

From Recovery Blocks to Concurrent Atomic Actions*

Brian Randell  Alexander Romanovsky  Cecilia M F Rubira
Robert J Stroud  Zhixue Wu  Jie Xu

University of Newcastle upon Tyne

Abstract. This paper reviews the development of error recovery structures that support general fault tolerance, and describes a new object-oriented scheme for error recovery in concurrent systems that generalizes existing schemes based on either conversations or transactions. This new scheme, which is based on what we term a Coordinated Atomic Action, is intended to facilitate the provision of means of tolerating hardware and software faults, and faults that have affected the environment of the computer system — and to do so for programs that involve cooperating concurrent processes, *and* the use of shared resources.

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 the possibility of residual design faults could not be denied, was taken as an interesting and worthwhile goal. We thus started work on the subject of design fault tolerance.

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 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 an examination of contemporary checkpoint and restart facilities, we came to realize that even though a variety of 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 treating multiple errors, including errors that were detected during the recovery process itself.

The first structuring technique that we developed was in fact the basic *recovery block* scheme. This scheme was soon extended to concurrent processes, via the introduction of the concept of a *conversation* as a means of error recovery in concurrent systems. In what follows we use the object-oriented structuring concepts that have been

* This paper is based largely on material in [129, 176] .

developed during the work of PDCS and PDCS2 projects to describe 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 error recovery in concurrent processes.

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 early on to a style of system design that is based on what we term *idealized fault-tolerant components* [8, 126]. Such components provide a means of system structuring that makes it easy to identify *which* parts of a system have *what* responsibilities for trying to cope with *which* sorts of fault.

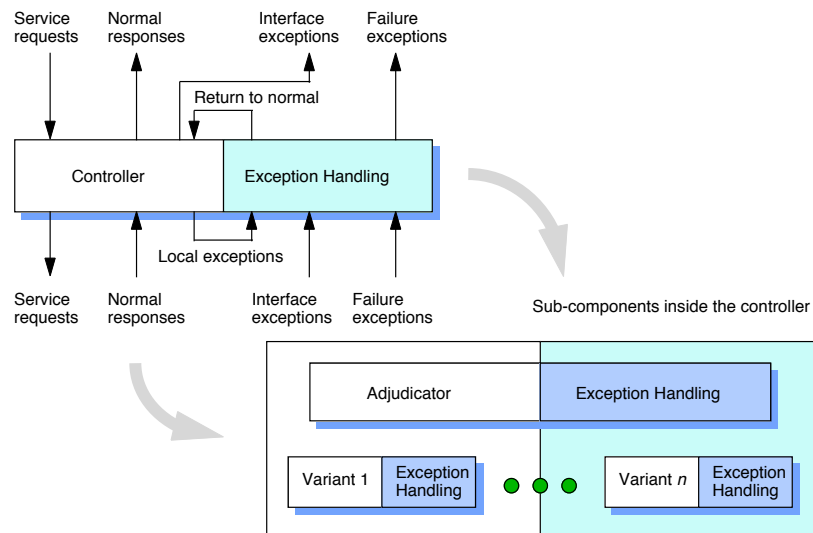


Fig. 1. Idealized component with diverse design.

We view a system as a set of components interacting under the control of a design (which is itself a component of the system) [93]. 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 that 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. exceptional) responses in the interface between interacting components, in a framework that minimizes the impact of these provisions on system complexity.

Three classes of exceptional situation 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 that 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. This is 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 the upper part of Fig. 1).

The notion of an idealized component is mainly concerned with interactions of a component with its environment. The structuring of exception handling is based on minimal assumptions about possible faults, and about the fault masking and the fault tolerance scheme adopted. Exception handling is often considered as being a limited form of software fault tolerance. However, such software cannot be regarded as truly fault-tolerant since some perceived departures from specification are likely to occur, although the exception handling approach can result in software that is robust in the sense that catastrophic failure can often be avoided.

However, 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 described in [128] for describing and comparing the various existing software fault tolerance schemes, and for guiding their actual implementation, is illustrated in the lower part of Fig. 1.

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 that 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 Fig. 1, 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.

3 Basic Recovery Blocks

The basic recovery block is aimed at providing fault-tolerant functional components that may be nested within a sequential program [60, 125]. 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 Fig. 1, and the acceptance test to the adjudicator, while the structure and semantics of the recovery block statement correspond to a description of the controller algorithm. On entry to a recovery block the state of the system must be saved by some underlying mechanism (which we termed a recovery cache) in order to provide backward error recovery. 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 an exit is made from the recovery block, 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 will invoke recovery in the enclosing block.

