A TOLERANT APPROACH TO FAULTS

DRAFT OF 2ND JUNE 2011

Michael Jackson

Newcastle University
jacksonma@acm.org

1 INTRODUCTION

In the first half of the 1970s important work was initiated at Newcastle [Randell71] on improving the reliability of computer systems. This work resulted in the recovery block concept for software fault tolerance [Horning+74, Randell75] and in associated fruitful techniques including the recursive cache for error recovery. This short paper discusses one aspect of an approach to structuring system behaviour in the large. In very broad terms, the approach can be regarded as an expansion and generalisation of the basic recovery block concept and of the use of error detection to improve reliability. It addresses behaviours rather than terminating procedures; it embraces the complete system, including both the computer and the problem world, rather than the software alone; and it generalises the notion of error to include any condition contradicting the current design assumptions. The resulting structures are abstract, in the sense that in general they do not correspond closely to implemented software architectures. The source of this structural mismatch is identified, and a justification offered for the utility of multiple structures in software development.

2 RECOVERY BLOCKS

The structure of a recovery block is:

```
ensure acceptance-test
    by alternate-block-0
    else by alternate-block-1
    else by alternate-block-2
    else by ...
    else error
```

in which the alternate blocks are terminating procedures and the acceptance test is a logical expression or a boolean function without side effects. The purpose of the structure is to improve reliability by tolerating software errors—that is, errors of software design or implementation. In the absence of error, execution of the whole recovery block consists in successful execution of the primary block `alternate-block-0` alone, followed by a successful check that its post-state satisfies the acceptance test.

Error in the primary block may be manifested in two ways: either by reaching an internal error state—for example, by attempting a division by zero or triggering a timeout—or by completing execution in a post-state for which the acceptance test fails. In either case, the state of the computation is then restored to the state on entry to the recovery block. Successive alternate blocks, if any, are then executed in similar fashion, until either one completes successfully or all have failed and the final error clause is reached indicating failure of the whole recovery block. Since an alternate block may itself be a recovery block, the scheme is fully recursive, and the recursive cache provides for acceptably economical saving of state and its restoration in the event of error at any level.

There are several possible reasons why an alternate block may succeed where the preceding block has failed. The failing block may have a purely internal design error. For example, if invoked with a particular argument value it may mismanage a data structure local to the block itself, causing a null pointer dereference or an infinite loop, or use of an inappropriate structure element. The later, successful, block may succeed simply because its design is different. It may have its own, different, internal design errors, but they may remain latent in the circumstance that caused the preceding block to fail. This is in effect an application of *n-version* programming. Another possible reason is that successive alternates are less ambitious and therefore simpler and less likely to contain design errors.

Randell gives [Randell75] the example of a recovery block for sorting an array that tries successively *quickersort*, *quicksort* and *bubblesort*. Another example of reduced ambition is the use of a less precise but more reliable algorithm for computing a real value.

The recovery block has only one acceptance test, which evaluated after each attempted alternate block execution. The test must therefore be applied to the variables accessible to the program that invokes the recovery block, not to the local variables, which may vary from alternate to alternate. It must be economical of resources, and is not therefore a full test of the post-condition of the recovery block. The designer must decide [Randell75] the appropriate level of rigour of the test.

3 ENLARGING THE SCOPE

In the original conception of recovery blocks it was clearly recognised that software errors could arise not only from faults of logic in program design but also from faults attributable to the software's environment. Quite apart from the possibilities of program failure due to computer hardware faults, other processes interacting with the designed software might depart from an agreed synchronization discipline or might introduce erroneous data values or corrupt data structures, infringing agreed invariant properties of shared data. In understanding a failure of any invoked software component we may always ask whether the failure is more properly attributable to the component itself or to the environment in which it is invoked. Has the software design failed to conform to its intended explicit—or implicit—specification? Or has the environment failed to conform to the software designer's legitimate assumptions? In recovery blocks, the emphasis is strictly on error detection and recovery. In pursuit of reliability, error diagnosis and attribution are regarded as secondary considerations.

