Design fault tolerance

B Randell

Abstract

The aim of this paper is to provide a personal perspective on the subject of design fault tolerance, and in particular software fault tolerance, as it has developed at Newcastle and elsewhere, and to speculate briefly on how the subject might advance in the future. The principal topics covered are the search for an appropriate set of basic concepts and definitions, the differing styles of fault masking provided by recovery blocks and N-version programs, the growing sophistication of error recovery techniques, particularly in distributed systems, and the problems of assessing the cost/effectiveness of design fault tolerance.

To be presented at the Symposium on The Evolution of Fault-Tolerant Computing, to be held in honour of Bill Carter, in Baden, Austria on 30th June, 1986.

Series Editor: M. J. Elphick
Bibliographical details

RANDELL, Brian

Design fault tolerance. [By] B. Randell.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1986.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 218.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

The aim of this paper is to provide a personal perspective on the subject of design fault tolerance, and in particular software fault tolerance, as it has developed at Newcastle and elsewhere, and to speculate briefly on how the subject might advance in the future. The principal topics are the search for an appropriate set of basic concepts and definitions, the differing styles of fault masking provided by recovery blocks and N-version programs, the growing sophistication of error recovery techniques, particularly in distributed systems, and the problems of assessing the cost/effectiveness of design fault tolerance.

About the author

Professor B. Randell has been a Professor of Computing Science in the Computing Laboratory since 1969.

Suggested keywords

DISTRIBUTED SYSTEMS
ERROR RECOVERY TECHNIQUES
N-VERSION PROGRAMS
RECOVERY BLOCKS
SOFTWARE FAULT TOLERANCE
SYSTEM STRUCTURING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.64404
U. D. C. 681.322.06 519.687
DESIGN FAULT TOLERANCE

Brian Randell
Computing Laboratory,
University of Newcastle upon Tyne

ABSTRACT

The aim of this paper is to provide a personal perspective on the subject of design fault tolerance, and in particular software fault tolerance, as it has developed at Newcastle and elsewhere, and to speculate briefly on how the subject might advance in the future. The principal topics covered are the search for an appropriate set of basic concepts and definitions, the differing styles of fault masking provided by recovery blocks and N-version programs, the growing sophistication of error recovery techniques, particularly in distributed systems, and the problems of assessing the cost/effectiveness of design fault tolerance.

(To be presented at the Symposium on The Evolution of Fault-Tolerant Computing, to be held in honour of Bill Carter, in Baden, Austria on 30 June 1986.)

Introduction

The subject of fault-tolerant computer design was already well established through the efforts of hardware designers such as Bill Carter, when in 1970, I and my colleagues first took an active interest in the subject. Indeed we were aware that the subject went back at least to the work of von Neumann in the mid-1950s. Where our work differed from much of what had gone before (or so we thought) was that we chose to concentrate on the topic which has since become known as design fault tolerance and, in particular, on the problems of tolerating residual software design faults. However, within a year or so of starting the project my hobby of digging into the origins of digital computing led me to find out that in 1837 Charles Babbage had written[12]:

"When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all."

This quotation comes from a paper, unpublished during his lifetime, which Babbage wrote within three years of starting work on what later became known as the Analytical Engine. (The idea of so organising the work of clerical staff "computers") engaged in producing mathematical tables was in fact mentioned in an earlier published paper - one by Lardner describing Babbage's Difference Engine[36].) The Analytical Engine was to be a completely automatic mechanical computer, programmed (as we would now say) by means of Jacquard cards. Babbage's designs for this machine exhibited a very creative concern for mechanical reliability, and his paper makes it clear that he thought that in operation the principal reliability problems would arise from what we would now term software, rather than hardware.
errors, and from mistakes in input data.