The overall success of the recovery block (RB) 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 the test is very complex. The development of simple, effective acceptance tests can thus be a difficult task, depending on the actual specification of the component.

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 by 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 idealized 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) that, 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 then invoke another alternate.

In general, as described in [104], 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 [36].) 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 and its environment to a mutually consistent state by sending the environment a pre-defined compensatory message.

Although each of the alternates within a recovery block endeavours to satisfy the same acceptance test, there is no requirement that they must all produce the same results [92]. 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(s) 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 case corresponds to a recovery block that contains a primary module and a null alternate [7, 6]. 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.

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 that the recovery block is invoked. Distributed (parallel) execution of recovery blocks [72] could solve this problem. 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. A 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 extensions natural and minimal.

Current work at Newcastle on such linguistic issues has been greatly influenced by the now very fashionable topic of object-oriented programming. It has been found convenient to try to exploit various characteristics of C++ [159], a language that has been used extensively at Newcastle in connection with work on distributed systems [149]. In particular, 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 mechanisms [136]. This separation makes it feasible to design general reusable software components implementing various fault tolerance strategies (including generalizations and combinations of recovery blocks, and N-version programs [13], and encompassing the use of parallelism), and aids the re-use of application-specific software components [128]. For example, a set of pre-defined C++ classes can be organized to support a general linguistic framework based on the abstract model represented by Fig. 1, and described in Section 2 [136].

4 Extensions and Applications of Basic Recovery Blocks

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

4.1 Distributed Execution of Recovery Blocks

H. Hecht was the first to propose the application of recovery blocks to flight control systems [57]. His work included an implementation of a watchdog timer that monitors the 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. M. Hecht *et al.* [58] reported some subsequent research and experiments.

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 to the uniform treatment of hardware and software faults [72, 74, 73]. A useful feature of their approach is the relatively low run-time overhead it requires, making it 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 the distributed recovery block, forward recovery can be achieved; but the node affected by a fault must invoke backward recovery by executing an alternate for data consistency with the other nodes.

4.2 Consensus Recovery Blocks

The consensus recovery block (CRB) [144] is an attempt to combine the techniques used in the recovery block and N-version programming (NVP). It is claimed that the CRB technique reduces the importance of the acceptance test used in the recovery block and is able to handle cases where NVP would not be appropriate because there are multiple correct outputs. The CRB requires the 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. However, the CRB is largely based on the assumption that there are no common faults between the variants. (This of course is often not the case, as was shown by such experiments as [77].) 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 cause a catastrophic failure.

4.3 Retry Blocks with Data Diversity

The retry block developed by Ammann and Knight [3] 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, but does change the data that the algorithm processes. It is assumed that there are certain data values 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.

4.4 Other Applications

Self-configuring optimal programming (SCOP) [27, 175], developed within PDCS, 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. The gain of efficiency would be limited if the supporting system was intended for a specific application — the hardware resources saved by the SCOP scheme would then be merely left idle. It is perhaps more appropriate if the application environment is complex and highly variable, e.g. a large distributed computing system that supports multiple competing applications.

Sullivan and Masson developed an algorithm-oriented scheme, based on the use of what they term Certification Trails [160]. 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.

5 Concurrent Programs

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. (Some research was also done on error recovery in the particular case of so-called competing processes, i.e. where the processes communicate only for resource sharing [148].) Having identified the dangers of what we came to term the *domino effect*, we came up with the notion of a *conversation* [125] — something that we later realized was a special case of a nested atomic action.

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 [96, 22, 93]. 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 [172]. Planned atomic actions must be maintained by imposing constraints on communication within the system.

When a system of cooperating processes employs recovery blocks, each process will be continually establishing and discarding checkpoints, and may also on occasion need to recover 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 starting points — the domino effect. This causes the loss of the entire computation performed prior to the detection of the error.

The conversation scheme provides a means of coordinating the recovery blocks of interacting processes so as to avoid the domino effect. A conversation, which generally involves two or more processes, constitutes a two-dimensional enclosure of recoverable activities of multiple interacting processes and creates a “time-space boundary” that process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line, and two side firewalls. A recovery line is a coordinated set of recovery points for interacting processes that are relying on backward error recovery. Such a recovery line is established on entry to the conversation before any process interaction occurs. A test line is a correlated set of acceptance tests for the interacting processes. The two side firewalls define exclusive membership; that is, a process inside a conversation cannot interact with a process that is not in the conversation. Fig. 2 shows an example where three processes communicate within a conversation and the processes P1 and P2 communicate within a nested conversation. It is worth particular notice that there may in practice be some means, such as shared resources, by which information will break through the side firewalls and thus defeat the effect of error recovery — this problem is known as “information smuggling”.

