1

The Evolution of the Recovery Block Concept

BRIAN RANDELL and JIE XU
University of Newcastle upon Tyne

ABSTRACT

This chapter reviews the development of the recovery block approach to software fault tolerance and subsequent work based on this approach. It starts with an account of the development and implementations of the basic recovery block scheme in the early 1970s at Newcastle, and then goes on to describe work at Newcastle and elsewhere on extensions to the basic scheme, recovery in concurrent systems, and linguistic support for recovery blocks based on the use of object-oriented programming concepts.

1.1 INTRODUCTION

A research project to investigate system reliability was initiated by the first author at the University of Newcastle upon Tyne in 1971. This was at a time when the problems of software reliability had come to the fore, for example through the discussions at the 1968 and 1969 NATO Software Engineering Conferences, concerning what at the time was termed the “software crisis”. Such discussions were one of the spurs to research efforts, in a number of places, aimed at finding means of producing error-free programs. However, at Newcastle the opposite (or more accurately the complementary) problem, namely that of what to do in situations where, perhaps despite the use of the best available means of achieving error-free code, the possibility of residual design faults could not be denied, was taken as an interesting and worthwhile goal.

A preliminary phase of the project involved a study of a representative set of large software systems, including a major banking system, and an airline reservations system. This provided interesting statistical data confirming that residual software faults were one of the most important causes of system failures and down-time. It was found that in all these systems, a sizeable proportion of their code and complexity was related to provisions for (mostly hardware) fault tolerance, such as data consistency checking, and checkpointing schemes. However, these provisions, though typically rather ad hoc, were often quite effective, and indeed managed to cope with some of the software errors that were encountered in practice during system operation, even though the fault tolerance provisions had not been specifically designed to do this.

We were well aware that if we were to develop techniques aimed explicitly at tolerating software faults we would have to allow for the fact that the principal cause of residual software design faults is complexity. Therefore the use of appropriate structuring techniques would be crucial — otherwise the additional software that would be needed might well increase the system’s complexity to the point of being counter-productive. Aided by what we had found in our examination of the checkpoint and restart facilities then being employed, we came to realize that although a variety of even quite disparate error detection mechanisms could usefully be employed together in a system, it was critical to have a simple, coherent and general strategy for error recovery. Moreover it was evident that such a strategy ought to be capable of coping with multiple errors, including ones that were detected during the error recovery process itself.

The first structuring technique that we developed was in fact the basic “recovery block” scheme. In what follows we use the structuring concepts that we later developed, in our description of this basic scheme, and of some of the ensuing research on recovery blocks carried out at Newcastle and elsewhere, before discussing some of the latest ideas that we have been investigating on the structuring of fault-tolerant software.

1.2 SYSTEM STRUCTURING

Our interest in the problems of structuring systems so as to control their complexity, and in particular that of their fault tolerance provisions, led us to a style of system design which is based on what we term *idealized fault-tolerant components* [Anderson and Lee 1981; Randell 1984]. Such components provide a means of system structuring which makes it easy to identify *what* parts of a system have *what* responsibilities for trying to cope with *which* sorts of fault.

We view a system as a set of components interacting under the control of a design (which is itself a component of the system) [Lee and Anderson 1990]. Clearly, the system model is recursive in that each component can itself be considered as a system in its own right and thus may have an internal design which can identify further sub-components. Components receive requests for service and produce responses. When a component cannot satisfy a request for service, it will return an exception. An idealized fault-tolerant component should in general provide both normal and abnormal (i.e. exception) responses in the interface between interacting components, in a framework which minimizes the impact of these provisions on system complexity. Three classes of exceptional situation (i.e. in which some fault tolerance response is needed) are identified. Interface exceptions are signalled when interface checks find that an invalid service request has been made to a component. These exceptions must be treated by the part of the system which made the invalid request. Local exceptions are raised when a component detects an error that its own fault tolerance capabilities could or should deal with in the hope that the component would return to normal operations after exception handling. Lastly, a failure exception is signalled to notify the component which made the service request that, despite the use of its own fault tolerance capabilities, it has been unable to provide the service requested of it (see Figure 1.1).

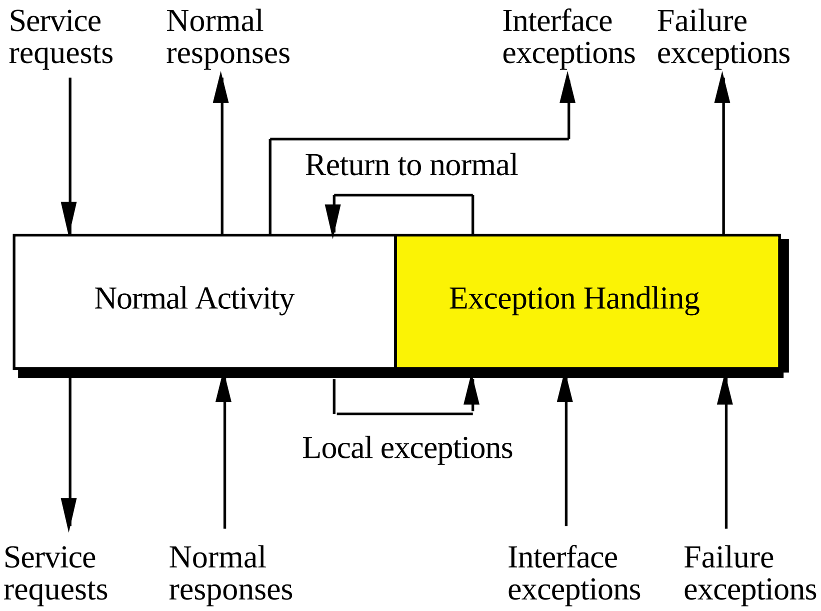

Figure 1.1 Idealized component.

Our notion of an idealized component is mainly concerned with interactions of a component with its environment. It makes minimal assumptions on faults, fault masking and the fault tolerance scheme adopted, in indicating how exception-handling should be structured. Exception handling is often considered as being a limited form of software fault tolerance — for example, by detecting and recovering an error, and either ignoring the operation which generated it or by providing a pre-defined and heavily degraded response to that operation. However, such software cannot be regarded as truly fault-tolerant since some perceived departure from specification is likely to occur, although the exception handling approach can result in software which is robust in the sense that catastrophic failure can often be avoided.

In order also to achieve effective design fault tolerance, capable of completely masking the effects of many residual software errors, it is necessary to incorporate deliberate redundancy, i.e. to make use of design diversity, in such systems. The structuring scheme that we have developed [Randell and Xu 1993] both for describing and comparing the various existing software fault tolerance schemes, and indeed for guiding their actual implementation, is illustrated in Figure 1.2.

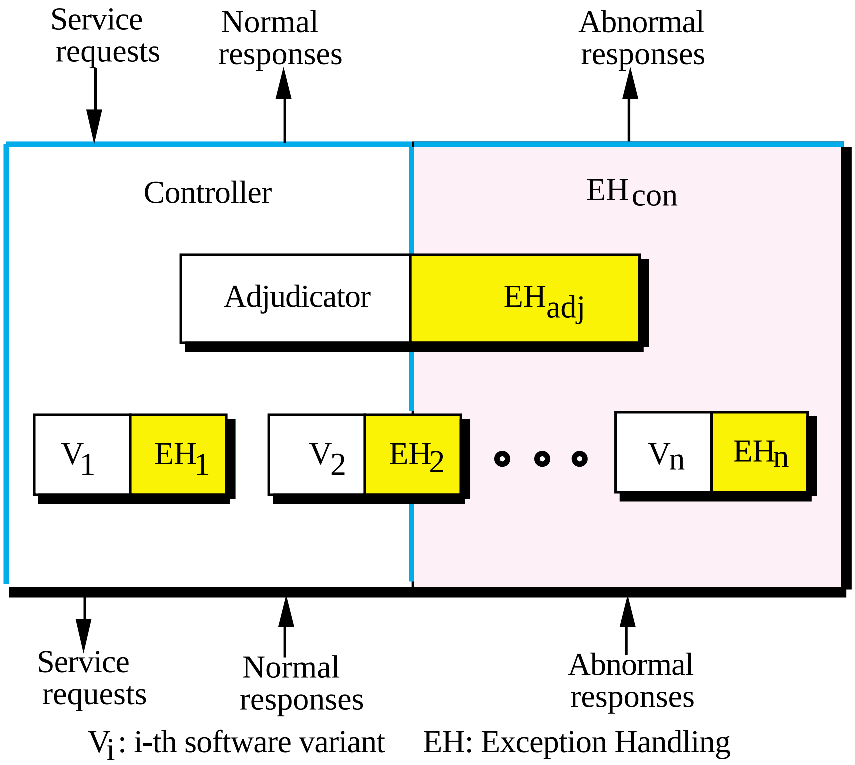

Figure 1.2 Idealized component with design redundancy.

