Abstract:

The fault tolerant approach to building a reliable system acknowledges that perfection is impossible (or at best, very expensive) and therefore tries to cope with the consequences of residual defects within the system. Fault tolerance has an established role in detecting and masking component faults in hardware systems, but has also been advocated as a defence against deficiencies of design. This paper argues, in question and answer format, the case for adopting design fault tolerance techniques in practical systems.

An earlier version of this paper appeared in:


Can Design Faults be Tolerated

By

T. Anderson

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 196
March, 1985

©1984 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
bibliographical details

ANDERSON, Thomas


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 196.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)

Dewey (18th):
001.64254
U.D.C.
681.322.06

Suggested keywords

DESIGN FAULT TOLERANCE
RECOVERY BLOCKS
RELIABLE SOFTWARE

Abstract

The fault tolerant approach to building a reliable system acknowledges that perfection is impossible (or at best, very expensive) and therefore tries to cope with the consequences of residual defects within the system. Fault tolerance has an established role in detecting and masking component faults in hardware systems, but has also been advocated as a defence against deficiencies of design. This paper argues, in question and answer format, the case for adopting design fault tolerance techniques in practical systems.

About the author

Dr. T. Anderson is currently a lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.
Introduction

The short answer to the question posed by the title is "Yes". A more cautious, and less simplistic, response would be that in certain circumstances, with appropriate provision of redundancy and allied supporting mechanisms, it is certainly possible to provide a measure of tolerance to faults of design. However, although this question may serve as an appropriate title, and starting point for discussion, it does not adequately address the significant issues concerning the application of fault tolerance techniques to deficiencies of design. As is usually the case, the first, and perhaps most important, step is to ask the right questions. In this paper, I propose to substitute five further questions in place of my title and, in answering those questions, will argue the case for the use of design fault tolerance in the development of reliable computing systems. In so doing I hope to justify the short and cautious answer already given above.

The following discussion will largely be conducted with reference to the use of design fault tolerance in software systems, since current techniques were devised primarily for use in software development. It is, however, customary (and accurate) to make the observation that the increasing complexity of VLSI designs suggests that software design techniques may also have a valuable contribution to make in the area of hardware design.

Is there any need for design fault tolerance?

The traditional approach to achieving reliability in computing systems has largely been based on fault prevention, the goal of which is to prevent system failure by ensuring that no faults can be present when the system is in operation. There are two aspects of fault prevention, which may be termed fault avoidance and fault removal.

Fault avoidance concerns those techniques which aim to avoid the introduction of faults during the design and construction of a system. Since this approach may not be completely successful, fault removal techniques are necessary to validate the implementation of a system and remove any faults which are thereby exposed.

If fault prevention is not expected to completely eradicate faults from a system, then fault tolerance techniques can be employed to provide a last line of defence. By incorporating redundant elements it may be possible to cope with the effects of a fault during system operation, and thus avert the occurrence of a failure (1).

The provision of tolerance to anticipated hardware faults has been a common practice for many years, for two reasons. First, hardware is by its nature built from physical components, and these are susceptible to the introduction of faults arising from the natural processes of decay and deterioration in the physical realm. Second, the effects of physical faults can often be categorised into
well understood "failure modes" for components, which greatly assists the selection of appropriate redundancy. The first reason establishes the need for fault tolerance in hardware, while the second facilitates its provision.

For software the situation is rather different. Software is by its nature abstract rather than physical and not subject to faults introduced during operation by "software rot" (contrary to popular belief). Of course, the representation of the software may be corrupted by the effects of a hardware fault, but that is a separate issue. Thus, any faults in the software itself are design faults, due to mistakes made during the development process which escaped the vigilance of fault prevention techniques. It follows that the effects of software faults are difficult, if not impossible, to anticipate - which makes the implementation of techniques to contend with those effects somewhat more demanding.

Nevertheless, there have been a number of proposals which advocate the use of fault tolerance in software. I would argue that the need for such techniques is, in principle, self-evident. A wide range of techniques are available for fault prevention in software (including notations for requirements, specification and programming, design methodologies, validation and verification, management and support environments ...) but though these may be highly beneficial, they certainly do not eliminate all faults from programs. Future software engineering developments may enable us to achieve such high standards of software design that fault tolerance has no role to play, but I suspect this will only be the case when either mechanically checked formal verification is possible and economical for practical systems, or software can be generated automatically from specifications. Even then, the problem of inadequate or inaccurate specifications will remain.

Current techniques for building reliable software systems rely largely on "exhaustive testing", that is, testing continued until either the project budget, or the software tester, is completely exhausted. The diminishing return on investment from such testing argues forcefully for the adoption of a wider range of techniques - which could sensibly include design fault tolerance.

Is design fault tolerance a mature technology?

A recent study in this area concluded that "all evidence indicates that [fault-tolerant] software technology has progressed sufficiently ... to move out of the laboratory into practical systems" (2, section 1.4). That is, the technology of design fault tolerance has been extensively developed in theoretical and experimental contexts, and is now ready to be adopted in practical systems.