In a realistic computer-based system—especially one whose reliability can be regarded as critical—the scope for software failure, however attributed, is writ large. The requirements and purposes of such a system are located in the *problem world*: that is, in the heterogeneous and variegated parts of the physical and human world outside the computer from which the computer draws its inputs and in which it is required to produce its effects. The environment of the designed software is the whole of the problem world: it includes the human operators, everything that is directly interfaced to the computer, everything mentioned in the system requirements, and everything lying on the causal chains between them. Few of these parts of the problem world have been explicitly engineered to satisfy an explicit specification. Even those that have—for example, the electromechanical equipment of a car, a chemical plant or a lift—cannot be perfectly reliable, and in a critical system provision must be made for maintaining safety in the event of their failure.

In the recovery block work it was already recognised that the need for reliability extended far beyond the software. The paper *System Structure for Software Fault Tolerance* [Randell75] includes this observation:

> "Indeed as a general approach to the structuring of a complex activity where the possibility of errors is to be considered, there seems to be no *a priori* reason why the structuring should not extend past the confines of the computer system. Thus, as others have previously remarked ..., the structuring could apply to the environment and perhaps even the activity of the people surrounding the computer system. The effectiveness of this approach to fault-tolerant system design will depend critically on the acceptance tests and additional alternate blocks that are provided."

Two interesting pieces of work in this eminently realistic spirit, from ten and twenty years after publication of the recovery blocks work, deserve mention. One is the use of 'audit programs' in AT&T's 5ESS telephone switch [Haugk+85]. Telephone call processing must deal with a problem world in which race conditions are common. Users may perform many unexpected and perverse sequences of actions. There are many call processing features and many interactions among them. Unsurprisingly, call processing software exhibited many errors. Execution of audit programs is interleaved with call-processing, to check the integrity of software data structures. If an error is detected, recovery may take various forms, including the forcible termination of a call in progress to prevent propagation of the error. The nature of the call processing function makes this draconian approach acceptable: only a negligible proportion of calls are terminated in this way. The availability

of the call-processing service is hugely improved, to the benefit of current and new calls, including, of course, a subscriber's likely retry of a call inexplicably terminated by the audit program.

A very different piece of work, reported in 1995, is the work on requirements monitoring [Fickas+95]. The idea here is to add software to a running system to determine how well the system is meeting its requirements. One example case study is a system to manage software licenses, allowing workers in an organisation to use as many duplicate copies of a piece of software as the organisation has bought licences, but no more. The added software monitors this requirement, together with other requirements such as fair and convenient use by the workers, reliable delivery of requested licenses, and acceptable consumption of system resources by the licence manager. The work differs from recovery blocks and from the 5ESS audit programs most significantly in addressing a very long time scale. The monitoring is not designed to improve reliability of the system in current operation by triggering immediate recovery action. Instead, its outputs are intended to be used offline, studied by developers and maintainers who will tune and redesign successive versions of the evolving system to correct design errors and adapt it to changing patterns of use.

4 COMPUTER-BASED SYSTEMS

The complexity of realistic computer-based systems does not only spring from the heterogeneous richness of their problem worlds. Such a system presents three aspects that can motivate a generalisation and enlargement of the original insights of recovery blocks.

First, the problem world is, in general, persistent. Its parts—or *domains*—have long lives, and over those lives they engage in continuing behaviours, interacting with the computer and with other, neighbouring, parts of the problem world. To understand and manage these interactions, developers must be explicitly concerned not only with the designed behaviour of the software but also with the given properties and behaviours of the problem world. In software terms, this means that the most important components are not terminating procedures but long-running processes. The essential criterion of success of most software components is not the achievement of a post-state satisfying a specified post-condition, but the continuing execution of a process that monitors, controls and evokes a desired continuing behaviour in the problem world. For understanding and analysing the system function and requirements, it is these desired continuing behaviours that should be the main subjects of descriptions and models of the problem world.