This shows an idealized component which consists of several sub-components, namely an adjudicator and a set of software variants (modules of differing design aimed at a common specification). The design of the component, i.e., the algorithm which is responsible for defining the interactions between the sub-components, and establishing connections between the component and the system environment, is embodied in the controller. This invokes one or more of the variants, waits as necessary for such variants to complete their execution and invokes the adjudicator to check on the results produced by the variants. As illustrated in Figure 1.2, each of these sub-components (even the adjudicator), as well as the component (controller) itself, can in principle contain its own provisions for exception handling, and indeed for full software fault tolerance, so the structuring scheme is fully recursive.

Obviously, the notion of structuring systems needs to be used in such a way as to achieve an appropriate structuring of the complex asynchronous activities to which the system can give rise, in particular those related to fault tolerance. In common with other groups, we make use of so-called *atomic actions* for this purpose. The activity of a group of components constitutes an atomic action if no information flows between that group and the rest of the system for the duration of the activity [Lee and Anderson 1990]. Atomic actions may be planned when the system is designed, or (less commonly) may be dynamically identified by exploratory techniques after the detection of an error. Planned atomic actions must be maintained by imposing constraints on communication within the system. Error recovery can be linked to the notion of an atomic action, which is said to form a restorable action if all components within the action retain the ability to perform a mutually consistent state restoration. These issues are discussed further in Section 1.6.

1.3 RECOVERY BLOCKS

In this Section, we discuss recovery blocks in detail, making use of the exception handling terminology introduced above. The basic recovery block relates to sequential systems. Details of extensions for use in concurrent systems are discussed in Section 1.6. The recovery block approach attempts to prevent residual software faults from impacting on the system environment, and it is aimed at providing fault-tolerant functional components which may be nested within a sequential program. The usual syntax is as follows:

```
ensure      acceptance test
by          primary alternate
else by     alternate 2
            .
            .
else by     alternate n
else error
```

Here the alternates correspond to the variants of Figure 1.2, and the acceptance test to the adjudicator, with the text above being in effect an expression of the controller. On entry to a recovery block, the state of the system must be saved to permit backward error recovery, i.e., establish a checkpoint. The primary alternate is executed and then the acceptance test is evaluated to provide an adjudication on the outcome of this primary alternate. If the acceptance test is passed then the outcome is regarded as successful and the recovery block can be exited, discarding the information on the state of the system taken on entry (i.e., checkpoint). However, if the test fails or if any errors are detected by other means during the execution of the alternate, then an exception is raised and backward error recovery is invoked. This restores the state of the system to what it was on entry. After such recovery, the next alternate is executed and then the acceptance test is applied again. This sequence continues until either an acceptance test is passed or all alternates have failed the acceptance test. If all the alternates either fail the test or result in an exception (due to an internal error being detected), a failure exception will be signalled to the environment of the recovery block. Since recovery blocks can be nested, then the raising of such an exception from an inner recovery block would invoke recovery in the enclosing block. The operation of the recovery block is further illustrated in Figure 1.3.

Obviously, the linguistic structure for recovery blocks requires a suitable mechanism for providing automatic backward error recovery. Randell produced the first such "recovery cache" scheme, a description of which was included in the first paper on recovery blocks [Horning et al. 1974] (although this scheme was later superseded [Anderson and Kerr 1976]). This paper also included a discussion of "recoverable procedures" — a rather complex mechanism that Lauer and Randell had proposed as a means of extending the recovery cacheing scheme to deal with programmer-defined data types. This part of the paper would undoubtedly have been much clearer if the ideas had been expressed in object-oriented terms — a point we will develop further in Section 1.7.

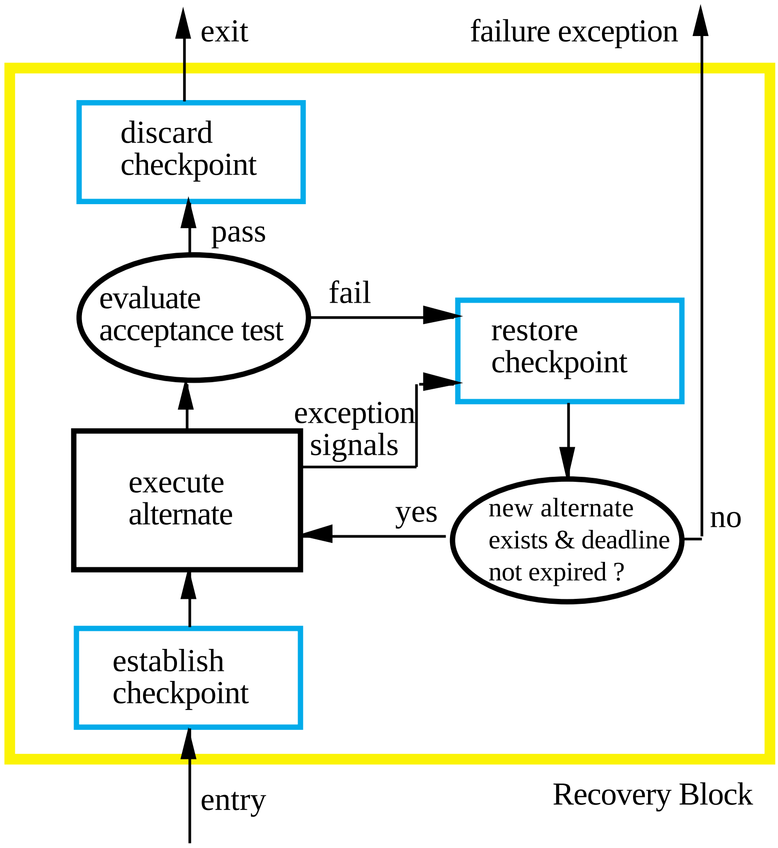

Figure 1.3 Operation of the recovery block.

[Horning et al. 1974] (although this scheme was later superseded [Anderson and Kerr 1976]). This paper also included a discussion of "recoverable procedures" — a rather complex mechanism that Lauer and Randell had proposed as a means of extending the recovery cacheing scheme to deal with programmer-defined data types. This part of the paper would undoubtedly have been much clearer if the ideas had been expressed in object-oriented terms — a point we will develop further in Section 1.7.

The overall success of the recovery block scheme rests to a great extent on the effectiveness of the error detection mechanisms used — especially (but not solely) the acceptance test. The acceptance test must be simple otherwise there will be a significant chance that it will itself contain design faults, and so fail to detect some errors, and/or falsely identify some conditions as being erroneous. Moreover, the test will introduce a run-time overhead which could be unacceptable if it is very complex. The development of simple, effective acceptance tests can thus be a difficult task, depending on the actual specification.

In fact, the acceptance test in a recovery block should be regarded as a last line of detecting errors, rather than the sole means of error detection. The expectation is that it will be buttressed by executable assertion statements within the alternates and run-time checks supported by the hardware. Generally, any such exception raised during the execution of an alternate will lead to the same recovery action as for acceptance test failure. Should the final alternate fail, for example by not passing the acceptance test, this will constitute a failure of the entire module containing the recovery block, and will invoke recovery at the level of the surrounding recovery block, should there be one.

In other words, each alternate should itself be an ideal fault-tolerant component. An exception raised by run-time assertion statements within the alternate or by hardware error-detection mechanisms may be treated by the alternate's own fault tolerance capabilities. A failure exception is raised to notify the system (i.e., the control component in our model) if, despite the use of its own fault tolerance capabilities, the alternate has been unable to provide the service requested of it. The control component may invoke then another alternate.

In general, as described in [Melliar-Smith and Randell 1977] forward error recovery can be further incorporated in recovery blocks to complement the underlying backward error recovery. (In fact, a forward error recovery mechanism can support the implementation of backward error recovery by transforming unexpected errors into default error conditions [Cristian 1982].) If, for example, a real-time program communicated with its (unrecoverable) environment from within a recovery block then, if recovery were invoked, the environment would not be able to recover along with the program and the system would be left in an inconsistent state. In this case, forward recovery would help return the system to a consistent state by sending the environment a message informing it to disregard previous output from the program.