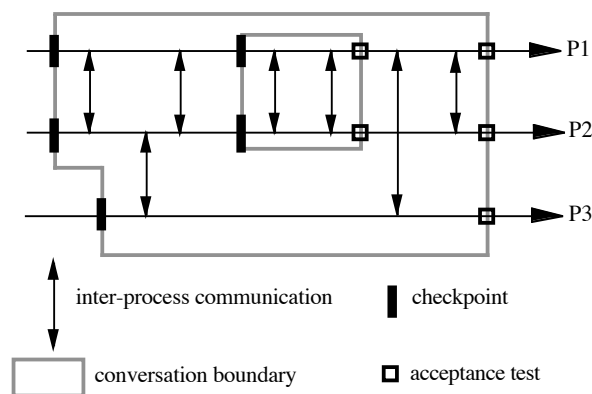


Fig. 2. Nested conversations.

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 [125, 137, 71]. Many authors have added language constructs to facilitate the definition of restorable actions based on this approach [138, 7, 55, 64]. In contrast, the coordination-by-machine approach relies on an “intelligent” underlying processor system that automatically establishes appropriate checkpoints of the interacting processes [70, 19, 79, 75]. 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, 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 [105, 172]. To

reduce synchronization delays introduced by controlled recovery, some researchers have focused on improvements in performance, such as the lookahead scheme and the pseudo-recovery block [138, 147, 74, 124].

6 Error Recovery in Concurrent Object-Oriented Systems

Recently there has been work in PDCS2 on a framework for fault tolerance in concurrent object-oriented programs that integrates conversations, transactions and exception handling, thus supporting the use of both forward and backward error recovery techniques to tolerate hardware and software design faults, and also environmental faults (i.e. faults that exist in or have affected the environment of the computing system).

6.1 Exception Handling, Conversations and Transactions

If an exception is raised by one or several processes within a conversation, then an appropriate error recovery mechanism must be invoked. A coordinated error recovery strategy between *all* the processes in the conversation is required [31]. It is important that all the participating processes have corresponding exception handlers for a given exception (though the use of a default exception handler provided by the underlying system is permitted). These handlers may either invoke appropriate recovery measures, or signal a further exception. (A resolution scheme is used to combine multiple exceptions into a single exception if they are raised at the same time — the multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions.)

In the event of error recovery, the error handlers can use a mixture of forward and backward recovery techniques. For example, the state of a process may be rolled back to the recovery line or compensating actions may be performed to correct the erroneous state. Note that the incorporation of forward error recovery techniques into the conversation framework provides a basis for coordinating the recovery measures taken by the system and its environment (which is typically incapable of simple backward recovery).

A conversation is successful only if all of the interacting processes pass their acceptance tests (and a global test if required) at the test line. If one or more of the interacting processes fails an acceptance test, then an exception is raised and all of the processes within the conversation must attempt to recover. If backward error recovery is used, the original state of each process is restored before allowing participating processes to retry, perhaps using an alternate. (Where the aim is merely to tolerate operational, e.g. hardware, faults such alternates might simply perform a “retry” rather than be of deliberately diverse design.) In principle, if only forward error recovery is used, then there is no need to establish a recovery line on entry to the conversation. However such a recovery line will certainly be needed if there is a requirement to guarantee that a failure of the fault tolerance mechanisms within the conversation leaves the original state of the system unchanged.

The well-established transaction concept is a logical user action that performs a sequence of basic operations on shared data or objects. In general, such shared objects are designed and exist independently of the user processes. A transaction protects shared objects by providing the well known ACID properties — atomicity, consistency, isolation and durability — for all the operations carried out within the

transaction [54]. Nested transactions [109] extend the transaction paradigm by providing the independent failure property for sub-transactions. Therefore, concurrency, i.e. concurrent sub-transactions, may be supported within a transaction. However, unlike a conversation in which multiple processes may enter the conversation asynchronously, concurrency between processes is hidden inside a transaction; that is, just one process can enter the transaction and exit later.

