SOFTWARE RELIABILITY: THE ROLE OF PROGRAMMED EXCEPTION HANDLING

P.M. Melliar-Smith*

B. Randell

Computing Laboratory, The University, Newcastle upon Tyne, U.K.

September 1976

The paper discusses the basic concepts underlying the issue of software reliability, and argues that programmed exception handling is inappropriate for dealing with suspected software errors. Instead it is shown, using an example program, how exception handling can be combined with the recovery block structure. The result is to improve the effectiveness with which problems due to anticipated faulty input data, hardware components, etc., are dealt with, while continuing to provide means for recovering from unanticipated faults, including ones due to residual software design errors.

1. Introduction

Discussions of software reliability are frequently marred by misunderstandings arising from incompatible preconceptions and terminology - for example some people have equated the terms 'software reliability' and 'program correctness' while others have assumed that 'software reliability' encompasses such concerns as the design of appropriate forms of system response to invalid input data.

The purpose of the present paper is twofold - to propose a set of terms and their definitions which might obviate further misunderstandings, and to discuss the relevance of programmed 'exception handling' to the problem of coping with residual design errors (or 'bugs') in programs.

Our informal, but hopefully precise, definitions are based closely on those given in [5]. To avoid needless specialisation the terminology is defined in general terms, and is not specific to computer programs. Rather it is relevant to all types of system, hardware as well as software. The terminology we use is intended to correspond broadly to conventional usage, but the definitions of some of the terms differ from previous practice, which typically has paid little attention to design inadequacies as a potential source of unreliability.

2. Systems and their Failures

We define a system as a set of components together with their interrelationships, which system has been designed to provide a specified service. The components of the system are themselves systems, and we term their interrelationships the algorithm of the system. There is no requirement that a component provide service to a single system; it may be a component of several distinct systems. The algorithm of the system is however specific to each system individually.

This definition of 'system' with its insistence that the service provided must be specified (but not necessarily prespecified), is intended to exclude systems which are "intelligent" in the sense of being capable of determining their own goals and algorithms. At present intelligent systems are not understood sufficiently to permit consideration of their reliability.

The internal state of a system is the aggregation of the external states of all its components. The external state of a system is the result of a conceptual abstraction function applied to its internal state. During a transition from one external state to another external state, the system may pass through a number of internal states for which the abstraction function, and hence the external state, are not defined. The specification defines only the external states of the system, the operations that can be applied to the system, the results of these operations, and the transitions between external states caused by these operations, the internal states being inaccessible from outside the system.

---

* Present Address: Stanford Research Institute, Menlo Park Calif., U.S.A.

The service provided by a system is regarded as being provided to one or more environments. Within a particular system, the environment of a given component consists of those other components with which it is directly interrelated.

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. There is however no implication that the event is actually recognised as having occurred. For example, if an environment does not make full use of the specifications of a system (i.e. if what Parnas [6] terms the environment's 'assumptions' are a proper subset of the specifications) certain types of failures will have no effect.

3. Errors and Faults

In contrast to the simple, albeit very broad, definition of 'failure' given above, the definitions we now present of 'error' and 'fault' are not so straightforward. This is because they aim to capture the element of subjective judgement which we believe is a necessary aspect of these concepts, particularly when they relate to problems which could have been caused by design inadequacies in the algorithm of a system.

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 subjective judgement that we wish to associate with the classification of a state as being an erroneous one derives from the use of the phrases "normal algorithms" and "which we do not attribute" in this definition - however further definitions are required before these matters can be discussed properly.)

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, 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.

Hopefully it will now be clear that the generality of our definitions of failure and fault has the intended effect that the notion of fault encompasses such design inadequacies as a mistaken choice of component, a misunderstood or inadequate specification (of either the component, or the service required from the system) or an incorrect interrelationship amongst components (such as a wrong or missing interconnection, in the use of hardware systems, or a program bug in software systems), as well as, say, hardware component failure due to ageing.

Note that the definition of an erroneous state depends on the subdivision of the algorithm of the system into normal algorithms and abnormal algorithms. These abnormal algorithms will typically be the error recovery algorithms. There are many systems in which that subdivision, and hence the designation of states as erroneous, is a matter of judgement.