In the first paper about recovery blocks [Horning et al. 1974], Horning et al list four possible failure conditions for an alternate: i) failure of the acceptance test, ii) failure to terminate, detected by a timeout, iii) implicit error detection (for example divide by zero), and iv) failure exception of an inner recovery block. Although the mechanism for implementing the time-out detection measure was not discussed by the authors, the original definition of recovery blocks does cover this issue. Several implementations of watchdog timers for recovery blocks have been described [Hecht 1976; Kim and Welch 1989]. Timeout can be provided as a syntactic form in the recovery block structure [Gregory and Knight 1985]. As with the **else error** part, the **timeout** part allows the programming of a "last ditch" algorithm for the block to achieve its goal, and is really a form of forward recovery since its effects will not be undone (at least at this level).

Although each of the alternates within a recovery block endeavours to satisfy the same acceptance test there is no requirement that they all must produce the same results [Lee 1978]. The only constraint is that the results must be acceptable — as determined by the test. Thus, while the primary alternate should attempt to produce the desired outcome, the further alternate may only attempt to provide a degraded service. This is particularly useful in real-time systems, since there may be insufficient time available for fully-functional alternates to be executed when a fault is encountered. An extreme corresponds to a recovery block which contains a primary module and a null alternate [Anderson and Knight 1983; Anderson et al. 1985]. Under these conditions, the role of the recovery block is simply to detect and recover from errors by ignoring the operation where the fault manifested itself. This is somewhat similar to forward error recovery because the manifestation of a fault will result in a loss of service to the environment. But the important difference is that forward recovery can only remove predictable errors from the system state, whereas such backward recovery can still cope with the unpredictable errors caused by residual design faults. (The only constraint is that the errors do not impact the actual recovery block mechanism.)

Most of the time, only the primary alternate of the recovery block is executed. (This keeps the run-time overhead of the recovery block to a minimum and makes good use of the system and hardware resources.) However, this could cause a problem: the alternates must not retain data locally between calls, otherwise these modules could become inconsistent with each other since not all of them are executed each time when the recovery block is invoked. The problem becomes more obvious while one attempts to design an alternate as an object. There is no guarantee that the state of the object is correctly modified unless the object is invoked each time. Distributed (parallel) execution of recovery blocks [Kim 1984] could solve this issue. An alternative solution is to design the alternate modules as memoryless functional components rather than as objects.

Unlike tolerance to hardware malfunctions, software fault tolerance cannot be made totally transparent to the application programmer although some operations related to its provision, such as saving and restoring the state of the system, can be made automatic and transparent. The programmer who wishes to use software fault tolerance schemes must provide software variants and adjudicators. Therefore, a set of special linguistic features or conventions is necessary for incorporating software redundancy in programs. The key point here is to attempt to keep the syntactic extension simple, natural and minimal. This will be further discussed in the Section 1.7.

1.4 EARLY IMPLEMENTATIONS AND EXPERIMENTS

The first implementation of recovery blocks involved defining and simulating a simple stack-oriented instruction set, incorporating a recovery cache [Anderson and Kerr 1976]. Simple test programs embodying recovery blocks could be run on this machine simulator, and subjected to deliberate faults. Test programs were run on one computer — a separate computer was used to provide data to, and to accept and check the output from, these programs. This second computer also provided facilities by means of which experimenters could make arbitrary changes to any locations in the simulated memory. Visitors to the project were typically challenged to use these facilities to try and cause a demonstration recovery block program to fail — their inability to do so was a persuasive argument for the potential of the recovery block scheme!

Another experimental system is described in [Shrivastava 1978; Shrivastava and Akinpelu 1978] in which recovery blocks were incorporated in the language Pascal. The modification was made to the kernel and interpreter of Brinch Hansen's Pascal system to support the syntax of recovery blocks and the associated recovery caches needed for state restoration. Based on this extension and on a few experimental programs, some performance measurements for recovery blocks were reported, which generally support the belief that recovery blocks do not impose any serious runtime and recovery data space overheads. For the sample programs, the run-time overhead ranged between 1 to about 11% of T1 (execution time of a program without any recovery facilities) when no errors are detected. When a primary fails, the time taken to restore system state was up to about 30% of T1. This experiment also showed that recovery caches made a substantial saving in space, compared with complete checkpointing.

The recovery cache mechanism should ideally form an integral part of a given computer; this not being possible for the existing hardware. The next major work at Newcastle on the implementation of the basic recovery block scheme involved the design and building of a hardware recovery cache for the PDP-11 family of machines [Lee et al. 1980]. This device was inserted into the bus between the CPU and memory modules without requiring hardware alterations. It intercepted writes to memory, and automatically determined whether the contents of the memory location that was about to be over-written needed to be saved beforehand. In order to minimize the overheads imposed on the host, special hardware was designed to enable concurrent operation of the recovery cache and the host system.

The controversial nature of software fault tolerance spurred extensive efforts aimed at providing evidence of the scheme's potential cost-effectiveness in real systems. (The developers of N-version programming [Avizienis and Chen 1977] were similarly motivated to undertake extensive experimental evaluations, as discussed in Chapter 2.) During 1981-84 therefore, a major project directed by Tom Anderson applied an extension of recovery blocks in the implementation of a Naval Command and Control system composed of about 8000 lines of CORAL programming, and made use of the above-mentioned hardware cache [Anderson et al. 1985]. The practical development work of the project included the design and implementation of a virtual machine which supported recovery blocks, together with extensions to the CORAL programming language to allow software fault-tolerance applications to be written in this high-level language. To maintain realism the system was constructed by experienced programmers in strict accordance with the official rules for defence-related software projects. Analysis of experimental runs of this system showed that a failure coverage of over 70% was achieved. The supplementary cost of developing the fault-tolerant software was put at 60% of the implementation cost. The system overheads were measured at 33% extra code memory, 35% extra data memory and 40% additional run time. These led to the conclusion that "by means of software fault tolerance a significant and worthwhile improvement in reliability can be achieved at acceptable cost" [Anderson et al. 1985].

Research at the Royal Military College of Science subsequently extended this experiment to the design of a demonstrator modelled on functions provided at the London Air Traffic Control Centre, and the results have reinforced confidence in the cost-effectiveness and the general applicability of the recovery block approach [Moulding and Barrett 1987].

1.5 EXTENSIONS AND APPLICATIONS OF BASIC RECOVERY BLOCKS

Many applications and varieties of recovery blocks have been explored and developed by various researchers. Some of typical experiments and extensions are considered below.

1.5.1 Distributed Execution of Recovery Blocks

H. Hecht was the first to propose the application of recovery blocks to flight control systems [Hecht 1976; Hecht and Hecht 1986]. His work included an implementation of a watchdog timer that monitors availability of output within a specified time interval and his model also incorporates a rudimentary system to be used when all alternates of the recovery block scheme are exhausted. Since then, further researches and experiments have been conducted by Hecht and his colleagues. For example, M. Hecht et al [Hecht et al. 1989] described a distributed fault-tolerant architecture, called the Extended Distributed Recovery Block, for nuclear reactor control and safety functions. Their architecture relies on commercially available components and thus allows for continuous and inexpensive system enhancement. The fault injection experiments during the development process demonstrate that the system could tolerate most single faults and dual faults.

K. H. Kim and his colleagues in the DREAM Laboratory have extensively explored the concept of distributed execution of recovery blocks, a combination of both distributed processing and recovery blocks, as an approach for uniform treatment of hardware and software faults [Kim 1984; Kim and Welch 1989; Kim and Yoon 1988; Welch 1983]. The details are given in Chapter 8. A useful feature of their approach is the relatively low run-time overhead it requires so that it is suitable for incorporation into real-time systems. The basic structure of the distributed recovery block is straightforward: the entire recovery block, two alternates with an acceptance test, is fully replicated on the primary and backup hardware nodes. However, the roles of the two alternate modules are not the same in the two nodes. The primary node uses the first alternate as the primary initially, whereas the backup node uses the second alternate as the initial primary. Outside of the distributed recovery block, forward recovery can be achieved in effect; but the node affected by a fault must invoke backward recovery by executing an alternate for data consistency with the other nodes. To test the execution efficiency of the approach, two experimental implementations and measurements have been conducted on distributed computer networks. The results indicate the feasibility of attaining fault tolerance in a broad range of real-time applications by means of the distributed recovery blocks.