Second, the notion of failure itself can be broadened to a concept that can be usefully applied to the large structure of a system. The recovery block scheme is based on terminating procedures, the correctness of whose results can be evaluated in the post-state alone. If the result of executing one procedure or, alternate block, is incorrect, it makes excellent sense to restore the pre-state and attempt to achieve a correct post-state by a different procedure. If the fault lies in the current invocation of the recovery block by the environment, it is reasonable to hope that the next invocation will not exhibit the same fault; and if the fault lies in the design of the alternate software block that has failed, it is reasonable to hope that the next alternate block will be more successful. The failure, whether attributed to the software or to its environment, is taken to be evidence of an exceptional condition that will probably not recur. For this reason it is eminently sensible to place little or no emphasis on diagnosing the cause of the exception, and to proceed straight to recovery and retry.

This argument is far less compelling in a computer-based system. Certainly there are transient errors, and means must be provided to detect them and recover from their effects. But often persistent errors will be both more common and more significant. For example, the characteristics of engineered problem domains of most kinds are liable to change over time. Occurrence of one failure suggests that a physical change has taken place that will probably cause the failure to be repeated. A relay that has stuck once is likely to stick again; a motor that is running slow is not likely to recover its full performance; a power supply that has failed is unlikely to be reinstated. Occurrence of such a failure may still allow system operation to continue, but to continue in a different mode in which only a degraded behaviour can be supported in the new context. In the original context it was assumed that the problem world was fault-free, and on that assumption the full system function was provided. Now the failure occurrence has shown that the assumption no longer holds, and henceforth a different behaviour is required, in a different context with different assumptions.

Third, the idea that different system behaviours, and hence different operational modes of the software, are to be associated with different problem world contexts, is clearly more generally applicable than solely as a response to problem world failures. At every level, a realistic computer-based system has many overlapping modes of operation, each requiring the software to permit, monitor or evoke related behaviours in the problem world. Even in the absence of failures, then, operation of the system can be seen as a complex structure of related contexts and behaviours. Departure from the current context or subcontext will sometimes be due, not to a failure in the software or the problem world, but simply to an event, a command, or a condition that signals a need to change the set of currently active modes.

5 CONTEXTS, PROPERTIES AND FAILURES

A simple example may be drawn from the operation of a passenger lift. A user wishing to travel from one floor to another requests lift service by pressing a button in the lift lobby to summon the lift, and another button in the lift car to request travel to the desired floor. The normal operation of the system provides lift service, carrying users from floor to floor in accordance with their wishes. On arrival at a floor where a user is to enter or leave the lift car, the lift car and lobby doors are opened, enough time is allowed for users to enter and leave, and the doors are closed.

This normal lift service is achieved by a cooperative behaviour of the users, the lift equipment, and the controlling software. The system should be designed to provide a satisfactory experience for the users. The time allowed between opening and closing the doors must be enough for the actual population of users in the operational context: more time is needed in a retirement home or an orthopedic hospital. The users must play their part: service quality can be spoiled by users who hold the doors open by placing a heavy suitcase between them, or by a mischievous teenager who spends an hour in the lift car pressing every request button whenever the lift reverses direction.

The possible failures of the lift equipment must also be considered. Provision of normal lift service depends on the control software interacting appropriately with the electromechanical equipment. To move the lift from floor 1 to floor 3, for example, the software must set the motor polarity to *up*, set the motor relay to *on*, wait until the sensor at floor 3 is *on*, and set the motor relay to *off*. The lift equipment must behave appropriately in response: when the motor relay is set to *on* the lift car must start to rise; when the car reaches floor 3 the sensor state must change from *off* to *on*; and when the motor relay is set to *off* the lift car must come to rest.

However, the motor may fail to start because its power supply has been interrupted, or the motor has burned out, or the motor relay has failed. The lift car may fail to rise because the hoist cable has broken. The floor 3 sensor may be stuck on. The motor may fail to stop because the motor relay is stuck on. Because a lift system is safety-critical, even these unlikely failures must be detected and handled.

