SOFTWARE FAULT TOLERANCE

B. RANDELL

Computing Laboratory,
University of Newcastle upon Tyne

Interest has grown in the possibility of obtaining improved system reliability by the provision of means for tolerating not only hardware or operational faults but also residual software design faults. A number of different approaches are being investigated, central to most of which is the idea of structuring a system so as to separate out provisions for automatic backward error recovery. Such provisions are comparatively simple for individual sequential system activities, but are made more difficult by the presence of parallelism. This difficulty is greatly increased in distributed systems, where there is no centralised control of the various parallel activities.

INTRODUCTION:

Although the term chosen as the title of this paper is sometimes, rather misleadingly, taken to mean “software-implemented tolerance of hardware failures”, it is used here to imply an approach to “tolerance of software faults”. Furthermore we use the term “software” rather generally, in fact as a shorthand for logical design, whether such design is encoded as bits in a program store or, say, geometrical patterns on silicon. Thus software fault tolerance is, effectively, design fault tolerance.

The notion of designing a system with the avowed aim of achieving some useful measure of design fault tolerance is fairly novel. The more common (and certainly more “respectable”) approach is to seek means of guaranteeing the absence of design faults. Though such an aim is praiseworthy, the current reality is that operational computing systems of even somewhat modest complexity contain design faults, mainly in the software - which is where most of the logical complexity is to be found. Thus software figures large in the problem of achieving high reliability from complex computing systems (such as large real-time systems) and interest has grown in the subject of software fault tolerance as an adjunct to, but not a substitute for, software fault avoidance.

ERROR DETECTION:

All fault tolerance involves the provision and utilisation of protective redundancy. Error detection, the starting point in any strategy of fault tolerance, involves checking the consistency of redundant information. Such consistency checking is performed during system operation, and complements any consistency checking performed beforehand, for example by a compiler, or by some scheme of manual or automated program verification. A number of different approaches exist to run time consistency checking. There have been rather ad hoc attempts to design redundant algorithms in such a way that no single logical error will cause some particular class of undesirable behaviour to occur [10]. However, the additional complexity that this approach entails is considerable, and surely counterproductive.

A related, but more structured approach, involves the provision of multiple (more-or-less independently designed) algorithms, whose outputs can be checked against each other. This approach, under the title “N-version programming”, which can be described as a software analogue to triple modular redundancy in hardware systems, has been investigated by Chen and Avizienis [7]. Several programming languages provide a language construct for purposes of error detection, namely an assert statement whose Boolean expression if evaluated to false indicates that the program is in error. The paper by Andreas [3] describes an experiment aimed at evaluating the effectiveness of the use of such a technique. The “acceptance test” introduced in [12] is an extension of the idea of an assert statement - acceptance tests allow the programmer to access, in the Boolean expression, both the current and prior values of program variables and so give a convenient form for checking data transformations.

Of the above error detection techniques, N-version programming is the only one which constitutes an overall approach to fault tolerance since it holds the promise of completely masking (single) faults; majority voting is used to identify results which are presumed to be error-free and which can be used as the basis of continued operation. The provision of three or more usefully different versions of a program module, capable in normal circumstances of producing identical results (preferably within similar time periods) is however a daunting task. Thus this technique can hardly be said to share the major virtue, namely the simplicity, of its hardware equivalent, triple modular redundancy, where the
redundancy is provided by the use of multiple components of identical design.

ERROR RECOVERY:

The other error detection techniques discussed above need to be complemented by strategies for automatic error recovery. One can identify two basically different strategies for error recovery, namely forward and backward. Forward error recovery involves continued use of the current (erroneous) state. Backward error recovery involves abandoning the current state, and going back to some earlier state (a so-called "recovery point") which is hoped to be error-free. Backward error recovery presumes that the mechanisms for saving and restoring system state information are reliable, but does not otherwise involve predicting the location and consequences of the fault(s) which gave rise to the detected error. Forward error recovery on the other hand depends critically on the accuracy of such predictions. For this reason it has been argued [15] that automatic forward error recovery techniques such as exception handling, though appropriate for predictable and simple faults such as invalid input data, are quite unsuitable for coping with the effects arising from design faults. These need techniques based on the use of backward error recovery, such as the recovery block scheme [12].