For example, in a storage system utilising a Hamming Code, one may regard the correction circuits as error recovery mechanisms and a single incorrect bit as an error. Alternatively (particularly with semiconductor storage) the correction circuits may be regarded as normal mechanism, and thus a single incorrect bit would not be regarded as an error, though two incorrect bits would be.

Note also that a demonstration that further processing can lead to a failure of the system indicates the presence of an error, but does not suffice to locate a specific item of information as the error. Consider a system affected by an algorithmic fault. The sequence of internal states adopted by this system will diverge from that of the "correct" system at some point, the algorithmic fault being the cause of this transition into an erroneous state. But there can be no unique correct algorithm. It may be that any one of several changes to the algorithms of the system could have precluded the failure. A subjective judgement as to which of these algorithms is the intended algorithm determines the fault, the items of information in error, and the moment at which the state becomes erroneous. Some such judgements may of course be more useful than others.

The significance of the distinction between faults and errors may be seen by considering the repair of a data base system. Repair of a fault may consist of the replacement of a failing program (or hardware) component by a correctly functioning one. Repair of an error requires that the information in the data base be changed from its currently erroneous state to a state which will permit the correct operation of the system. In most systems, recovery from errors is required, but repair of the faults which cause these errors although very desirable is not necessarily essential for continued operation.

4. Fault-Tolerant Computing Systems

A system can be designed to be fault-tolerant by incorporating into it abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures. The degree of fault-tolerance will depend on the success with which erroneous states corresponding to likely faults are identified and detected, and with which such states are repaired.

The software of a computing system serves to structure that system by expressing how some of the storage locations are to be set up with information which represents programs. These will then control some of the interrelationships amongst hardware components, for example, that the potential communication path between two I/O devices via working store is actually usable. Such

software can of course be designed so that the computing system as a whole is tolerant of faults due to certain types of hardware failures.

However the software can itself be viewed as a system, and its components and their interrelationships discussed in terms of the programming language that was used to construct it. Thus in a block-structured language each block can be regarded as a component, which is itself composed out of, and expresses the interrelationships amongst, smaller components such as declarations and statements (including blocks).

The only faults that can be present in a non-physical system such as a software system are algorithmic faults. However from the earlier discussion of such faults, it will be seen that the term covers much more than 'conventional' program bugs.

Algorithmic faults arise from unmastered design complexity, and can of course exist in the hardware as well as the software of a computing system. However due to such matters as the differing relative costs of modifications to hardware and to software, it is traditional for very complex design issues to be relegated to the software area, where they all too often give rise to algorithmic faults.

The idea of attempting to design computing systems which can tolerate algorithmic as well as mechanical faults is fairly novel. There is a tendency to assume that delivered hardware is free from algorithmic faults, and that what is needed is a means of ensuring that the software is also free from such faults - research to this end includes that on formal specification and validation of programs, and on methodologies for program testing and debugging. (Incidentally, the general assumption that hardware designs are correct may well not survive for much longer, given the ever increasing complexity of function that is being incorporated into a single LSI chip:)

In our research at the University of Newcastle upon Tyne on system reliability [1,4,5,8] we have adopted as a basic premise that all large scale computing systems at all times contain multiple potential faults, and that these will include algorithmic as well as mechanical ones. There will be faults in the hardware, in the peripherals and in the operating system, in the logic design and in the hardware components, in the basic systems design, in the applications programs and in the information stored; in the actions of the operations staff and the maintenance engineers; and in the environment of the computer system. These faults may be due to the wearing out of a component, to a design inadequacy, to a statistical uncertainty (noise), to human frailty, or to an evolution of the requirements on the system as yet unmatched in the implementation.

We have been investigating the practicability of incorporating abnormal algorithms into such computing systems in order that all such types of fault can be tolerated, so that a system can provide continuous and trustworthy service to its environment without the need for any human intervention. Some of these types of fault are susceptible to existing techniques for error detection and recovery; others are not, particularly those faults of design inadequacy. Many approaches to reliable operation depend on the correct design of the system, together with complete knowledge of the possible failure modes of the components of the system. In contrast, we have chosen to investigate techniques which do not assume the absolute correctness of the algorithms. Moreover, since the number of possible failure modes of a component increases very rapidly as the component becomes more complex, much more rapidly than the number of correct modes of operation, we have felt it impracticable to rely on enumerating the possible failure modes of components, let alone design algorithms to detect or accommodate each possible component failure mode individually.