Returning to the present century, as I have explained in the Introduction to the recently published compendium of Newcastle Reliability Project papers[46], one of the influences on my thinking, at the time we started the project, was the discussions I had taken part in at the 1968 NATO Software Engineering Conference:

"One major theme of the conference was the great disparity between the level of reliance that organisations were willing to place on complex real time systems and the very modest levels of reliability that were often being achieved – for example, it was also at about this time that there was considerable public debate over the proposed Anti-Ballistic Missile System, which we understood was to involve relying completely on a massively complicated computer system to position and detonate a nuclear device in the upper atmosphere in the path of each incoming missile!

At the NATO Conference there was thus much discussion about improved methods of software design, though there was a mainly implicit assumption that high reliability was best achieved by making a system fault-free, rather than fault-tolerant. Another much-debated topic concerned the practicality of attempting to provide rigorous correctness proofs for software systems of significant size and complexity. Such discussions, I am sure, played a large part in ensuring that, by the time I reached Newcastle, I was seeking to do something constructive about the problems of achieving high reliability from complex computing systems, and yet was feeling rather pessimistic about the practicality of proving the correctness of other than relatively small and simple programs.

The plan for a major research project at Newcastle on system reliability in fact was developed very quickly in discussion with my colleague Jim Eve... From the start, our aim was to study the general problems of achieving high reliability from complex computing systems, rather than concentrate on problems specific to a particular application area or make of computer. Quoting from the original project proposal: "The intent is to investigate problems concerned with the provision of reliable service by a computing system, notwithstanding the presence of software and hardware errors. The approach will be based on the development of computer architecture and programming techniques which facilitate the structuring of complex computing systems so that the existence of errors can be detected and the extent of their ramifications be determined automatically, and so that uninterrupted service (albeit probably of degraded quality until the faulty hardware or software is repaired) can be provided ... [The proposed project] is thus parallel and complementary to work on achieving high reliability from individual hardware components, and on program validation. Both of these topics are of importance, but it is clear that for the foreseeable future the designers of large-scale computing systems will not be able to achieve adequate system reliability by depending entirely on the reliability of the hardware and software components which make up their system."

Such concentration on system structuring issues has been a continued characteristic of much of our work. Another characteristic, and one that was not originally intended, was the extent of the efforts that we felt it appropriate to devote to issues of terminology. In what follows, I will briefly discuss these issues, before addressing various system structuring topics, and the problems of evaluating the cost/effectiveness of design fault tolerance. In so doing, my aim will be to provide a personal perspective on the subject of design fault tolerance as it has developed over the years, at Newcastle and elsewhere, before indulging myself briefly in some speculations as to how the subject might develop in the future.
Terminology Issues

After I and my colleagues at Newcastle had for some time been investigating techniques for designing software so as to include the redundancy necessary to tolerate residual design faults, we became aware of the inadequacies, for our purposes, of the standard terms and definitions then in use by the hardware fault tolerance community. These terms took the concept of a fault as their starting point, and defined it by enumerating the various classes of regrettable but unavoidable component behaviour which were to be classed as faults: for example stuck-at-one faults, bridging faults, etc. The notion of fault was thus synonymous with that of a component failing to meet its specification in some way, and there seemed to be an assumption that the cause of any system failure could in principle be identified unambiguously with one or more such component failures.

The definitions therefore made no allowance for such explicit design faults as the accidental use of a wrong component, or wrong inter-component connections. However the only software faults are design faults, of just such a character. Moreover, it is important to note that identification of the actual fault which has caused a complex software system to fail can be a somewhat subjective affair—depending on how it is thought the software should have been designed in the first place, then various different pieces of code will be identified as being at fault, and differing amounts of processing up to the point of system failure as having been erroneous.

It was Michael Melliar-Smith who first became actively concerned with these terminological (and conceptual) problems—I remember him even paying a special visit to the University's Department of Philosophy to find out whether anyone there had thought at all deeply about the notion of an "error". He got very little help though he reported that they seemed to know a lot about the notion of "truth". The solution that he eventually arrived at involved using a recursive definition of the term "system", and taking the concept of system failure (with respect to some specification) as the starting point, in terms of which fault and error could then be defined.