Transactions are usually intended to tolerate hardware-related failures such as node crashes and communication failures, and most transaction mechanisms do not deal with the possibility of software design faults within a transaction that could also be a cause of data inconsistency. Moreover, since transactions hide the effects of concurrency by guaranteeing serialisability, it is not possible for concurrent entities (i.e. interacting processes) to synchronize their activities according to the ordering of their transactions and this could be an additional source of faults.

Conversations provide a framework for programming explicit cooperative concurrency amongst a set of processes or objects that have been designed to interact with each other. In contrast, transactions are used to deal with concurrency implicitly by serialising accesses to objects that are shared by independently designed actions, i.e. objects that have simply been designed to be interacted with (typically termed *shared objects*). Since both kinds of interactions are important, we argue that fault-tolerant concurrent software should combine the mechanisms of conversations and transactions in order to resolve the problems caused by hardware and software faults in the presence of both shared objects and concurrent entities.

Because shared objects have been designed and implemented separately from the applications (i.e. objects) that make use of them, they thus have to be responsible for ensuring their own integrity in the face of concurrent updates and possible failures. However objects that have been designed to interact with each other are collectively responsible for their own integrity. For such objects it may well be possible to use forward error recovery since the designer will know what progress each of the set of objects is intended to make. Backward error recovery can be designed without the need of such knowledge and so is the typical form of recovery used for objects that are individually responsible for their own integrity.

Shared objects that are under the control of a transaction system will guarantee the ACID properties if all the operations on them are performed from within an atomic activity. We will describe these transactional objects as being *atomic* because they provide guarantees of atomicity for objects that interact with them. Interactions via shared objects that are not atomic should occur within the context of a conversation and will require explicit mechanisms for concurrency control and error recovery.

6.2 Object-Oriented Concurrency

Computations are carried out in concurrent systems by the cooperation of several separate (or asynchronous) execution threads. Features for supporting concurrency in an object-oriented (OO) programming language may be added as an extra layer on top of the OO features, or may be fully integrated with the language. We concentrate on the latter because such solutions encompass the concepts of object and process into a single abstraction.

In our proposed framework for coordinated error recovery we view a concurrent OO system as a collection of interacting objects. Concurrent execution threads

correspond to executions of operations on a group of objects. What we are actually concerned with is concurrent executions of operation bodies and coordinated error recovery between a set of such executions. Consequently, there is no need to distinguish between active and passive objects at this level of abstraction. Since a general error recovery mechanism should make no assumptions about the synchronization mechanism that is being used, our model will not specify this mechanism. To avoid extra complexities, we assume in the model that an object must execute just one of its operations at a time. It is therefore conceptually correct by this model to consider objects, rather than individual operations, as participants of a coordinated activity.

6.3 Coordinated Atomic Actions

We use the term *coordinated atomic action* (or CA action) to characterize an activity between a group of interacting objects that combines properties of conversations and transactions and integrates exception handling. Objects that are involved in a CA action and not shared with other actions are called *participating* objects; objects that are shared with more than one CA action are called *external* objects and must be atomic.

A CA action has the following basic properties:

- A CA action that relies on backward error recovery must provide a recovery line in which the recovery points of the objects participating in the action are properly coordinated so as to avoid the domino effect.
- CA actions must provide a test line consisting of a set of acceptance tests, one for each participating object, and a global test for the whole.
- All the objects accessed by a CA action must invoke appropriate forward and/or backward recovery measures cooperatively once an error is detected inside the action, in order to reach some mutually consistent conclusion.
- Error recovery for participating objects in a CA action requires the use of explicit error coordination mechanisms within the CA action; objects that are external to the CA action and can be shared with other actions must be atomic and provide their own error coordination mechanisms (in order to prevent information smuggling).
- Nesting of CA actions is permitted.

On entry to a CA action, a participating object establishes a recovery point if backward error recovery is required and, thereafter, may only communicate with other objects participating in the action and with external objects that are atomic. Note that the participating objects in a particular CA action may enter the action asynchronously. Accessing an external atomic object from within a CA action in effect involves starting some kind of transaction. If all the current participants complete and pass their acceptance tests, then any recovery points taken on entry are discarded, transactions involving external atomic objects are committed and the CA action is exited. If, for any reason, some participating object fails to complete or to satisfy its acceptance test, appropriate recovery measures must be invoked. For this purpose, a CA action is organized as several *CA action attempts*. The first attempt is the normal activity that results from executions of the primary alternates of

cooperative participating objects. Subsequent attempts either consist of the activity of the set of exception handlers, or of the activity of doing backward recovery followed by the next set of alternates. Transactions involving external atomic objects must be aborted during backward error recovery. New transactions started by subsequent attempts may involve different sets of external objects by reason of diverse design. The concept of a CA action thus suggests a quite general solution where both forward and backward recovery techniques can be used in a complementary or combined manner.