Thus the techniques of fault-tolerant system design that we have been developing, such as recovery blocks and conversations [8], do not assume correct algorithms or make any assumptions about the nature of faults. They aim to provide error detection and recovery strategies which should be applicable whenever a system fails to provide its specified service, for whatever reason.

These techniques do not attempt to diagnose the fault responsible for the errors which are detected, or to repair such faults. The error symptoms of a residual fault may be obscure and misleading, while the correct diagnosis and repair is not necessarily unique. Consequently we regard diagnosis and repair as operations to be performed off-line, and requiring human intelligence.

We do however assume that the faults to be recovered from are those residual faults remaining after reasonable efforts to obtain a reliable system. In particular we assume that the software has been designed as well as possible, using well-chosen design methodologies, together with validation techniques such as formal proofs of correctness and systematic testing. It can of course be argued that such validation techniques, which using Azivienis' terminology [2] could be described as the method of "software fault intolerance", are more productive of software reliability than attempts such as ours at software fault tolerance. Our view is that each has its place.

The argument for this viewpoint is not solely that of disbelief in the completeness with which a complex software system can be validated. Rather, it also concerns the significance that can be attributed to the experience one obtains from using such a system. Extensive usage of a hardware system whose failures are caused by faults arising from component ageing and the like provides statistics which can be a useful predictor of the likely frequency and seriousness of further failures. In contrast, the statistics gathered of failures of a software system relate merely to the history of its modification and usage (i.e. the particular sets of input data, the relative timings of input activities, etc.). Over the life of a system of any complexity whatsoever (e.g. a 64-bit multiplier), only an infinitesimal proportion of the possible uses can actually occur. Thus, other than to the extent that future use will exactly match past use, failure statistics from a complex software system are not a useful predictor of the frequency of further

failures. More importantly, particularly if the designer has relied totally on software fault intolerance, these statistics will not even predict the possible seriousness of further failures.

5. Exceptions

Just as we do not regard our techniques for tolerating algorithmic faults as a substitute for efforts to reduce the incidence of such faults in a system, so also do we not regard them as a complete substitute for explicit recovery from errors caused by anticipated faults. In a software system the sections of the program text that relate to such explicit recovery actions are sometimes termed "exception handlers". However the concept of an "exception" (as for example, described by Goodenough [3] is by no means necessarily limited to such use, and is indeed quite separate from our concepts of error, fault and failure as the following discussion and definitions attempt to make clear.

The specified service that a component of a system is designed to provide might include activities of widely differing value to its environments. No matter how undesirable, none that fall within the specifications will be termed failures. However the specification can be structured so as to differentiate between a standard service, and zero or more exceptional services. For example, the standard service to be provided by an adder would be to return the sum of its inputs, exceptional services to indicate that an arithmetic overflow has occurred, or that an input had incorrect parity.

Within a system, a particular exception is said to have occurred when a component explicitly provides the corresponding exceptional service. The algorithm of the system can be made to reflect these potential occurrences by incorporating exception handlers for each exception.

These definitions match the intent, but not the form of the definitions given by Goodenough, who states:

"Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker ... In essence, exceptions permit the user of an operation to extend an operation's domain (the set of inputs for which effects are defined) or its range (the effects obtained when certain inputs are processed)."

However, in contrast to Goodenough, we have taken care to avoid the use of the word 'failure' in discussing exceptions. This is not mere pedantry. Rather it is a consequence of the very basic view we take of failures, namely as occuring when and only when a system or component does not perform as specified. Although a system designer might choose to treat certain exceptions as component failures (which he might or might not provide abnormal algorithms to deal with), we regard the various schemes for exception handling (e.g. Parnas [7], Goodenough [3] and Wasserman [9]) and our technique of recovery blocks as complementary rather than competitive.

A basic feature of the recovery block scheme is that, because no attempt is made to diagnose the particular fault that caused an error, or to assess the extent of any other damage the fault may have caused, recovery actions have to start by returning the system to a prior state, which it is hoped precedes the introduction of the error, before calling an alternate block. Should this prior state not precede the introduction of the error, more global error detection, and more drastic error recovery, is likely to occur later. (The associated 'recovery cache' mechanisms [1,4] automate the state saving required for this scheme.)