The first published account of these ideas is contained in a paper [40] which Michael and I wrote on recovery blocks and exception handling. Quoting from this paper:

"A failure of a system occurs when that system does not perform its service in the manner specified, whether because it is unable to perform the service at all, or because the results and the external state are not in accordance with the specifications. A failure is thus an event....

We term an internal state of a system an erroneous state when that state is such that there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault....

The term error is used to designate that part of the state which is "incorrect". An error is thus an item of information, and the terms error, error detection and error recovery are used as casual equivalents for erroneous state detection and erroneous state recovery.

A fault is the mechanical or algorithmic cause of an error, while a potential fault is a mechanical or algorithmic construction within a system such that (under some circumstances within the specification of the use of the system) that construction will cause the system to assume an erroneous state. It is evident that the failure of a component of a system is (or rather, may be) a mechanical fault from the point of view of the system as a whole."

This set of concepts and definitions served as a common basis for the sets of lectures making up an advanced course on computing system reliability which was
first given in Newcastle in 1978, and repeated in California the following year. This course, incidentally, provided us with our first opportunity for extended interaction with Bill Carter, (though I personally had known Bill from my years at the IBM Research Center, Yorktown Heights), who provided a splendid series of lectures on hardware fault tolerance for the course[18].

Perhaps the most gratifying early response to our work on concepts and terminology came from some of the staff of International Computers Ltd. Although they were not involved in the design of fault-tolerant computing systems, they found that use of the terminology helped to clarify the varied responsibilities involved in designing, manufacturing, installing and maintaining computer systems — in other words they applied the terminology to ICL itself, viewing the whole enterprise as a fault-tolerant system!

In subsequent years improved versions of these definitions were produced, in particular by Tom Anderson and Pete Lee for their book "Fault Tolerance: Principles and Practice" [4]. Their work was one of the inputs to the very extensive discussions that took place within the IEEE Technical Committee on Fault Tolerant Computing and IFIP Working Group 10.4, starting in 1981/82. These discussions have, I hope, now culminated in the broadly similar, though significantly developed and extended, definitions documented in the paper by Jean-Claude Laprie[35], to whom is also due the introduction of the term "dependability" for the generic concept subsuming such system characteristics as reliability, availability, safety and security. Our original formulation had led us to speak of "reliability with respect to a particular specification", and so to think of safety and security, for example, merely as special cases of reliability. This attitude tended to provoke fruitless arguments with the safety and security communities, whereas the term dependability is somewhat more neutral.

The power of terminology to clarify (or confuse) continually impresses me. My most recent personal experience in this line in fact concerns the topic of security. With a colleague, John Dobson, I have lately been involved in discussions on the topic of how to design and structure highly secure computing systems. These discussions have their origins almost entirely in our realisation that there are useful analogies to be discovered by considering security and reliability to be different special cases of dependability. In particular, this has led us to question a number of current practices with respect to the design of multi-level secure systems (such as the use of a so-called "Trusted Computing Base") and to feel that we now have a much better conceptual grasp on how to build computing systems intended to exhibit both high security and high reliability. The approach we have started to develop is outlined in papers[42,26] which we suspect the security community will view as provocative but possibly interesting, whereas the reliability community may well regard them as belatedly obvious.

In essence, what we have done is realise that two research areas, which have developed largely independently of each other, each with its own jargon, its own conferences and its own fashions, can be usefully coalesced into a single subject. After all, in computing science we have more than enough unsolved problems, without having to create spurious ones simply through problems of terminology.

System Structuring

The provision of design fault tolerance requires the incorporation of useful redundant information into the design, for purposes of detecting and recovering from, or directly masking, the effects of residual design faults. This additional information will be useful only to the extent that it does not itself suffer from faults which are equivalent to those which it is intended to help tolerate — something which is facilitated (but unfortunately not guaranteed) by having as independent as
possible a design process, i.e. by achieving a high degree of "design diversity"[8].