To provide tolerance to design faults, redundancy must be extended to cover the design. Avizienis terms this redundancy "design diversity" (3). Two different approaches have been proposed, namely recovery blocks (4) and multi-version software (5). Although these are often viewed as distinct (even competitive) methods, the
differences are principally implementation issues rather than major conceptual matters. Indeed, an elementary generalisation can be presented which encompasses both approaches.

Suppose we have a software module $M_1$ which receives input and produces output as shown in Figure 1.

![Diagram of Module](image)

**Figure 1. Single Module**

To provide tolerance to possible design faults in $M_1$ we supply independently designed alternative versions $M_2, \ldots, M_n$ and apply one or more of these to the input. We must decide which of the $n$ possible outputs is actually to be used and so a selection must be made by an adjudication module A. This is depicted in figure 2.

![Diagram of Design Diversity](image)

**Figure 2. Design Diversity**

The simple structure of figure 2, highly reminiscent of hardware NMR structures, is sufficient to represent either recovery blocks or N-version programming. The only substantive generalisation is that the form of the adjudication algorithm has not been specified.

In N-version programming, module A uses either simple majority voting, or inexact voting when permitted tolerances on outputs preclude a unique correct output. Recovery blocks usually apply a fixed acceptance test to the output in a predetermined priority sequence (Lee (6) suggested variant forms of acceptance test). Other adjudication algorithms are possible, of course, and have been suggested by the authors of hybrid schemes (7,8).
The standard descriptions of recovery blocks assume sequential execution of $M_1, \ldots, M_n$ when required, with the ability to regenerate an initial state, whereas $N$-version programming was envisaged as employing parallel execution on multiple processors, with state replication. From a strict semantic viewpoint, these are mere details of implementation. $N$-versions could be executed serially, just as the recovery block alternates could be performed in parallel.

The study of fault tolerant software (2) quoted above summarised the results of seven attempts to develop software reliability models (most recently by Scott et al. (7)) for design fault tolerance notations. Much of this work suffers from a lack of empirical validation, and depends heavily on assumptions which may be questioned. Nevertheless, all the models do confirm the potential for reliability enhancement which design fault tolerance offers.

Other recent work has addressed the applicability and effectiveness of design fault tolerance in real-time systems (9, 10) and in concurrent systems (11). Cristian has continued work on the interrelationship of fault tolerance and exception handling mechanisms and notations (12). A lot of earlier work on design fault tolerance has been summarised elsewhere (1). It can surely be claimed that the conceptual development of design fault tolerance has received considerable attention and is now well understood. But is it of relevance to the implementation of systems in practice?

Can design fault tolerance be utilised in practical systems?

There are many instances of the use of design fault tolerance in the software of practical systems with high reliability requirements. However, this use is often ad-hoc, unstructured and of limited fault tolerance capability. It is usually referred to as defensive programming and is often flagged in the software with the comment "This should never be executed but ...".

Multi-version software has been developed for a small number of practical systems, usually at the insistence of the relevant regulatory authority that the software should not constitute a single point of failure. Examples are the slat and flap control system of the Airbus A310 (13) and the flight control system of the Boeing 737-300 (14). Both of these systems employ dual dissimilar versions of the entire software. Outputs are compared, and if a discrepancy is detected the systems revert to a passive mode of operation, alerting the flight crew. Similar approaches have been adopted in systems for railway signalling (15) and nuclear reactor shutdown.

Perhaps the best known instance of design fault tolerance is that used in the NASA Space Transportation System, the "Space Shuttle" (16). A single back-up computer runs in parallel with four primary computers. The primary computers execute the normal software system whereas the back-up computer executes an alternative version of the software for mission critical functions. Error detection is
performed by comparison, voting, built in self-checking, and the ultimate acceptance test - the astronauts themselves. A switch-over to using the back-up software can only be initiated manually by the crew.

None of these practical systems makes use of design fault tolerance in the modular and hierarchical fashion which is possible using recovery blocks or N-version programming. Hierarchical use of recovery blocks has been achieved in a research project at Newcastle which has implemented a software system of realistic scale to assess the effectiveness of design fault tolerance techniques (17). The actual application selected supported a subset of the facilities of a naval command and control system, and was implemented in accordance with commercial practice by experienced programmers. Approximately 8,000 lines of program (written in CORAL) generated nearly 50 Kbytes of machine code.

Since the application was designed as a concurrent real-time system containing 14 separate processes, it was necessary to devise a notation and mechanism supporting a form of "conversation" (4,11) which would coordinate the recovery capability of interacting processes. The resulting structure was called a dialogue (since this means a conversation of a formal or restricted nature) and will be described in a forthcoming paper (18). Essentially, dialogues are used to define multiprocess recovery blocks which are statically nested at compile time. Additional features facilitate their use in cyclic computations.