The recovery block scheme can be regarded as a software equivalent to hardware stand-by sparing. On detection of an error by an acceptance test or by any other means, such as an assert statement or a hardware check, the current computation is abandoned, and the system state reset to a previously saved recovery point. A stand-by algorithm (or "alternate") is then invoked, but just for the purpose of dealing with the current set of input data. It afterwards resumes its status as a stand-by spare. The alternate may perhaps have been designed to fulfill exactly the same function as the primary algorithm, though presumably with less efficiency. However it may well serve to provide some minimal service which just enables the computing system to continue in operation. It is our experience that the design of alternates is not a great problem. The scheme seems capable of providing useful improvements in system reliability at moderate cost [1,2], but the problem of providing significantly reliable improvements turns on the difficulty of designing good acceptance tests, a difficulty which is shared by the designers of assertions intended for use by program verifiers.

SYSTEM STRUCTURE:

Backward error recovery schemes such as are utilised in the recovery block scheme are only seen as such because a system is viewed as being structured in such a way as to delineate the state saving mechanism from the rest of the system. In fact system structuring plays a crucial role in fault tolerance, and especially in software fault tolerance [20]. This is particularly the case in systems which give rise to parallel activities. Structure is our principle intellectual tool for mastering complexity - we seek to be able to understand a complex system by viewing it as being constructed from less complex components which have planned interrelationships. However unless the system is designed to constrain, even in the presence of faults, the interactions between components so as to adhere to these planned interrelationships, then this structuring cannot provide a rational basis for provisions for fault tolerance. For example a state-saving mechanism which can easily be corrupted by the rest of the system cannot usefully be thought of as providing backward error recovery. Equally, if possibly erroneous information can flow unimpeded between distinct parallel activities, whether these arise from multiple tasks executed by a single processor, or from multiple interconnected computers, provisions for fault tolerance cannot usefully be based on the activity structure. Efficient and reliable means of tightly constraining erroneous information flow (e.g. [18]) are therefore of considerable value, and can also contribute to the provision of error detection.

PARALLELISM:

Parallelism, such as that produced by multiprocessor systems (which is usually advocated as a contribution to the provision of hardware fault tolerance) introduces a number of difficult problems into the task of achieving software fault tolerance. It introduces new classes of design fault, such as synchronization and deadlock problems, but, more significantly, makes the task of achieving satisfactory error recovery (whether forward or backward) more difficult. Forward error recovery amongst a set of parallel activities involves the provision of supplementary corrective information flow which attempts to compensate for earlier, now suspect, information flow. The implementation of such compensation schemes is rarely attempted except under the more optimistic assumptions regarding the severity of the fault and the extent of subsequent erroneous information flow. Rather it is left as an exercise for the system user! Backward error recovery is of course conceptually much simpler. However careful synchronisation of precautionary state-saving actions (which can severely impact system performance) is needed if the so-called "domino effect" [19] is to be avoided. The domino effect, i.e. repeated state restoration amongst several activities in response to a single error detection, can be avoided by ensuring or, more accurately, optimistically assuming the complete validity of all information received by each
activity [22]. This assumption is in fact typical of data base systems, which take the signalling by the user of completion of an update transaction as his guarantee that the update information is correct. It is just too bad if he later finds that this was not the case.

The signal marking the end of a transaction is a particular case of the important concept of "commitment" [11,20], i.e. the abandonment, either explicitly or implicitly, deliberately or accidentally, of some degree of backward error recovery capability. The simplest commitment act is that of deliberately and explicitly discarding a recovery point. Uncompensatable information flow from one activity to another is a possibly accidental act of commitment on the part of the sender if it has no means of forcing the recipient to go back to a recovery point which pre-dated the receipt of the information.