Since the principal cause of residual design faults is complexity, the use of appropriate structuring techniques is crucial if the addition of means of design fault tolerance to a system is not to increase its complexity to the point of being counter-productive. Thus issues of system structuring have been a principal concern of the Newcastle Reliability Project from the outset.

The first structuring technique which we developed was the recovery block scheme. Part of the thinking behind this scheme was that although good use could be made of a variety of different error detection mechanisms in a system, it was highly advantageous to have a single coherent error recovery strategy. I remember similar ideas being put forward by Roger Needham, who once suggested to me that the best way to implement a time-sharing system was to concentrate first on producing a dependable "checkpoint and restart" facility, before gradually adding in the various other features that users expected!

One source of the recovery block concept was, if my memory serves me correctly, a survey that a founder-member of the project, Tony Mascall, made of a number of then-current large real-time systems and, in particular, of their facilities for restarting after a failure. I remember feeling dissatisfied with the schemes he described, which seemed unduly ad hoc and limited. However it took Jim Horning, who was visiting from Toronto during the Summer of 1973, and Mike Melliar-Smith to find a simple linguistic formulation for structuring error detection and recovery in sequential programs which, being recursive in form, allowed nested recovery regions and so removed one of my principal objections to the earlier checkpoint and restart schemes. Their recovery block structure required a suitable mechanism for providing automatic backward error recovery, something that was central to the idea of having primary blocks and alternates be independent of each other. I produced the first such "recovery cache" scheme, a description of which was included in the first paper on recovery blocks[31] (although it was later superseded[3]), together with a discussion of "recoverable procedures" - a mechanism that Hugh Lauer and I had proposed as a means of extending the recovery block scheme to deal with programmer-defined data types.

By 1975 we had moved on to consider the problems of providing structuring for error recovery among sets of cooperating processes and, having identified the dangers of what we came to term the "domino effect", had come up with the notion of a "conversation" - something which we later realised was a special case of a nested atomic action, or transaction. These ideas were presented that year at the Los Angeles Conference on Reliable Software[41], a conference at which Al Avizienis, following in Babbage's footsteps, discussed the possibility of carrying out the same computation concurrently or sequentially by three or more independently-written programs and using either hardware or programmed comparisons to mask errors, i.e. what later became known as N-version programming[9,20].

The recovery block scheme can be viewed as a software equivalent of the well-known hardware technique of standby sparing - however one in which (i) standby spares (the alternates) are switched in automatically and temporarily, as a result of a programmed check (the acceptance test). rather than until the module they were replacing had been repaired, and (ii) the spare is not just a replica of the primary component, so that the provision of graceful functional, as well as performance, degradation is permitted. N-version programming is of course an even more direct analogue to a hardware scheme, that of N-Modular Redundancy, though it uses design diversity, rather than simple replication, as a means of providing multiple modules. Also it can allow more complex schemes of voting, because of such difficulties as that of obtaining absolutely identical results from differently-designed numerical algorithms.
In the years that followed, much further work was done on both schemes. Comparing them, their principal difference is that though both attempt to achieve fault masking, recovery blocks also provide a hierarchical error recovery strategy, which can be invoked when fault masking is known to have been unsuccessful. The fault masking strategies provided by the two schemes differ in various ways. One is that N-version programs are obviously well-suited to take advantage of multiple processors. However the alternates within a recovery block could be executed in parallel with the primary block, even though the acceptance test is to be applied to each in turn. (Indeed schemes have been produced for continuing with the tentative execution of the code following a recovery block, in parallel with the evaluation of its acceptance test[32].)