The approach to failure detection and handling that was suggested in the preceding section is based on a separation of concerns—or, rather, a separation of contexts. Considering the lift equipment, we distinguish the *healthy* context from the *faulty* context. Accordingly, we identify three behaviours to be supported by the software. First, the provision of normal lift service. This is appropriate only in the *healthy* context, and is to be designed on the associated assumption that the equipment is not faulty. Second, the preservation of safety in the presence of equipment faults. This is appropriate only in the enclosing *faulty* context, and is to be designed on the associated assumption that the equipment is faulty. Third, the monitoring of the equipment to detect and report the presence of a fault. This is appropriate in the enclosing *healthy-or-faulty* context. In addition, there must be software to implement a control structure to manage the three behaviours. It must ensure that initially lift service and fault monitoring are active; then, if a fault is reported, it must pre-emptively terminate lift service and activate the safety preservation behaviour.

By separating the two contexts, this approach clarifies design of the software for normal lift service behaviour. This designer is relieved of the obligation to check continually that the *healthy* context is in force, and is entitled to rely on the assumptions explicitly associated with that context. The designer of the software for preservation of safety is similarly relieved of the obligation to check that the *faulty* context is in force, and is entitled to rely on the assumptions explicitly associated with that context. The designer of the software for fault monitoring is relieved of the obligation, when a fault is

detected, to take any action beyond faithfully reporting the fact. All three are relieved of the responsibility to pass control to another behaviour. In the spirit of the recovery block structure, this is the responsibility of the control structure software.

The strength and importance of the assumptions in the *healthy* context is clear. The assumptions in the *faulty* context and *healthy-or-faulty* context must also be strong and must also be made explicit. In the *faulty* context, the safety preservation behaviour must rely on whatever equipment properties are necessary to the action it takes. For example, if it applies the emergency brake to lock the lift car in the shaft it is relying on the healthy functioning of the brake. In the *healthy-or-faulty* context, the fault monitoring behaviour is relying on the assumption that particular faults will exhibit certain symptoms. The *healthy* and *faulty* contexts must be related by the implication $\neg$*faulty*$\Rightarrow$ *healthy*, ensuring that the normal lift service behaviour is active only when its assumptions are known to hold.

The approach can be readily extended to handle faults at more than one level. For example, the fault monitoring report might distinguish between major and minor equipment faults. A major fault must be handled by applying the emergency brake, trapping passengers in the lift car until they can be rescued by a maintenance engineer. A minor fault, such as a stuck sensor, may perhaps be handled by moving the lift car to the ground floor—or to the floor nearest to the ground with a working floor sensor—and opening the doors to release the passengers. In the case of a major fault lift service is abandoned. A minor fault may permit service subject to some form of restriction such as withdrawal of service from a floor with a stuck sensor. The careful separation of contexts allows for the proper treatment of a major fault that occurs while a minor fault is being handled.

Analogies can be drawn between the separation of concerns in this approach and the separation achieved by the recovery block scheme. A recovery block separates the procedures that can work in different contexts; it specifies a priority ordering between them in which a procedure with more desirable specified effects is tried before a less desirable; it treats the monitoring of context as a procedure (more strictly, a function) in its own right; and it clearly separates the control structure from the procedures it controls.

Of course, there are also clear differences. ‘Monitoring the context’ in a recovery block is really monitoring the effect of the most recently attempted procedure: no attempt is made to identify the deviation of the context from the programmer’s implicit assumptions. Saving and restoring the computational state is integral to the scheme, and is both possible and valuable for a terminating procedure whose desired result can be expressed in a testable postcondition; but it makes little sense for an aborted continuing behaviour whose side-effects in the problem world have typically been completely beneficent and, in any case, are usually impossible to undo. In a realistic system the necessary control structure for behaviours may be more complex and irregular than the highly disciplined structure of a recovery block.

6 CONTEXTS, PROPERTIES AND REQUIREMENTS

In addition to continuing, non-terminating behaviours, the functionality of a realistic computer-based system does, of course, include bounded terminating behaviours that involve both the computer and the problem world and correspond to meaningful requirement that can be largely captured in post-conditions. The post-condition is rarely more than a partial specification, because the involvement of the problem world will almost always impose additional constraints and subsidiary behaviours that must be observed as the behaviour progresses through intermediate states towards termination. An example is closing the doors in the lift system. The pre-condition and post-condition are *doors-open* and *doors-closed*; but progress towards to postcondition involves the door motors and sensors, and may involve obstruction of the doors by a user or a large object. So even here, where the goal of the behaviour is readily specified, the software and its requirement cannot be adequately understood in terms of the (pre-condition, post-condition) abstraction.