DISTRIBUTED SYSTEMS:

All of the above discussion has been concerned with, or at least couched in terms of, what has to be regarded as a simple (though important) special case, namely that of systems which are under some central control and hence possess an instantaneously observable global state. A more realistic but, given our present level of understanding, much less tractable view of systems is as giving rise to multiple asynchronous activities lacking any complete scheme of central control, i.e. as being "distributed systems". Such a picture really should be used, explicitly, when one considers even a simple uni-processor computer as being part of a larger system encompassing its environment of users and mechanisms. The computer provides but one activity amongst many - the lack of central control is manifested, for instance, by the independent failure ability of these many activities. The distributed system viewpoint is also obviously necessary when one considers computing systems composed of possibly large numbers of individual computers, interconnected by some form of data communication network.

There has been much work on fault tolerance for computer networks and, more recently, distributed data base systems, aimed mainly against computer and communication faults, and mistakes by users and operators, rather than against residual design faults. Such work has normally been based on predicting the types and immediate consequences of possible faults (e.g. [13]), but not necessarily in great detail and so has often been based on backward error recovery in the individual computers, so that lost messages can be retransmitted, or corrupted files reconstructed. However the lack of any central control means that one computer can be continuing to use information that another computer has already found to be erroneous. Moreover tasks such as that of establishing consistent (system-wide) recovery points, and of determining when it can be presumed to be safe to discard a recovery point require very intricate synchronization schemes. These must be capable of coping with sudden loss of communication or processing facilities, and are a subject of considerable current research (e.g. [6,11,16,21]), much of which stems from the pioneering work of Davies and Bjork [5,6].

One thing that is clear, is that, both for reasons of reducing the incidence of residual design faults and in order to be able to introduce a credible degree of fault tolerance, perhaps even of design faults, improved methods of structuring distributed computing systems and the supporting system software are necessary if large scale general purpose distributed systems are going to achieve really high levels of reliability. Our own efforts in this direction are based on trying to extend ideas on levels of abstraction and atomic actions [2,4,20].

CONCLUSIONS:

How then can we summarize the role of software, i.e. of design, fault tolerance? What contribution can we expect it to make to system reliability? One thing is certain; at present such questions are not amenable quantitatively through reliability modelling and prediction. Design fault statistics are the statistics of human frailty. There are now claims that such statistics can be of value in predicting the amount of further effort required to validate (i.e. debug!) a large program to some (unquantified) degree of completeness [17]. However this is a far cry from obtaining credible estimates of the mean time between failures of a system controlled by such a program, since the testing will not have been exhaustive, nor guaranteed to have been fully representative of all possible operational conditions. This is the case whether or not the program contains measures aimed at achieving a degree of software fault tolerance. We do not have the statistics, or, more importantly, the models needed for such estimations. Rather we must appeal to qualitative arguments in favor of software fault tolerance, such as those put forward by Melliar-Smith [14].

Software verification, testing and fault tolerance are each based on certain assumptions, e.g. for verification that a sound proof can be produced, for testing that sufficient test cases can be tried, etc. Melliar-Smith points out that the three approaches have but a single assumption in common, namely that the system specification is correct, and that otherwise each approach has some unique merits, so that the careful use of all three together provides the best chance of high reliability.
This then is as far as we can go at present - software fault tolerance, though still at a very early stage of development, shows promise of enabling us to obtain increased levels of overall reliability from complex computer systems, but is at best a supplement to, rather than a substitute for, attempts of software fault avoidance.

REFERENCES:


[14] Melliar-Smith, P.M., If We Have Program Proof, Why Do We Need Software Fault Tolerance, Presentation at IEEE CS/FTC Workshop on Designing for the Unexpected, St. Thomas (Dec. 1978).