The other obvious difference between the ways in which the two schemes attempt to mask faults concerns the method by which results are checked for errors - in the one case by checking the answers produced by rival algorithms for exact (or possibly near) identity, in the other case principally via the use of a separate programmed acceptance test. (For a more comprehensive discussion of this point, see Chapter 9 of [4].) In fact, as Tom Anderson has recently shown[1], the use of a general adjudication mechanism, which first filters then arbitrates between the results produced by the redundant modules, can provide a general fault masking scheme within which those of recovery blocks and N-version programming appear as special cases - moreover, one could embed this scheme within the sort of hierarchical error recovery scheme provided by recovery blocks, if one does not wish to assume that faults are invariably successfully masked.

Since the early work on recovery blocks, the Newcastle project has gone on to consider error recovery in more generality, and to develop an overall approach to exception handling in multi-level systems. We now regard recovery blocks as just a convenient notation for a stylized form of exception handling, in which backward error recovery is the only form of error recovery used. However, forward error recovery is more appropriate for predictable and well-defined errors (e.g. simple hardware component faults and input errors), leaving backward error recovery to be used just for faults whose consequences have not been, or cannot be, accurately specified (such as unmasked residual design faults).

Based on this approach of providing both forward and backward error recovery via exception handling, work has been undertaken by Flavio Cristian at Newcastle, and subsequently elsewhere, on formal means of specifying and of designing programs incorporating exception handling[21] and in extending these ideas to the provision of robust abstract types[22]. The exception handling scheme which we have used is based on the "termination model". Operations on an abstract type are expected to provide a suitable exception report and to terminate, unless of course they can terminate normally. The user of an operation is expected to be able to assume that, in the absence of such a report, the operation has indeed been successful, and that no further checking of its results is needed. This, of course, is the software analogue of the well-known hardware scheme of self-checking circuits introduced by Bill Carter[19]. (Incidentally, another interesting link between the worlds of hardware and software fault tolerance is provided by the work, at the University of Waterloo, on extending both the theory and the practice of error detecting and correcting codes to various types of data structure, such as linked lists and binary trees[47, 48, 15]. Redundancy is added to such data structures so that it would require a given minimum number of elementary changes to modify a valid instance of a structure into another valid instance. If less than this number are produced, as a result of a fault, the error can be detected. Error correction consists in transforming the erroneous instance into a valid instance, using a minimum number of elementary changes.)
The main thrust of the work at Newcastle in recent years has been to extend these ideas on exception handling to situations involving asynchronous activity. We first considered problems relating to competition for shared resources[13], and then of cooperation among time-shared processes[44] within a single computing system, before concentrating particularly on distributed systems - not just distributed computing systems, but systems which could, for example, include people and machines, interacting with, and perhaps through, one or more computing systems. In common with other groups, we have made much use of the concept of an atomic action – the crucial notion of providing atomic actions as an explicit linguistic construct, independent of any particular means of guaranteeing atomicity, had been introduced to us by David Lomet when he spent the academic year 1975-76 at Newcastle on leave from the IBM Research Center, Yorktown Heights. Lomet's work, like similar ideas then being developed in the database world, can be seen as a way of making explicit some of the ideas behind the notion of "spheres of control" - pioneering work by Charlie Davies[23,24] which has had a continuing influence on our own thinking (see, for example the paper by Santosh Shrivastava[45]) particularly since we had the pleasure of his participation in the 1978 and 1979 lecture courses referred to earlier.

As mentioned above, our own work on reliability in distributed systems has concentrated on the problems of coping with erroneous data flow rather than design faults per se. In fact comparatively few of the many groups working on distributed computing systems or distributed database systems have had an explicit interest in design fault tolerance. Rather, their recovery techniques are mainly intended to deal with occurrences such as processor crashes, disk failures, deadlocks, faulty input data, etc. Nevertheless the backward recovery facilities they provide are of potential relevance to errors due to software design faults. (For a recent survey of this work, see [29].) In our own treatment, as indicated earlier, we have not stopped at the provision just of backward error recovery, but have gone further, and investigated the provision of atomic actions for which both forward and backward recovery are permitted, so producing an overall framework suitable for tolerating both anticipated and unanticipated faults in asynchronous systems[17] - our belief is that such facilities, which approach the generality inherent in the spheres of control concept, will often be required in general distributed systems, even if it is possible to manage without them in distributed database systems.