Generally, then, the functionality of a computer-based system must be understood in terms of behaviours; and the behaviours must be understood in their appropriate contexts, where the associated assumptions about problem world properties make the behaviour feasible. The idea applies not only to the treatment of faults but also, more widely, to any large-scale variation in required system behaviour. The lift system has healthy and faulty contexts, and it also has contexts determined by operational demands. For example, in addition to the normal lift service, the system has an operational

mode for use by fire brigade personnel in the event of a fire, and another for use by an inspector who is testing the lift before issuing a renewed certificate of safety; the maintenance engineers, too, may need to operate the lift under software control provided by the system. These modes differ functionally: for example, in fire brigade mode the system may ignore the call buttons on the floors, responding only to the request buttons in the lift car, and door opening and closing may be performed only when the corresponding request buttons are pressed in the lift car. In maintenance mode it may be permissible to open the lift car doors when the lift is between floors. In test mode it may be desirable to simulate equipment faults in order to test the system response. In any of these operational modes major or minor faults can occur: their treatment is likely to depend on the current operational mode as well as on the nature of the fault. Design of the control structure software will reflect decisions about the compatibility of the behaviours to be controlled, and the way in which conflicts can be resolved between behaviours that must coexist.

Additional design concerns arise whenever interaction between the computer and a domain of the problem world is switched from one software-controlled behaviour to another. In many cases, a problem world domain imposes a general requirement on the behaviours in which it can engage, and infringing the requirement causes serious harm. For example, in a system to control traffic lights a general requirement is that to avoid collisions North-South and East-West traffic may never be allowed to proceed simultaneously. In the lift system it may be a requirement that the motor direction is never reversed while the motor is running, and that a specified minimum period must intervene between directions in which the motor relay is *off* to allow the motor to come to rest. If the traffic light system can impose different phasing regimes—for example, to serve different traffic patterns at different times of day—it is simple to define a behaviour for each pattern that satisfies the general requirement to avoid collisions. Similarly, it is easy to define the behaviours for the different lift operating modes so that each one satisfies the requirement for an *off* period. The concern arises because the concatenation of two behaviours may not satisfy the requirement that each satisfies individually.

For these examples, this behaviour-switching concern can be addressed in more than one way. Each behaviour can be modified to restrict its possible termination points, so that termination is always preceded by the required ‘neutral’ period. Each behaviour can be modified so that the required ‘neutral period’ always occurs immediately on activation. A brief, terminating, ‘smoothing behaviour’ can be designed and interposed between the earlier behaviour—which may have been preemptively terminated—and the later behaviour. (This ‘smoothing behaviour’ plays a role in behaviour structures that can be regarded as analogous to the restoration of state in recovery blocks.) In other examples, the behaviour-switching concern will arise in different forms and can be addressed by different techniques.

7 UNDERSTANDING, VERIFICATION AND IMPLEMENTATION

For a computer-based system, there may be many behavioural requirements: on the behaviours expected of the human operators or users; on the behaviours in which a problem domain such as the lift motor or the road traffic vehicles can engage without harm; on the cooperative behaviours of human and other parts of the problem world that are necessary to achieve the purposes of the system. The developers of a computer-based system rich in behavioural requirements must pay explicit attention to those behaviours, both those that will be evoked by the software in the problem world, and those that the problem world domains will engage in on their own continuing or sporadic initiative. They must present clear representations of those behaviours to the various interested stakeholders, and obtain their willing comprehension and assent.

Behaviours are inherently sequential temporal structures and assemblages and abstractions of concurrent sets of such temporal structures. Human comprehension and assent to proposed behaviours of a system is a necessary, but not sufficient, prerequisite of successful development and design. Because human faculties have so often proved fallible, it is necessary also to apply formal reasoning techniques to verify that their products are not logically flawed. One successful approach to verification rests on representing behaviours in a fragmented form that can be characterised as ‘operations on a state’ or ‘event-driven programming’. Such a fragmented form is convenient for formal techniques of verifying invariants that should hold over a behaviour, but it is a poor representation for purposes of human understanding.