When exceptions are treated as component failures in a software system that uses recovery blocks, they will lead to the system being backed up to a prior state and an alternate block being called. This will be appropriate when the exception is undesirable, and the system designer does not wish to provide an individual means of dealing with it.

Putting this the other way, exceptions can be introduced into the structure of a system which uses recovery blocks, in order to cause some of what would otherwise be regarded as component failures (leading to automatic back-up) to be treated as part of the normal algorithm of the system, by whatever explicit mechanisms the designer wishes to introduce for this purpose. Failures might of course still occur, in either the main part of the algorithm, or in any of the exception handlers, and if they do they will lead to automatic back-up. Such introduction of exceptions can therefore be thought of as a way of dealing with special or frequently occurring types of failure, in the knowledge that the recovery block structure remains available as a "back-stop".

However we would argue strongly against relying on exception handling as a means of dealing with algorithmic faults. Programmed exception handling involves predicting faults and their consequences, and providing pre-designed means of on-line fault diagnosis. Thus although it can be of value in dealing with foreseen undesirable behaviour by hardware components, users, operations staff, etc., it is surely not appropriate for dealing with software faults - predictable software faults should be removed rather than tolerated. Indeed the incorporation of programmed exception handlers to deal with likely software faults would in all probability, because of the extra complexity it would add to the software, be the cause of introducing further faults, rather than a means of coping with those that already exist. On the other hand when used appropriately for anticipated faults of other types they can provide a useful means of simplifying the overall structure of the software, and hence contribute to reducing the incidence of residual design faults.

As described in [8], the recovery block scheme can be applied to any programming language in which a program which is structured into blocks evokes a process which can be regarded as structured into operations, where the acts of entering and leaving each operation are explicit, and are properly nested in time. The scheme does not depend on the particular form of block structuring that is used, or the rules governing the scopes of variables, methods of parameter passing, etc. Thus there is no particular

difficulty in combining the scheme with programming language facilities for exceptions and exception handlers. By way of illustration, an example program which uses recovery blocks and procedure-oriented exception handling [9] is given in the Appendix.

Conclusions

Exception handling is one of many programming language design issues which is the subject of current debate, in this case a debate which has not been helped by a lack of agreement on a terminology for discussing the various basic issues concerning system and software reliability. The aim of this paper has been to clarify these issues, and to argue that despite the views that have been argued to the contrary elsewhere (eg. Parnas [7] and Wasserman [9]),explicit exception handling is not an appropriate means for providing software fault tolerance. Instead we view it as a potentially valuable adjunct to any viable scheme for detecting and recovering from software errors, which could improve the effectiveness with which anticipated faults due to input data, operators, hardware components, and the like were dealt with.

Acknowledgements

The writing of this paper has been aided by numerous discussions with colleagues on the reliability project at the University of Newcastle upon Tyne, and in the Computer Systems Research Group at the University of Toronto, and also at meetings of the IFIP Working Group 2.3 on Programming Methodology.

References

1. T. Anderson and R. Kerr. Recovery Blocks in Action: a system supporting high reliability. Proc. Int. Conf. on Software Engineering, San Francisco (13-15 Oct. 1976).

2. A. Avizienis. Fault-Tolerance and FaultTolerance: Complementary Approaches to Reliable Computing. Proc. Int. Conf. on Reliable Software. Sigplan Notices 10,6 (June 1975) pp. 458-464.

3. J.B. Goodenough. Exception Handling: Issues and a Proposed Notation. Comm. ACM 18,12 (1975) pp. 683-696.

4. J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A Program Structure for Error Detection and Recovery. Operating Systems (E. Gelenbe and C. Kaiser: Eds.) Lecture Notes in Computer Science, Vol. 16. Springer Verlag, New York (1974) pp. 177-193.

5. P.M. Melliar-Smith. Error Recovery for Resource Managers. Technical Report, Computing Laboratory, University of Newcastle upon Tyne (in preparation).

6. D.L. Parnas. Information Distribution Aspects of Design Methodology. Proc. IFIP Congress 1971 pp. TA26-30.

7. D.L. Parnas. Response to Detected Errors in Well-Structured Programs. Dept. of Computer Science, Carnegie-Mellon Univ. (July 1972).