In addition to work on recovery techniques, the world of distributed computing systems has also been the source of a quite separate thread of research of relevance to design fault tolerance. This is work on various synchronisation and consistency algorithms which are expected to cope with arbitrary behaviour (including that due to possible design faults) by the processors and communications devices involved. Perhaps the best known such work is that on so-called Byzantine agreement protocols[34]. Note that the actual protocol design is assumed to be correct - direct attempts at designing distributed algorithms which are in some sense inherently fault-tolerant are very rare, though one most impressive example has been presented by Dijkstra[25]. Were such heroic efforts necessary for all forms of design fault tolerance, the future of the field would be bleak indeed. As it is, the sorts of approach discussed earlier, which are based on the assumption that reliable structuring facilities are first provided to aid fault masking and perhaps error recovery, seem more widely applicable, since they allow users to employ conventional programming techniques yet still gain a measure of design fault tolerance.
The Effectiveness of Design Fault Tolerance

In 1981 Tom Anderson and Pete Lee concluded the chapter on Software Fault Tolerance in their book [4] with the statement:

"Unfortunately, it may take some kind of computer-controlled disaster to bring about recognition that fault prevention techniques are not sufficient for high software reliability, and that software fault tolerance can and should be provided in application areas which have high reliability requirements. There are few theoretical reasons for not adopting fault tolerance in software systems, although practical justification must await the results of on-going research efforts."

In fact during 1981-84 a project at Newcastle led by Tom Anderson undertook a large scale investigation of the cost-effectiveness of software fault tolerance based on recovery blocks and recoverable "dialogues" between multiple processes. (A dialogue provides a restricted form of the concept of a conversation[5].) This project involved the implementation of a model Naval Command and Control System, and the use of the PDP11/45 hardware recovery cache which had been built earlier[37]. The model, though only partial, was as realistic as possible, being designed and implemented by people who were already experienced in this application area, using approved software design practices and languages, and with the whole exercise being closely monitored by Navy personnel. The actual evaluation was based mainly on comparing the reliability of the system with its fault tolerance provisions either enabled or disabled. Quoting from the final report of the project[2]:

"Over the entire program of experiments, the event counts show that 222 failures could have occurred due to "bugs" in the software of the command and control system. But of these 222 potential failures only 52 actually happened - the other 165 were masked by the use of software fault tolerance. This represents an overall success rate of 74 percent... Projections suggest that with further improvements to the recovery software a coverage factor of over 90% could have been achieved... The supplementary cost of incorporating fault tolerance in the command and control system was approximately 60 percent... Overheads in system operation were measured as: 33 percent extra code memory, 35 percent extra data memory, and 40 percent extra run time (though the system still had to meet its real-time constraints)... Our overall conclusion is that these experiments have shown that by means of software fault tolerance a significant and worthwhile improvement in reliability can be achieved at acceptable cost."

Ideally of course one would perform an experiment involving a large number of carefully controlled trials - this has been the approach to evaluating the effectiveness of N-version programs which has been taken by the UCLA group (who have recently developed a software test-bed for facilitating the assessment of N-version programs, and their parallel execution on a set of processors[10]). Of necessity these evaluation exercises, of which there have by now been several, have used much smaller, and hence somewhat less realistic, examples than the model command and control system.