An earlier form of dialogue, based on dynamic process structuring, met with little success and was quickly replaced by the static form. Thereafter, the only difficulty encountered by the system developers was in devising the acceptance tests needed to provide run-time error detection. Many of these tests were selected without difficulty, but certain situations caused problems. These were resolved by resorting to a structural consistency check of primary data structures. The overall conclusion of the system developers was that the design fault tolerance techniques, though novel, were certainly usable in building a practical system.

Can design fault tolerance improve system reliability?

Very little information is available as yet on the effectiveness of multi-version software in practical systems, though most projects report that construction of dual versions was of great assistance during development as a means of simplifying testing procedures. Similarly, only limited encouragement can be drawn from the reliability models for design fault tolerance mentioned earlier. Of course, the Space Shuttle diverse software provided perhaps the most famous bug ever recorded (16) by failing to synchronise, and aborting the first launch. (Note that the fault tolerance operated flawlessly on that widely publicised occasion.)

Experimentation at UCLA with N-version programs (19) involved the implementation of 18 versions of an airport scheduling program by students (each version contained about 400 PL/I statements). All 816
triad combinations of these versions were evaluated as 3-version programs. In 27% of these combinations, two correct versions succeeded in masking the faulty computations of a defective third version. Only in 3% of the cases was an incorrect result produced. These experiments confirmed the positive results of preliminary evaluation studies (5) on N-version programming.

Our aim at Newcastle in applying fault tolerance to the design of a naval command and control system was to obtain a quantified evaluation of the effectiveness of design fault tolerance techniques in the context of a practical system. When the software had been thoroughly tested and was considered ready for "operational" use, a lengthy series of experimental runs was performed using a simulated tactical environment and a variety of action scenarios.

A detailed analysis of these runs showed that on 53 occasions the software would have failed in the absence of fault tolerance, but by means of fault tolerance, failure was averted in 40 of these situations. Thus a failure coverage of 0.75 was achieved, and statistical analysis indicates that we can be 90% confident that the coverage exceeds 0.67. A further 12 events were analysed, of which four were ignored due to uncertainties in classification, four represented needless recovery, and the remaining four events were failures caused by the use of fault tolerance. If these four failures are offset against the 40 successes, then the notional coverage drops to 0.68.

The above analysis was guided by experience in running the command and control system in two modes: with and without fault tolerance. A direct comparison between these two modes of operation provided additional evidence of reliability enhancement. The Mean Time Between Failure for the fault tolerant software was 0.74 hours whereas without fault tolerance the MTBF was reduced to 0.31 hours (ratio 2.36). The proportion of missions completed without failure was 56% with fault tolerance enabled, compared with 47% when fault tolerance was not enabled (ratio 1.19).

Many of the failures of the fault tolerant system would not have occurred if reliable recovery mechanisms had been available, as would surely be the case if these techniques were to be used in practice for a succession of different application systems. If failures due to defective recovery are eliminated the experimental results indicate that a failure coverage of 90% and a nine-fold improvement in MTBF could be achieved. Almost 90% of missions would have been completed successfully.

Is design fault tolerance a cost-effective means of achieving reliability?

This final question is the most pertinent, but unfortunately a definite answer cannot as yet be given.

The results of the Newcastle project, summarised in the previous section, certainly indicate that design fault tolerance techniques can yield a significant increase in reliability. But was this
improvement worthwhile? The reliability enhancement was achieved at a
cost of: 60% extra in software development, 33% extra code summary,
35% extra data memory, and 40% run time overhead (largely due to
additional synchronisation). These figures are probably on the high
side, reflecting the novelty of the techniques, the extent of their
utilisation, and the lack of fine tuning of the completed system.
Furthermore, increased development costs can be offset by gains from
economies in software testing.

However, the ability to engineer the reliability of a system is
not so much a consequence of the availability of techniques for
improving reliability as it is dependent on information concerning
the relative cost-effectiveness of those techniques. In order to
construct a system which will have a given level of reliability,
within a fixed budget and adhering to project time-scales, the
reliability engineer needs to select appropriate techniques and
apportion the amount of effort to be devoted to each. Only when data
is available on techniques of fault avoidance and removal as well as
for fault tolerance will it be possible to make a rational
determination of the best mix of reliability techniques. In the
absence of such data (as is largely the case for software) I would
argue for the eclectic approach. Optimal solutions are rarely
achieved by putting all one's eggs in one basket. A well-engineered
approach to building highly reliable software is likely to be based
on striving for perfection, but at the same time recognising that
imperfections will still be present - and therefore design fault
tolerance will be needed to cope with them.

Conclusion

To provide a summary I reiterate my questions and answers.

Is there any need for design fault tolerance? Potentially yes, given our current inability to achieve perfection.

Is design fault tolerance a mature technology? Yes, in the sense that it is well developed and ripe for exploitation.

Can design fault tolerance be utilised in practical systems? Yes, this has been demonstrated.

Can design fault tolerance improve system reliability? Yes, experiments confirm that a substantial improvement can be achieved.

Is design fault tolerance a cost-effective means of achieving reliability? A firm maybe. This is the crucial question.
References