1.5.2 Consensus Recovery Blocks

The consensus recovery block (CRB) [Scott et al. 1985] is an attempt to combine the techniques used in the recovery block and N-version programming [Avizienis and Chen 1977]. It is claimed that the CRB technique reduces the importance of the acceptance test used in the recovery block and is able to handle the case where NVP would not be appropriate since there are multiple correct outputs. The CRB requires design and implementation of *N* variants of the algorithm which are ranked (as in the recovery block) in the order of service and reliance. On invocation, all variants are executed and their results submitted to an adjudicator, i.e. a voter (as used in N-version programming). The CRB compares pairs of results for compatibility. If two results are the same then the result is used as the output. If no pair can be found then the results of the variant with the highest ranking are submitted to an acceptance test. If this fails then the next variant is selected. This continues until all variants are exhausted or one passes the acceptance test.

[Scott et al. 1987] developed reliability models for the recovery block, N-version programming and the CRB. In comparison, the CRB is shown to be superior to the other two. However, the CRB is largely based on the assumption that there are no common faults between the variants. (This of course is not totally true according to the experiments in [Knight et al. 1985; Scott et al. 1984].) In particular, if a matching pair is found, there is no indication that the result is submitted to the acceptance test, so a correlated failure in two variants could result in an erroneous output and would cause a catastrophic failure.

1.5.3 Retry Blocks with Data Diversity

A retry block developed by Ammann and Knight [Ammann and Knight 1987; Ammann and Knight 1988] is a modification of the recovery block scheme that uses data diversity instead of design diversity. Data diversity is a strategy that does not change the algorithm of the system (just retry), but does change the data that the algorithm processes. It is assumed that there are certain data which will cause the algorithm to fail, and that if the data were re-expressed in a different, equivalent (or near equivalent) form the algorithm would function correctly. A retry block executes the single algorithm normally and evaluates the acceptance test. If the test passes, the retry block is complete. If the test fails, the algorithm executes again after the data has been re-expressed. The system repeats this process until it violates a deadline or produces a satisfactory output. The crucial elements in the retry scheme are the acceptance test and the data re-expression routine.

A description of some experiments with the retry block is presented by the authors. Coordinates to a radar system were altered to lie on the circumference of a small circle centered on the point, taking advantage of the fact that this application's data had limited precision. The radius of the circle and the re-expression algorithm were both changed to generate an indication of their influence. Although the overall performance of the retry block varied greatly, a large reduction in failure probability for some of the faults is observed in the study. Compared with design diversity, data diversity is relatively easy and inexpensive to implement. Although additional costs are incurred in the algorithm for data re-expression, data diversity requires only a single implementation of a specification. Of course, the retry scheme is not generally applicable and the re-expression algorithm must be tailored to the individual problem at hand and should itself be simple enough to eliminate the chance of design faults.

1.5.4 Self-Configuring Optimal Programming

SCOP (Self-configuring optimal programming) [Bondavalli et al. 1993; Xu et al. 1993] is another attempt to combine some techniques used in RB and NVP in order to enhance efficiency of software fault tolerance and to eliminate some inflexibilities and rigidities. This scheme organizes the execution of software variants in phases, dynamically configuring a currently active variant set, so as to produce acceptable results with the relatively small effort and to make the efficient use of available resources. The control can be parameterized with respect to the level of fault tolerance, the amount of available resources and the desired response time. Since highly dynamic behaviour can cause complexity of control and monitoring, a methodology for devising various instances of SCOP is developed by simplifying the on-line process at the price of the complex off-line design.

The gain of efficiency would be limited when the supporting system is intended for a specific application — the hardware resources saved by the SCOP scheme would be merely left idle. It is perhaps more appropriate if the application environments are complex and highly variable, such as a large distributed computing system that supports multiple competing applications.

1.5.5 Other Applications

Sullivan and Masson developed an algorithm-oriented scheme, based on the use of what they term Certification Trails [Sullivan and Masson 1990; Sullivan and Masson 1991]. The central idea of their method is to execute an algorithm so that it leaves behind a trail of data (certification trail) and, by using this data, to execute another algorithm for solving the same problem more quickly. The outputs of the two executions are compared and considered correct only if they agree. An issue with the data trail is that the first algorithm may propagate an error to the second algorithm, and this could result in an erroneous output. Nevertheless, the scheme is an interesting alternative to the recovery block scheme, despite being perhaps of somewhat limited applicability.

Delta-4 was a collaborative project carried out within the framework of the European Strategic Programme for Research in Information Technology (ESPRIT) [Powell 1991]. Its aim was the definition and design of an open, dependable, distributed computer system architecture. The Delta-4 approach deals mainly with hardware fault tolerance, but also addresses the issue of design faults. [Barrett and Speirs 1993] describes the integration of software fault tolerance mechanisms into the existing Delta-4 architecture. The authors claimed that the incorporation of recovery blocks and dialogues (structures for supporting inter-process recovery) into the Delta-4 framework is obtained without significant overheads.

1.6 RECOVERY IN CONCURRENT SYSTEMS

Work at Newcastle on this topic dates from 1975, when we began to consider the problems of providing structuring for error recovery among sets of cooperating processes. (A few researches were also made into error recovery among the particular sets of so-called competing processes where the processes communicate only for resource sharing [Shrivastava and Banatre 1978; Shrivastava 1979].) Having identified the dangers of what we came to term the *domino effect*, we had come up with the notion of a *conversation* [Randell 1975] — something which we later realized was a special case of a nested atomic action.

1.6.1 Conversations

When a system of cooperating processes employs recovery blocks, each process will be continually establishing and discarding checkpoints, and may also need to restore to a previously established checkpoint. However, if recovery and communication operations are not performed in a coordinated fashion, then the rollback of a process can result in a cascade of rollbacks that could push all the processes back to their beginnings — the domino effect. This causes the loss of entire computation performed prior to the detection of the error. Figure 1.4 illustrates the domino effect with two communicating processes.

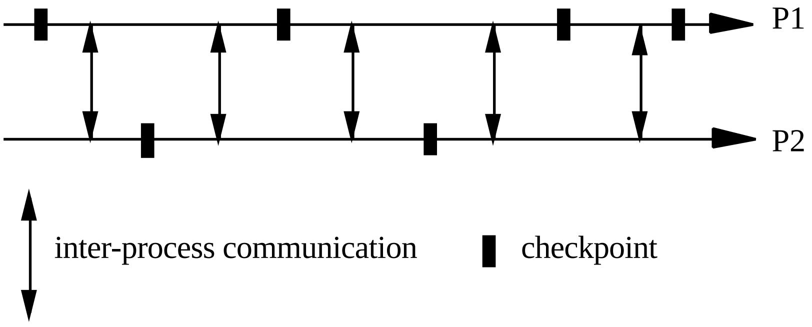

Figure 1.4 The domino effect.

The conversation scheme is in our view one of the fundamental approaches to structured design of fault-tolerant concurrent programs. It provides a means of coordinating the recovery blocks of interacting processes to avoid the domino effect. Figure 1.5 shows an example where three processes communicate within a conversation and the processes P1 and P2 communicate within a nested conversation. Communication can only take place between processes that are participating in a conversation together. The operation of a conversation is: (i) on entry to a conversation a process establishes a checkpoint; (ii) if an error is detected by any process then all the participating processes must restore their checkpoints; (iii) after restoration all processes use their next alternates; and (iv) all processes leave the conversation together. The concept of conversation facilitates failure atomicity and backward recovery in cooperating process systems in a manner analogous to that of the atomic action mechanism in object-based systems. In fact, this terminological distinction between the area of communicating process systems and that of of object-based systems is, we claim, of only surface importance [Shrivastava et al. 1993].