In some cases, the reliability improvements observed have been somewhat modest. For example, Al Avizienis and John Kelly have described[11] an experiment involving the programming of an "airport scheduler" exercise independently by thirty Computer Science students. Eighteen acceptable programs were produced, and used to make up 816 different 3-version programs. Running independently, the programs produced in total a correct answer for 73.1% of the transactions, reported an error for 5.8%, and failed to detect an error in 21.1% of the transactions. Running in 3-version programs, the percentage of correct transactions went up to 80.1%, and the percentage in which there were failures to detect an error fell to 3.6%. (The
percentage of transactions that reported an error increased to 16.3%, mainly because the voting procedure required strict equality.)

Much higher reliability gains are implied by the figures reported for a subsequent experiment at the University of Virginia and the University of California at [33], which was based on a small, but nevertheless reasonably realistic aerospace example. Each of twenty-seven experienced programmers, working independently, implemented and carefully tested one program. These programs were then each subjected to a total of a million tests, of which just 18,962 failed. (Thus even running independently, the programs achieved a success ratio in excess of 99.9%.) Although no actual N-version program trials are reported, it is clear from the data given in the paper that if these programs had been used to construct 3-version programs the vast majority of these potential failures would have been masked. Indeed I estimate that an average failure coverage factor of at least 93% would have been achieved, though of course at the cost of a threefold increase in resources. Unfortunately such facts are not immediately apparent from the paper, since the authors have chosen to concentrate their entire discussion on another (admittedly important) issue. This is that their results confirm that the reliability of an N-version system may not be as high as theory predicts under the assumption that independently written programs will fail independently - in practice people occasionally make the same mistakes even when they are working completely independently. (Similar cautions will of course also apply to predictions concerning recovery block programs.)

Despite their limitations, such assessment efforts are to be applauded, particularly when one remembers how few software (or hardware) design tools and methodologies have been adopted (or rejected) on the basis of an adequate quantitative evaluation exercise. In fact another important reason for pursuing the difficult task of producing cost/effectiveness assessments of the different software fault tolerance schemes is to provide constructive guidelines as to how and when such schemes should best be used. For although software fault tolerance is best thought of as complementary to, rather than competitive with, software fault prevention schemes such as testing and formal verification, it is not obvious how a given system development project should subdivide its limited time and resources between these different activities. There have nevertheless been some attempts to provide such guidelines, such as that by Hech[30], and work on software reliability modelling is often aimed at assessing the amount of further testing which would be worthwhile.

Currently, the use of software fault tolerance in anger, so to speak, is largely limited to systems calling for ultra high reliability[49] - and few systems, even those in which system failures would endanger human life, aim to provide completely automatic means of tolerating software design faults. Instead, in most cases just duplicate programs are used[27,39,38], so as to provide error detection and reporting, i.e. "fail-stop" operation[43], rather than fault masking. (Incidentally, such duplication has also been used to assist the testing process[28].)

In the present circumstances of comparatively little industrial experience, or experimental data, perhaps the best argument that can be deployed in favour of the use of software fault tolerance is one due to Michael Melliar-Smith. Formal verification, testing and fault tolerance each depend on a set of assumptions. For example, verification depends, inter alia, on the adequacy of the theorem proving tools and techniques, testing on the choice of tests and the care with which their results are inspected, and fault tolerance on the completeness of the error detection scheme and the degree of actual design diversity achieved. In addition, the three schemes all depend on having an adequate system specification. By using formal verification and testing and fault tolerance the number of assumptions on which belief in the reliability of a system rests can be reduced to the absolute minimum, namely that the system specification is satisfactory, for this is the single assumption that is shared
by all three approaches.

**The Future**

Continued improvements in hardware cost/performance and in memory capacities are providing the world with a seemingly irresistible opportunity to design, implement and (try to) rely on ever larger and more complex software systems. There might of course be breakthroughs in formal verification, or in test case generation and analysis - however in the interim the future will surely bring increased acceptance and use, and continued development, of various software fault tolerance schemes.

