", Proc. 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985, pp. 14-19.

[Spector 84] A. Spector, D. Gifford, "Case Study: The Space Shuttle Primary Computer System", Communications of the ACM, Vol. 27, No. 8, September 1984, pp. 872-890.

[Spencer 69] A.E. Spencer, F.S. Vigilante, "No. 2 ESS system organization and objectives", Bell System Technical Journal, October 1969, p. 2607.

[Shaw 76] D.E. Shaw, "Managing a Software Emergency", Datamation, November 1976, pp. 48-50.

[Shimeall 88] Timothy J. Shimeall, Nancy G. Leveson, "An Empirical Comparison of Software Fault Tolerance and Fault Elimination", Proc. 2nd Workshop on Software Testing, Verification, and Analysis, Banff, July 1988.

[Strigini 85] L. Strigini and A. Avizienis, "Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution", Proc. IFAC Workshop SAFECOMP '85, Como, Italy, 1985, pp. 167-172.

[Taylor 80a] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Improving Software Fault-Tolerance", IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 585-593.

[Taylor 80b] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Some Theoretical Results", IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 595-602.

[Taylor 82] D.J. Taylor, J.P. Black, "Principles of Data Structure Error Correction", IEEE Transactions on Computers, Vol. C-31, No 7, July 1982, pp. 602-608.

[Taylor 85] D.J. Taylor, J.P. Black, "Guidelines for Storage Structure Error Correction", Proc. 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985, pp. 20-22.

[Taylor 87] D.J. Taylor, "Crash Recovery for Binary Trees", Proc. 17th International Symposium on Fault-Tolerant Computing, Pittsburgh, Pennsylvania, July 6-8 1987, pp. 80-84.

[Theuretzbacher 86] N. Theuretzbacher, "Using AI-Methods To Improve Software Safety", Proc. 5th IFAC Workshop SAFECOMP'86, Sarlat, France, 14-17 October 1986.

[Traverse 88a] P. J. Traverse, "AIRBUS and ATR System Architecture and Specification", in [Voges 88], pp. 95-104.

[Traverse 88b] P. J. Traverse, "Surete' des systemes informatiques embarques a bord d'avions - Dependability of digital computers on board of airplane", presentation at the 3e Colloque International sur la Securite' Aerienne et Spatiale, Academie Nationale de l'Air e de l'Espace, Toulouse, France, September 1988, (in French).

[Tso 86] K. S. Tso, A. Avizienis, and P.J. Kelly, "Error Recovery in Multi-Version Software", Proc. IFAC Workshop SAFECOMP'86, Sarlat, France, 1986.

[Tso 87] K. S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation", Proc. 17th International Symposium on Fault-Tolerant Computing, Pittsburgh, Pennsylvania, July 6-8 1987, pp. 127-133.

[Turner 87] D.B. Turner, R.D. Burns, H. Hecht, "Designing micro-based systems for fail-safe travel, IEEE Spectrum, Vol. 24, No. 2, February 1987, pp. 58-63.

[van der Beken 85] H. van der Beken, "Safety aspects of the operation of the JET Computer System", presented at the 4th IFAC Workshop on Safety of Computer Control Systems (SAFECOMP 85), Como, Italy, 1-3 October 1985 (not in the preprinted proceedings).

[Vouk 88] M.A. Vouk, "On back-to-back-testing", Proc. COMPASS 88 (3rd annual conference on Computer Assurance), Gaithersburg, Maryland, 27 June -1 July 1988, pp. 84-91.

[Voges 88] U. Voges (ed.), "Software diversity in computerized control systems", Dependable Computing and Fault-Tolerance series, Vol. 2, Springer-Verlag 1988.

[Wensley 78] J.H. Wensley, "SIFT: design and analysis of a fault-tolerant computer for aircraft control", Proc. IEEE, Vol. 66, No. 10, October 1978, pp. 1240-1255.

[Yau 75] S. S. Yau and R. C. Cheung, "Design of Self-Checking Software", Proc. 1st International Conference on Reliable Software, Los Angeles , 21-23 April 1975, pp. 450-457.

[Yoshihara 83] K. Yoshihara, Y. Koga, T. Ishihara, "A Robust Data Structure Scheme with Checking Loops", Proc. 13th International Symposium on Fault-Tolerant Computing, Milano, Italy, 1983, pp. 241-248.

[Yount 84] L.J. Yount, "Architectural solutions to safety problems of digital flight-critical systems for commercial transports", Proc. 6th Digital Avionics Systems Conference, Baltimore, Maryland, December 1984, pp. 28-35.