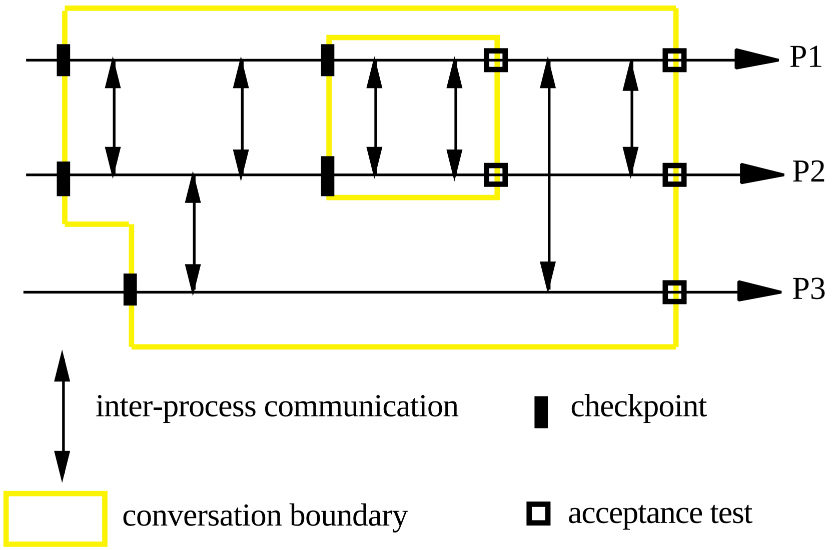

Figure 1.5 Nested conversations.

Considerable research has been undertaken into the subject of concurrent error recovery, including improvements on the conversation and different implementations of it. There are at least two classes of approaches to preventing the domino effect: the coordination-by-programmer approach and the coordination-by-machine approach. With the first approach, the application programmer is fully responsible for designing processes so that they establish checkpoints in a well coordinated manner [Randell 1975; Russell 1980; Kim 1982]. Many authors have added language constructs to facilitate the definition of restorable actions based on this approach [Russell and Tiedeman 1979; Anderson and Knight 1983; Gregory and Knight 1985; Jalote and Campbell 1984; Jalote and Campbell 1986]. In contrast, the coordination-by-machine approach relies on an "intelligent" underlying processor system which automatically establishes appropriate checkpoints of interacting processes [Kim 1978; Barigazzi and Strigini 1983; Koo and Toueg 1987, Kim and You 1990]. If restorable actions are unplanned, so that the recovery mechanism must search for a consistent set of checkpoints, such actions would be expensive and difficult to implement. However, such exploratory techniques have the advantage that no restrictions are placed on inter-process communication and that a general mechanism could be applied to many different systems [Merlin and Randell 1978; Wood 1981]. To reduce synchronization delays introduced by controlled recovery, some researches have focused on the improvement of performance, such as the lookahead scheme and the pseudo-recovery block [Kim et al. 1976; Kim and Yang 1988; Ramanathan and Shin 1988; Russell and Tiedeman 1979; Shin and Lee 1984].

1.6.2 Extensions and Implementations of Conversations

The original description of conversations provided a structuring or design concept without any suggested syntax. [Russell and Tiedeman 1979] proposed a syntax called the name-linked recovery block for the concept of conversations. Kim [Kim 1982] presents three different syntactic forms for conversations based on the monitor structure. The different implementations deal with the distribution of the code for the recovery blocks of individual processes. The tradeoff is either to spread the conversation among the individual processes such that all of the code of each process is in one location or have all the code for the conversation in one location.

There was no provision for linked forward error recovery in the original conversation scheme. Campbell and Randell [Campbell and Randell 1986] proposed techniques for structuring forward error recovery measures in asynchronous systems and generalized ideas of atomic actions so as to support fault-tolerant interactions between processes. A resolution scheme is used to combine multiple exceptions into a single exception if they are raised at the same time.

Issarny extended their work to concurrent object-oriented systems by defining an exception-handling mechanism for parallel object-oriented programming [Issarny 1993a]. This mechanism was then generalized to support both forward and backward error recovery [Issarny 1993b]. Following the proposal in [Campbell and Randell 1986], Jalote and Campbell described a system which contains both forward and backward error recovery within a conversation structure (also known as an FT-action). Their system was based on communicating sequential processes (CSP) [Hoare 1978] with one extension (the `exit`) statement.

Forward error recovery in an FT-action [Jalote and Campbell 1985] is achieved through linked exception handlers where each process has its own handler for each exception. When an exception is raised by a process it is propagated to all the participating processes within the FT-action. Each process then executes its own handler for that exception. Backward recovery within an FT-action is obtained by recovery blocks. Every participating process is required to have the same number of alternates. An FT-action can combine the two schemes so that forward and backward error recovery are used within the same structure. It can also cope with the issue of real-time applications through a simple timer.

Real-time applications may suffer from the possibility of *deserters* in a conversation — if a deadline is to be met then a process that fails to enter the conversation or to reach its acceptance test could cause all the processes in the conversation to miss that deadline [Kim 1982]. Russell and Tiedeman [Russell and Tiedeman 1979] considered relaxing the requirement for all processes exiting together so as to enable some protection against deserter processes, but this could lead to the domino effect. Campbell, Horton and Belford [Campbell et al. 1979] proposed a deadline mechanism for dealing with timing faults. Anderson and Knight [Anderson and Knight 1983] proposed *exchanges* as a simplification of conversations where the cyclic nature of real-time systems is exploited. An exchange is a conversation in which all participating processes enter upon initiation and terminate upon exit. Error recovery is particularly easy as the recovery data is only that needed upon initiation, which should only be a small amount of frame dependent data.

Gregory and Knight [Gregory and Knight 1985] identified a set of problems associated with conversations. They argued that there ought to be two types of acceptance test — one for each process within a conversation to check its own goal and one for the whole structure of the conversation to check the global goal. In addition, within a conversation or other structures mentioned above the set of processes that attempt their primary alternate is the same as the set of processes which attempt all other alternates, i.e. they all roll back and try again with their further alternates. This is overly restrictive and affects independence of algorithm between alternates. In an effort to solve these problems, the authors developed two concepts — a *colloquy* that contains many *dialogs.*

A dialog is a way of enclosing a set of processes in an atomic action. It provides no retry method and no definition of the action to be taken upon failure. If any failure occurs, the dialog restores all checkpoints and fails, signalling the failure to the surrounding colloquy. A colloquy that contains a set of dialogs controls the execution of dialogs and decides on the recovery action to be taken if the dialog fails. The colloquy provides a means of constructing alternates using a potentially different set of processes, thereby permitting true diverse design. The dialog and colloquy allow time constraints to be specified and are accompanied by syntactic proposals that are extensions to the Ada language.

However, when attempting the integration of the syntax for the colloquy into Ada, the authors found several new and potentially serious difficulties which arise because of a conflict between the semantics of modern programming languages and the needs of concurrent backward recovery [Gregory and Knight 1989].

1.6.3 Practical Difficulties and Possible Solutions

The practical problems mentioned in [Gregory and Knight 1989] fall into the general categories of (i) program structure, (ii) shared objects, and (iii) process manipulation. All the problems have the potential to allow the state outside a dialog (or a conversation) to be contaminated by changes inside the dialog, i.e., *information smuggling*. For example, a major inconsistency exists between the preferred structure of concurrent programs (e.g. involving the use of service processes) and the structure for planned recovery. To avoid information smuggling, the planned backward recovery could cause the *capture effect* for service processes — in other words processes outside the (nested) dialogs cannot use the service processes until the completion of all the dialogs. Shared objects are another significant source of information smuggling, but no simple approaches solve the problem. Smuggling can occur with process manipulations (e.g., dynamic process creation) also. An initial solution to the problem merely raises several other issues. Given the complexity and subtlety of these problems, Gregory and Knight concluded that "the only workable solution might be that programming language design begin with backward error recovery as its starting point." Nevertheless, some preliminary and partial solutions can be found in [Clematis and Gianuzzi 1993; Gregory 1987].

The actual programming of a conversation is another major difficulty associated with the concept. Constructing an application into a sequence of conversations is not a trivial task. The application programmer has to select a boundary composed of a set of checkpoints, acceptance tests and the side walls to prevent information smuggling. This boundary should be integrated well into the structure of processes. [Tyrrell and Holding 1986] suggested a way of identifying adequate boundaries of conversations based on the specification of the application using Petri Nets. [Carpenter and Tyrrell 1991] proposed an alternative solution in which the CSP notation [Hoare 1978] is used to describe the application and conversation boundaries are identified through a trace evaluation, but such traces would cause an explosion of states even for simple applications. In practice, however, it is possible for some special applications to decide on the conversation placement without full trace evaluation [Tyrrell and Carpenter 1992].

1.7 LINGUISTIC SUPPORT FOR SOFTWARE FAULT TOLERANCE

General linguistic supports for software fault tolerance are concerned with much of our latest work. If the design of software fault-tolerant systems is to become widely used on a routine basis, one of important problems that has to be solved is how to develop and provide appropriate linguistic notations and the corresponding environments, which should effectively support the development of fault-tolerant programs without greatly complicating the program's implementation, readability, and maintenance.