However, I suggest that what is really needed is a broader perspective on the whole subject. Formal verification, testing and fault tolerance are all essentially just schemes for checking the consistency of a redundant body of information, namely a system's specification, design and implementation. They merely differ as to when, and therefore how, such checking takes place - either prior to the provision of any input data, or after some sample data is supplied but before the system is put into service, or while the system is in service and operating on real input data. Should such a consistency check fail, this implies that the implementation and/or the specification is at fault. (Ideally the specification is so much simpler than the implementation that the latter possibility can be ignored - in practice, all too often this is not the case.) When the three techniques are compared in this way, it seems clear that there ought to be a more coherent way of discussing their relative merits, and some more methodical way of deploying each to its best advantage - indeed of performing meaningful (and perhaps computer-assisted) tradeoffs between them.

Some of the work we have carried out can be seen in this light - for example that on formal methodologies for the automatic placement of certain kinds of acceptance test[14], and on "safe" programming, a scheme which makes use both of formal verification and of run-time evaluation of assertions[6]. However, much more work could be done along these, and similar, lines.

A second issue is that, just as it is important to regard fault tolerance, when applied to design faults, as a part of a subject which also includes formal verification and testing, so is it necessary to view the problems of tolerating design faults within a perspective which encompasses all types of fault. The problems of tolerating different kinds of fault have much in common. For example, they share a common dependence on mechanisms for limiting the spread of errors. In many cases, quite general fault tolerance mechanisms can be devised, which are capable of tolerating a variety of faults. This is just as well, since it is often difficult, if not impossible, to determine the exact cause of a failure in a complex computing system. However if design fault tolerance is to have its full impact on the reliability of large and sophisticated hardware/software systems, rather than just on relatively modest safety-critical software modules, say, then further research is needed towards making design fault tolerance a well-integrated part of an overall reliability strategy.

The third and final issue I wish to mention is that, as VLSI integration levels are increasing, improved hardware and software design methodologies and tools aimed at fault prevention are appearing. Unfortunately, it is by no means obvious that such improvements are doing more than keep up with the increases in complexity, if that. It therefore seems likely that design fault tolerance will be needed for VLSI, as well as software, particularly if wafer-scale integration succeeds in allowing even more complex designs to be attempted.

All this provides, I would argue, yet another reason to seek to minimise differences between hardware and software design methodologies, tools, languages and support environments. There seems to be no inherent reason why one should not use
the same specification and high level design languages (and hence design fault tolerance schemes) regardless of whether the ultimate aim is an implementation stored in memory, or laid out as intricate geometrical patterns on silicon. This would facilitate the exploration of differing hardware/software tradeoffs, and also the testing and assessment of designs prior to their being committed to silicon.

Some investigations of these issues are being made at Newcastle, as elsewhere. For example, a prototype system for converting programs written in the OCCAM language directly to (schematic versions of) VLSI layouts has been constructed by a Ph.D. student, Mike Lynch. Secondly, Martin McLauchlan and Albert Koelmans, who are involved in the development of a successor to the VLSI design language STRICT ("Strongly Typed Recursive Integrated Circuit")[16], are starting to consider what set of facilities it should contain related to the provision of design fault tolerance. These are, however, but initial explorations of what I feel sure will be a very interesting and fruitful area.

Concluding Remarks

Despite my comments above, to date the topic of design fault tolerance has largely been the province of software designers, who have tended to have little in common with hardware designers working on conventional fault tolerance. Few people have bridged this gap at all successfully - one of the most notable exceptions being Bill Carter, both through the wide scope of his achievements (e.g. from self-checking circuits, to formal verification of microprograms) and through the "systems thinking" which characterises his approach to the whole subject of reliability. It has therefore been both a pleasure and a privilege to provide this brief contribution on the topic of design fault tolerance to this symposium held in his honour.

Acknowledgements

I am indebted to Tom Anderson for his helpful comments on a draft of this paper. Research at Newcastle on system reliability is sponsored by the UK Science and Engineering Research Council, and the Ministry of Defence.


43. R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An approach to