If a fragmented representation of behaviour is developed for the whole of a realistic system, it may be developed by a process of refinement. In this process the development progresses, step by step, from an initial highly abstract representation of the system to successively more concrete representations. At every step the representation is fragmented, and therefore essentially unstructured. The whole process induces a structure based only on the refinement relation. This is, of course, radically different from a problem-oriented structure over behaviours.

Implementing the software, even for a small system, will inevitably demand yet another structure. Both the requirements-oriented behaviour structure and the verification-oriented refinement structure are concerned in their different ways with the system as a whole, comprising both the problem world and the computer executing the software. The software implementation, by contrast, is a description only of the behaviour of the software itself. Certainly the software may embody an analogical model [Jackson95] of some parts of the problem world, but the model and the world are quite distinct. Each has properties and behaviours that the other does not share, and the causal properties of their interactions with other parts of the system differ radically.

The software, then, demands an architectural structure to add to the behavioural requirements and the verification refinement structures. So development needs at least three different structures—and, no doubt, others that we do not mention here. This profusion of structures presents a serious challenge to current development approaches, for which we are, arguably, ill-prepared. Many of us have wanted to believe, in the words of the present writer [Jackson75], that the structure of the software solution should closely reflect the structure of the problem. This principle may hold good for certain classes of comparatively simple programs. It now seems too simple for computer-based systems of realistic size and complexity. Instead we should be working harder to understand relationships among the several disparate structures necessary to effective development of dependable computer-based systems.

8 ENVOI

We have strayed far from recovery blocks, which furnished the original motivation of this short paper. In doing so we have perhaps done less than justice to the most fundamental concept preceding and underlying the recovery block work. This is the explicit recognition [Randell71] that program ‘correctness’, in the sense of faultlessly satisfying an agreed specification, is not the only way to achieve system reliability, and often may not even be a practical way. One obstacle is the difficulty of obtaining, early enough, a complete software specification that accurately captures what is needed. A specification is more likely to be “little more than an initial bargaining offer, subject to renegotiation as the system implementation proceeds and the designers and customers start getting detailed feedback” [Randell71]. This obstacle presents itself still today, in a form greatly strengthened by the hugely increased, and still growing, complexity of computer-based systems. The recovery block work aimed to circumvent this obstacle. Its authors explicitly accepted the possibility of software failure, and placed its detection and treatment within a coherent and effective structure that could significantly improve system reliability. And they recognised the crucial importance of structure in the larger computer-based systems that were to come.

REFERENCES

[Fickas+95] Stephen Fickas and Martin S Feather; *Requirements Monitoring in Dynamic Environments*; in Proceedings of the Second IEEE International Symposium on Requirements Engineering, pages 140-147, York, March 27-29 1995.

[Jackson75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[Jackson95] Michael Jackson; *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices*; Addison-Wesley, 1995.

[Haugk+85] G Haugk, F M Lax, R D Royer and J R Williams; *The 5ESS(TM) switching system: Maintenance capabilities*; AT&T Technical Journal, 64(6 part 2), pages 1385-1416, July-August 1985.

[Horning+74] James J Horning, Hugh C Lauer, P M Melliar-Smith, Brian Randell; *A Program Structure For Error Detection And Recovery*. Proceedings of International Symposium on Operating Systems, Rocquencourt, April 23-25 1974, pages 171-187, LNCS 16, Springer 1974.

[Randell71] Brian Randell; *Highly Reliable Computing Systems*; Computing Laboratory, University of Newcastle, Technical Report 20, July 1971.

[Randell75] Brian Randell; *System Structure for Software Fault Tolerance*; IEEE Transactions on Software Engineering Volume SE-1 Number 2, pages 220-232, June 1975.

C B Jones and J L Lloyd eds, Dependable and Historic Computing: Essays Dedicated to Brian Randell on the Occasion of his 75th Birthday, LNCS 6875, Springer Verlag, pages 273–282, 2011.