1.7.1 Design Notations and Environments

[Liu 1992] proposed a design notation for a wide class of fault-tolerant software structures, mainly offering generality and flexibility in a modular fashion. [Bondavalli and Simoncini 1992] showed that their BSM design description language is sufficient for expressing the typical structures of software fault tolerance, such as recovery blocks and N-version programming, without requiring semantic extensions. [Ancona et al. 1990] described a mechanism, called the *Recovery Metaprogram* (RMP), for the incorporation of fault tolerance functions into application programs, giving programmers a single environment that lets them use the appropriate fault tolerance scheme.

The architecture proposed in [Ancina et al. 1990] contains three components: the application program, the RMP and the kernel. The application programmer must define the software variants and the validation test, and indicate which portions of the application program are involved in fault tolerance. The RMP implements the controllers and the supporting mechanisms for four different schemes, inserting a number of breakpoints in the program. When a breakpoint is reached, the application program is suspended and the kernel activates the RMP which takes actions to support the fault tolerance scheme chosen. The RMP is then suspended, and the application program is reactivated until the next breakpoint is reached. The implementation of the RMP approach may incur an additional cost in the form of intensive context switches and kernel calls.

However, different from those languages and environments discussed above, our major work has been greatly influenced by the now very fashionable topic of object-oriented programming. In particular, we have found it convenient to try to exploit various characteristics of C++ [Stroustrup 1991], a language that has been used extensively at Newcastle in connection with work on distributed systems [Shrivastava et al. 1991].

1.7.2 Implementing Software Fault Tolerance in C++

The recent extension of C++ to include generic classes and functions (“templates”), and exception handling (“catch” and “throw”) makes it possible to implement both forward and backward error recovery in C++ in the form of reusable components that separate the functionality of the application from its fault tolerance [Rubira-Calsavara and Stroud 1993]. More generally, such facilities show prospect of providing a convenient means of achieving high levels of reuse. This would apply both to general software components implementing various fault tolerance strategies (including generalizations and combinations of recovery blocks, and N-version programs, and encompassing the use of parallelism) and to application-specific software components [Randell and Xu 1993]. We also provided a set of pre-defined C++ classes to support a general object-oriented framework for software fault tolerance based directly on the abstract model represented by Figure 1.2, and described in Section 1.2 [Xu et al. 1994]. However, there remain certain strategies and types of structuring that cannot be implemented entirely (or elegantly) in a language like C++ even given such mechanisms as generic functions and inheritance. Instead, the programmer who wishes to employ these strategies has to obey certain conventions. For example, the application programmer who wishes to make use of our C++ classes would have to include explicit calls in each operation of an object to facilities related to the provision of state restoration.

Adherence to such conventions can be automated, by embodying them into a somewhat enhanced version of C++ and using a pre-processor to generate conventional C++ programs automatically. Although the pre-processor approach can be quite practical it does have disadvantages. In particular the language provided to application programmers becomes non-standard since programmers have in some circumstances during program development to work in terms of the program generated by the pre-processor, rather than of the one that they had written. The alternative, that of leaving it to the programmer to adhere to the conventions, is of course a fruitful source of residual program faults. But developing a new language that provides adequate syntax and runtime support to enable the implementation of various software fault tolerance could cut the work off from the mainstream of programming language developments and thus have difficulty in achieving wide acceptance.

1.7.3 Reflection and Reflective Languages

As mentioned in Section 1.3, it has to be the responsibility of the application programmers for developing software variants, acceptance tests, and even application-specific voters. Special language features and/or programming conventions therefore cannot be avoided completely. In consideration of software reliability, the key problem would become how a set of simple (thus easy to check) programming features can be developed with powerful expressibility to enable the implementation of software fault tolerance and how the supporting mechanisms can be provided in a more natural and modular manner rather than by an ad-hoc method such as system calls. Recent developments in the object-oriented language world, under the term “ *reflection*” [Maes 1987], show considerable promise in this regard.

A reflective system can reason about, and manipulate, a representation of its own behaviour. This representation is called the system's meta-level [Agha et al. 1992]. Reflection improves the effectiveness of the object-level (or base-level) computation by dynamically modifying the internal organization (actually the meta-level representation) of the system so as to provide powerful expressibility. Therefore, in a reflective programming language a set of simple, well-defined language features could be used to define much more complex, dynamically changeable constructs and functionalities. In our case, it could enable the dynamic change and extension of the semantics of those programming features that support software fault tolerance concepts, whereas the application-level (or object-level) program is kept simple and elegant [Xu et al. 1994]. Although C++ itself does not provide a metalevel interface, Chiba and Masuda [Chiba and Masuda 1993] describes an extension of C++ to provide a limited form of computational reflection, called Open C++, whose usefulness in expressing software fault tolerance we are now investigating.

However, quite what reflective capabilities are needed for what forms of fault tolerance, and to what extent these capabilities can be provided in more-or-less conventional programming languages, and allied to the other structuring techniques outlined in this Chapter, remain to be determined. In particular, the problems of the combined provision of significant software fault tolerance and hardware fault tolerance, and of evaluating cost-effectiveness, are likely to require much further effort.

When considering support for software fault tolerance in concurrent object-oriented programming, we face a greater challenge because, on the one hand, mainstream object-oriented languages such as C++ and Eiffel [Meyer 1992] do not at present address concurrency and, on the other hand, a large number of different models for concurrent object-oriented programming have been proposed but none has yet received widespread acceptance. There exist only a few tentative proposals for treating concurrent error recovery such as the Arche language [Benveniste and Issarny 1992]. However, the reflection technique seems to be a more promising approach to the structuring of concurrent object-oriented programs [Yonezawa and Watanabe 1989].

1.8 CONCLUSIONS

Looking back on the developments that have occurred since the recovery block concept was first introduced, it is we hope fair to claim that it has proved a very useful abstraction, and starting point for much subsequent research, elsewhere as well as at Newcastle. That at Newcastle can be characterized as mainly involving over the years:

(i) a gradual extension of the original very basic scheme to deal with ever more complex situations, while retaining as much as possible of the essential simplicity of structuring provided by the basic scheme, and more recently (and perhaps rather belatedly)

(ii) the investigation of appropriate linguistic support for recovery blocks and their generalizations using object-oriented structuring concepts.

Whilst we now regard recovery blocks, and N-version programming for that matter, simply as special cases of a more general scheme, there has been a somewhat surprising continued interest by others — especially those involved with statistical experiments and with mathematical modelling (for example [Arlat et al. 1990; Pucci 1990; Tai et al. 1993; Tomek 1993]) — in the basic schemes. This is very flattering, but “real-world” usage of recovery block concepts (see for example [Giloth and Prantzen 1983; Haugk et al. 1985; Gray and Siewiorek 1991; Simon et al. 1990; Gopal and Griffeth 1991]) has always had to deal with such complexities as input-output, parallelism, hardware faults, etc. — so we would urge more concentration on the richer forms of structuring for error recovery and for design diversity which have since been developed, and which we have attempted to describe in the later sections of this chapter.

Acknowledgements

Our research at Newcastle was originally sponsored by the UK Science and Engineering Research Council and by the Ministry of Defence, but in recent years it has been supported mainly by two successive ESPRIT Basic Research projects on Predictably Dependable Computing Systems (PDCS and PDCS2). Needless to say, as will be obvious from the references we have given, the work we have attempted to summarize here has been contributed to by a large number of colleagues. It would be invidious to name just some of these, but we are very pleased to acknowledge our indebtedness to all of them.

References

[Agha et al. 1992] G. Agha, S. Frolund, R. Panwar and D. Sturman, “A linguistic framework for dynamic composition of dependability protocols,” in *DCCA-3,* pp. 197-207, Mondello, 1992.

[Ammann and Knight 1987] P.E. Ammann and J.C. Knight, “Data diversity: an approach to software fault tolerance,” in *17th Int. Symp. Fault-Tolerant Comput.,* pp. 122-126, Pittsburgh, 1987.

[Ammann and Knight 1988] P.E. Ammann and J.C. Knight, “Data diversity: an approach to software fault tolerance,” *IEEE Trans. Comput.,* vol. 37, no. 4, pp.418-425, 1988.

[Ancona et al. 1990] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis and E.B. Fernandez, "A system architecture for fault tolerance in concurrent software," *IEEE Computer*, vol. 23, no. 10, pp.23-32, 1990.