8. B. Randell. System Structure for Software Fault Tolerance. IEEE Trans. on Software Engineering SE-1,2 (June 1975) pp. 220-232.

9. A.I. Wasserman. Procedure-Oriented Exception Handling. Medical Information Science, University of California, San Francisco (1976).

Appendix

Figure 1 shows a section of program text which incorporates programmed exception handling within a recovery block structure. The example, and the form of exception handling shown, are based on that given by Wasserman [9].

The basic form of the example is

```
ensure consistent_inventory
by process_updates
else by refuse_updates
else error
```

The implicit assumption is that the program is maintaining an inventory file whose consistency is to be checked after each related sequence of updates, to determine whether this sequence can be incorporated. The updating process uses the procedure 'checknum' to read and check the updates. This procedure provides an exception handler for some of the exceptions that can be raised by the 'read' routine, so that the person providing the inputs can have two chances of correcting each input.

The procedure 'checknum' is taken directly from Wasserman [9], but has been simplified to take account of error recovery facilities provided by the recovery block structure in which it is used. More detailed notes on the example follow.

Line 2 The Boolean expression 'consistent inventory' will be evaluated if and when 'process_updates' reaches its final 'end'. If the expression is true, the alternate block 'refuse_updates' will be ignored and the information stored by the underlying recovery cache mechanism, in case the effects of 'process_updates' had to be undone, will be discarded. Otherwise this information will be used to nullify these effects, before 'refuse_updates' is called, after which the Boolean expression 'consistent_inventory' is checked again.

Line 4 In Wasserman's scheme a group of separate exceptions can be gathered together, as here to define the exception 'goof', using the exceptions 'overflow', 'underflow' or 'conversion'. It is assumed that all three can be signalled by the routine 'read' - the first two perhaps being built-in exceptions that the hardware signals, the third being implemented by the routine 'read' itself.

Line 7 The procedure 'message' is an exception handler defined within 'checknum'. The first two occasions on which it is called it used Wasserman's scheme for retrying the procedure which raised the exception (see line 14), but on the next occasion it signals error. (In Wasserman's version of this routine, 'message' raised the special exception called 'fail' which caused the whole program to be aborted. Here we assume that error just causes the current alternate block to be abandoned .)

Line 18 Here 'checknum' calls 'read' and arranges that the exception 'goof' (i.e. the exceptions 'overflow', 'underflow' or 'conversion') will be handled by the procedure 'message', but that if

'read' signals 'ioerror' this will cause 'proceSS_updates' to be abandoned. In the original version of the example a further exception handler was provided, for use when 'ioerror' was signalled. This exception handler indicated that, but did not explain how, "any required cleanup" was to be 'done.

Line 20 All that is illustrated of the main body of 'process_updates' is that it counts the number of updates, which it reads and checks using the routine 'checknum'.

Line 32 The second alternate block 'refuse_updates' is called if the first alternate block 'process_updates' abandons its task, or fails to pass the acceptance test, for any reason (including of course, any residual design error within its code). If this happens, all changes that 'process update' has made to the inventory will be undone, and the integer 'update_no' will be reset. This integer is then used for an apologetic message to the user.

```
 1
 2 ensure consistent_inventory by
 3 process_updates: begin integer num;
 4            exception goof = overflow or underflow or conversion;
 5            procedure checknum (integer j);
 6                        global integer count = 0;
 7                        procedure message;
 8                              begin count := count +1;
 9                                    write ('please try again");
10                                    if count> 3 then
11                                          begin write ("three strikes-
                                                                you're out);
12                                                signal error
13                                          end
14                                          else retry;
15                              end message;
16                        begin /* body of checknum */
17                              ...
18                              read (j) [goof: message, ioerr:error]
19                        end checknum;
20                        begin /* Start of main body */
21                              ...
22                              while updates_remain do
23                                    begin update_no := update_no + 1;
24                                          ...
25                                          checknum(num);
26                                    ...
27                                    end
28                              ...
29                        end main body
30                  end process_updates
31 else by
32 refuse_updates:  begin write ("sorry - last update accepted was number");
33                              write (update_no)
34                  end
35 else error
36 ...
```

Figure 1 AN EXAMPLE OF A PROGRAM WHICH INCORPORATES PROGRAMMED EXCEPTION HANDLING WITHIN A RECOVERY BLOCK STRUCTURE