[Anderson et al. 1985] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Sotware fault tolerance: an evaluation," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1502-1510, 1985.

[Anderson and Lee 1981] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice,* Prentice-Hall, 1981.

[Anderson and Kerr 1976] T. Anderson and R. Kerr, "Recovery blocks in action: a system supporting high reliability," in *2nd Int. Conf. on Software Engineering,* pp. 447-457, San Francisco, 1976.

[Anderson and Knight 1983] T. Anderson and J.C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Trans. Soft. Eng.*, vol. SE-9, no. 3, pp.355-364, 1983.

[Arlat et al. 1990] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability modelling and evaluation of software fault tolerant systems," *IEEE Trans. Comput.*, vol. 39, no. 4, pp.504-513, 1990.

[Avizienis and Chen 1977] A. Avizienis and L. Chen, "On the implementation of N-version-programming for software fault-tolerance during execution," in *Int. Conf. Comput. Soft. and Appli.,* pp. 149-155, New York, 1977.

[Barigazzi and Strigini 1983] G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points," in *13th Int. Symp. Fault -Tolerant Comput.,* pp. 48-55, Milano, 1983.

[Barrett and Speirs 1993] P.A. Barrett and N.A. Speirs, "Towards an integrated approach to fault tolerance in Delta-4," *Distrib. Syst. Eng.,* no. 1, pp.59-66, 1993.

[Benveniste and Issarny 1992] M. Benveniste and V. Issarny. *Concurrent programming notations in the object-oriented language Arche,* Research Report, no. 1822, Rennes, France, INRIA, 1992.

[Bondavalli et al. 1993] A. Bondavalli, F. DiGiandomenico and J. Xu, "Cost-effective and flexible scheme for software fault tolerance," *Comput. Syst. Sci. & Eng.,* no. 4, pp.234-244, 1993.

[Bondavalli and Simoncini 1992] A. Bondavalli and L. Simoncini, "Structured software fault tolerance with BSM," in *3rd Workshop Future Trends of Distrib. Comput. Syst.,* Taipei, 1992.

[Campbell et al. 1979] R.H. Campbell, K.H. Horton and G.G. Belford, "Simulations of a fault-tolerant deadline mechanism," in *9th Int. Symp. Fault-Tolerant Comput.,* pp. 95-101, Madison, 1979.

[Campbell and Randell 1986] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 8, pp.811-826, 1986.

[Carpenter and Tyrrell 1991] G.F. Carpenter and A.M. Tyrrell, "Software fault tolerance in concurrent systems: conversation placement using CSP," *Microprocessing and Microprogramming*, vol. 32, pp.373-380, 1991.

[Chiba and Masuda 1993] S. Chiba and T. Masuda, “Designing an extensible distributed language with a meta-level architecture,” in *ECOOP'93*, pp. 482-501, 1993.

[Clematis and Gianuzzi 1993] A. Clematis and V. Gianuzzi, “Structuring conversation in operation/procedure-oriented programming languages,” *Comput. Lang.*, vol. 18, no. 3, pp.153-168, 1993.

[Cristian 1982] F. Cristian, “Exception handling and software fault tolerance,” *IEEE Trans. Comput.*, vol. C-31, no. 6, pp.531-540, 1982.

[Giloth and Prantzen 1983] F.K. Giloth and K.D. Prantzen, “Can the reliability of digital telecommunication switching systems be predicted and measured?,” in *13th Int. Symp. Fault - Tolerant Comput.*, pp. 392-397, Milano, 1983.

[Gray and Siewiorek 1991] J. Gray and D.P. Siewiorek, “High-availability computer systems,” *IEEE Computer*, vol. 24, no. 9, pp.39-48, 1991.

[Gopal and Griffeth 1991] G. Gopal and N.D. Griffeth, “Software fault tolerance in telecommunications systems,” *ACM Operating Systems Review*, vol. 25, no. 2, pp.112-116, 1991.

[Gregory 1987] S.T. Gregory. *Programming language facilities for backward error recovery in real-time systems*. PhD Dissertation, Dept. of Comput. Sci., Univ. of Virginia, 1987.

[Gregory and Knight 1985] S.T. Gregory and J.C. Knight, “A new linguistic approach to backward error recovery,” in *15th Int. Symp. Fault-Tolerant Comput.*, pp. 404-409, Michigan, 1985.

[Gregory and Knight 1989] S.T. Gregory and J.C. Knight, “On the provision of backward error recovery in production programming languages,” in *19th Int. Symp. Fault-Tolerant Comput.*, pp. 506-511, Chicago, 1989.

[Haugk et al. 1985] G. Haugk, F.M. Lax, R.D. Rover and J.R. Williams, “The 5 ESS switching system: maintenance capabilities,” *AT&T Technical Journal*, vol. 64, no. 6, pp.1385-1416, 1985.

[Hecht 1976] H. Hecht, “Fault-tolerant software for real-time applications,” *ACM Computing Surveys*, vol. 8, no. 4, pp.391-407, 1976.

[Hecht and Hecht 1986] H. Hecht and M. Hecht, “Software reliability in the system context,” *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.51-58, 1986.

[Hecht et al. 1989] M. Hecht, J. Agron and S. Hochhauser, “A distributed fault tolerant architecture for nuclear reactor control and safety functions,” in *Real-Time Syst. Symp.*, pp. 214-221, Santa Monica, 1989.

[Hoare 1978] C.A.R. Hoare, “Communicating sequential processes,” *CACM*, vol. 21, no. 8, pp.666-677, 1978.

[Horning et al. 1974] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell, “A program structure for error detection and recovery,” *Lecture Notes in Computer Science*, vol. 16, pp.177-193, 1974.

[Issarny 1993a] V. Issarny, “An exception handling mechanism for parallel object-oriented programming: towards reusable, robust distributed software,” *Journal of Object-Oriented Programming*, vol. 6, no. 6, pp.29-40, 1993a.

[Issarny 1993b] V. Issarny, “Programming notations for expressing error recovery in a distributed object-oriented language,” in *1st Broadcast Open Workshop*, pp. 1-19, Newcastle, 1993b.

[Jalote and Campbell 1984] P. Jalote and R.H. Campbell, “Fault tolerance using communicating sequential processes,” in *14th Int. Symp. Fault-Tolerant Comput.*, pp.347-352, Florida, 1984.

[Jalote and Campbell 1986] P. Jalote and R.H. Campbell, “Atomic actions for software fault tolerance using CSP,” *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.59-68, 1986.

[Kim 1978] K.H. Kim, “An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation rules,” in *Int. Conf. Parallel Processing,* pp. 58-68, 1978.

[Kim 1982] K.H. Kim, “Approaches to mechanization of the conversation scheme based on monitors,” *IEEE Trans. Soft. Eng.*, vol. SE-8, no. 3, pp.189-197, 1982.

[Kim 1984] K.H. Kim, “Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults,” in *4th Int. Conf. Distributed Comput. Sys.,* pp. 526-532, 1984.

[Kim et al. 1976] K.H. Kim, D.L. Russell and M.J. Jenson, *Language tools for fault-tolerant programming,* Tech. Memo. PETP-1, Electronic Sciences Lab., USC, 1976.

[Kim and Welch 1989] K.H. Kim and H.O. Welch, “Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications,” *IEEE Trans. Comput.*, vol. 38, no. 5, pp.626-636, 1989.

[Kim and Yang 1988] K.H. Kim and S.M. Yang, “An analysis of the performance impacts of lookahead execution in the conversation scheme,” in *7th Symp. Reli. Distrib. Syst.,* pp. 71-81, Columbus, 1988.

[Kim and Yoon 1988] K.H. Kim and J.C. Yoon, “Approaches to implementation of a repairable distributed recovery block scheme,” in *18th Int. Symp. Fault-Tolerant Comput.,* pp. 50-55, Tokyo, 1988.

[Kim and You 1990] K.H. Kim and J.H. You, “A highly decentralized implementation model for the programmer-transparent coordination (PTC) scheme for cooperative recovery,” in *20th Int. Symp. Fault-Tolerant Comput.,* pp. 282-289, Newcastle, 1990.

[Knight et al. 1985] J.C. Knight, N.G. Leveson and L.D.S. Jean, “A large scale experiment in N-version programming,” in *15th Int. Symp. Fault-Tolerant Comput.,* pp. 135-140, Michigan, 1985.

[Koo and Toueg 1987] R. Koo and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *IEEE Trans. Soft. Eng.*, vol. SE-13, no. 1, pp.23-31, 1987.

[Lee 1978] P.A. Lee, “A reconsideration of the recovery block scheme,” *Computer Journal*, vol. 21, no. 4, pp.306-310, 1978.

[Lee and Anderson 1990] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice,* Second Edition, Springer-Verlag, 1990.

[Lee et al. 1980] P.A. Lee, N. Ghani and K. Heron, “A recovery cache for the PDP-11,” *IEEE Trans. Comput.*, vol. C-29, no. 6, pp.546-549, 1980.

[Liu 1992] C. Liu, “A general framework for software fault tolerance,” in *IEEE Workshop Fault-Tolerant Parallel & Distrib. Syst.,* Amherst, 1992.

[Maes 1987] P. Maes, “Concepts and experiments in computational reflection,” *SIGPLAN Notices*, vol. 22, no. 12, pp.147-155, 1987.

[Melliar-Smith and Randell 1977] P.M. Melliar-Smith and B. Randell, “Software reliability: the role of programmed exception handling,” *SIGPLAN Notices*, vol. 12, no. 3, pp.95-100, 1977.

[Merlin and Randell 1978] P.M. Merlin and B. Randell, “State restoration in distributed systems,” in *8th Int. Symp. Fault-Tolerant Comput.,* pp. 129-134, Toulouse, 1978.

[Meyer 1992] B. Meyer. *Eiffel: The Language,* Prentice Hall, 1992.

[Moulding and Barrett 1987] M.R. Moulding and P. Barrett. *An investigation into the application of software fault tolerance to air traffic control systems: project final report,* 1049/TD.6 Version 2, RMCS, 1987.

[Powell 1991] D. Powell (Ed.). *Delta-4: A Generic Architecture for Dependable Distributed Computing,* Springer (Berlin), 1991.

[Pucci 1990] G. Pucci, “On the modelling and testing of recovery block structures,” in *20th Int. Symp. Fault-Tolerant Comput.,* pp. 353-363, Newcastle, 1990.

[Ramanathan and Shin 1988] P. Ramanathan and K.G. Shin, “Checkpointing and rollback recovery in a distributed system using common time base,” in *7th Symp. Reli. Distrib. Syst.,* pp. 13-21, Columbus, 1988.

[Randell 1975] B. Randell, “System structure for software fault tolerance,” *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.

[Randell 1984] B. Randell, “Fault tolerance and system structuring,” in *4th Jerusalem Conf. on Information Technology,* pp. 182-191, Jerusalem, 1984.

[Randell and Xu 1993] B. Randell and J. Xu, “Object-oriented software fault tolerance: framework, reuse and design diversity,” in *1st PDCS2 Open Workshop,* pp. 165-184, Toulouse, 1993.

[Rubira-Calsavara and Stroud 1993] C.M.F. Rubira-Calsavara and R.J. Stroud, “Forward and backward error recovery in C++,” in *1st PDCS2 Open Workshop,* pp. 147-164, Toulouse, 1993.

[Russell 1980] D.L. Russell, “State restoration in systems of communicating processes,” *IEEE Trans. Soft. Eng.*, vol. SE-6, no. 2, pp.183-194, 1980.

[Russell and Tiedeman 1979] D.L. Russell and M.J. Tiedeman, “Miltiprocess recovery using conversations,” in *9th Int. Symp. Fault-Tolerant Comput.,* pp. 106-109, 1979.

[Scott et al. 1984] R.K. Scott, J.W. Gault and D.F. Mcallister, "Investigating version dependence in fault tolerant software," in *AGARD Conf. Proc. 360,* 1984.

[Scott et al. 1985] R.K. Scott, J.W. Gault and D.F. Mcallister, "The consensus recovery block," in *Total Sys. Reli. Symp.,* pp. 74-85, 1985.

[Scott et al. 1987] R.K. Scott, J.W. Gault and D.F. Mcallister, "Fault tolerant software reliability modeling," *IEEE Trans. Soft. Eng*., vol. SE-13, no. 5, pp.582-592, 1987.

[Shin and Lee 1984] K.G. Shin and Y. Lee, "Evaluation of error recovery blocks used for cooperating processes," *IEEE Trans. Soft. Eng*., vol. SE-10, no. 6, pp.692-700, 1984.

[Shrivastava 1978] S.K. Shrivastava, "Sequential pascal with recovery blocks," *Software - Practice and Experience,* vol. 8, pp.177-185, 1978.

[Shrivastava 1979] S.K. Shrivastava, "Concurrent pascal with backward error recovery: language features and examples," *Software - Practice and Experience,* vol. 9, pp.1001-1020, 1979.

[Shrivastava and Akinpelu 1978] S.K. Shrivastava and A.A. Akinpelu, "Fault-tolerant sequential programming using recovery blocks," in *8th Int. Symp. Fault-Tolerant Comput.,* pp. 207, Toulouse, 1978.

[Shrivastava and Banatre 1978] S.K. Shrivastava and J.-P. Banatre, "Reliable resource allocation between unreliable processes," *IEEE Trans. Soft. Eng.*, vol. SE-4, no. 3, pp.230-241, 1978.

[Shrivastava et al. 1991] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An overview of the Arjuna distributed programming system," *IEEE Software,* vol. 8, no. 1, pp.66-73, 1991.

[Shrivastava et al. 1993] S.K. Shrivastava, L. V. Mancini and B. Randell, "The duality of fault-tolerant system structures," *Software - Practice and Experience,* vol. 23, no. 7, pp.773-798, 1993.

[Simon et al. 1990] D. Simon, C. Hourtolle, H. Biondi, J. Bernelas, P. Duverneuil, S. Gallet, P. Vielcanet, S. DeViguerie, F. Gsell and J.N. Chelotti, "A software fault tolerance experiment for space applications," in *20th Int. Symp. Fault-Tolerant Comput.,* pp. 28-35, Newcastle, 1990.

[Stroustrup 1991] Stroustrup. *The C++ Programming Language,* Addison Wesley, 1991.

[Sullivan and Masson 1990] G.F. Sullivan and G.M. Masson, "Using certification trails to achieve software fault tolerance," in *20th Int. Symp. Fault-Tolerant Comput.,* pp. 423-431, Newcastle, 1990.

[Sullivan and Masson 1991] G.F. Sullivan and G.M. Masson, "Certification trails for data structures," in *21st Int. Symp. Fault-Tolerant Comput.,* pp. 240-247, Montreal, 1991.

[Tai et al. 1993] A. Tai, A. Avizienis and J. Meyer, "Evaluation of fault-tolerant software: a performability modeling approach," in *Dependable Computing for Critical Applications 3, ed. C. E. Landweh, B. Randell and L. Simoncini*, pp.113-135, Sprinter-Verlag, 1993.

[Tomek 1993] L.A. Tomek, J.K. Muppala and K.S. Trivedi, "Modeling correlation in software recovery blocks," *IEEE Trans. Soft. Eng*., vol. 19, no. 11, pp.1071-1086, 1993.

[Tyrrell and Carpenter 1992] A.M. Tyrrell and G.F. Carpenter, “The specification and design of atomic actions for fault tolerant concurrent software,” *Microprocessing and Microprogramming*, vol. 35, pp.363-368, 1992.

[Tyrrell and Holding 1986] A.M. Tyrrell and D.J. Holding, “Design of reliable software in distributed systems using the conversation scheme,” *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 9, pp.921-928, 1986.

[Welch 1983] H.O. Welch, “Distributed recovery block performance in a real-time control loop,” in *Real-Time Sys. Symp.*, pp. 268-276, Virginia, 1983.

[Wood 1981] W. Wood, “A decentralised recovery control protocol,” in *11th Int. Symp. Fault-Tolerant Comput.*, pp. 159-164, 1981.

[Xu et al. 1993] J. Xu, A. Bondavalli and F. DiGiandomenico, *Software fault tolerance: dynamic combination of dependability and efficiency*, Tech. Report, no. 442, Comput. Sci., Univ. of Newcastle upon Tyne, 1993.

[Xu et al. 1994] J. Xu, B. Randell, C.M.F. Rubira-Calsavara and R.J. Stroud, "Towards an object-oriented approach to software fault tolerance," in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Texas, June 1994.

[Yonezawa and Watanabe 1989] A. Yonezawa and T. Watanabe, “An introduction to object-based reflective concurrent computation,” *SIGPLAN Notices*, vol. 24, no. 4, pp.50-53, 1989.