Through the use of appropriate protocols it is possible to have a CA action whose participating objects are held in various of the different computers forming a distributed computing system. Indeed users in the environment of a computing system can also be viewed as objects participating in a CA action if they adhere to appropriate protocols — the practicality of this possibility is greatly enhanced by the fact that a CA action can provide a structure and strategy for forward error recovery. For example, the system could send compensatory messages to users in order to correct earlier messages that were later discovered to have been erroneous. In this way, a CA action can effectively deal with cooperative activities between application programs and environments that cannot be rolled back, using forward error recovery.

Fig. 3 shows an example that combines different forms of error recovery into a single CA action in which object 1 uses the exception handler H to do forward recovery while object 2 is rolled back and then tries its second attempt. The effects of operations on external atomic objects are undone completely when the first attempt of the CA action fails.

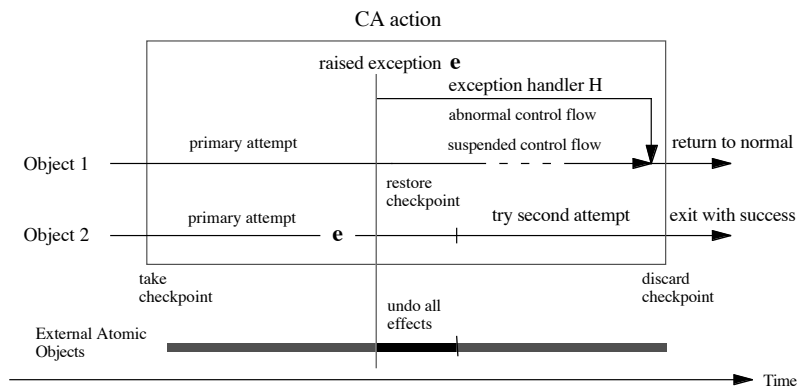


Fig. 3. Combined forms of coordinated error recovery.

The transaction mechanism that supports atomic objects is independent of the mechanism used to implement CA actions, and atomic objects can be being used by different CA actions concurrently. Atomic objects generally contain no design redundancy, but may have their own mechanisms for concurrent access control. Each execution of a CA action behaves like a transaction with respect to the external atomic objects it accesses, and each CA action attempt during execution may be thought of as a nested transaction. Since any effect that a CA action has on external atomic objects shared with other CA actions only becomes visible if the CA action

terminates successfully, unexpected information smuggling between CA actions via external shared objects can effectively be avoided.

CA actions can be nested. A nested CA action is still atomic during its execution (even with respect to its parent and sibling actions). When it completes successfully, its results can only be revealed within its parent action. All the effects of the nested CA action can thus be undone by its parent if the need arises and appropriate recovery points have been taken. Concurrent nested CA actions behave like nested transactions with respect to external atomic objects involved in transactions with their parent action. Thus, although they may be allowed to use the external atomic objects held by their parent action, they must compete for them in a strictly controlled manner. Nested CA actions may also acquire some external atomic objects that are not held by the parent action. However, these external atomic objects cannot be simply released — they should be passed onto the parent action so as to enable possible error recovery. Within a CA action, new objects may be created and then destroyed. If it is necessary to keep the newly created objects after the completion of the creating CA action, the availability of the newly created objects will be strictly limited to the parent action.

6.4 Exception Handling in CA Actions

In object-oriented systems it is appropriate for exceptions to be represented by instances of classes and therefore have a type [37, 78]. This makes it possible to use the type hierarchy to group exceptions together and to define a single handler to cope with a group of related exceptions.

It should be noticed that different participating objects in a CA action may raise different exceptions concurrently. The exception tree proposed by Campbell and Randell [31] is an appropriate mechanism for combining these multiple exceptions into a single exception. For a given exception, the corresponding exception handlers may either invoke appropriate recovery measures, or signal a further exception. Similarly, transactions involving external atomic objects must be either aborted, or else, if practically possible, forward error recovery mechanisms must be used to compensate for any erroneous updates they have made to external atomic objects.

To ensure the proper combination of forward and backward recovery, the CA action structure will guarantee that an exception is raised if the acceptance test fails or a run-time error is detected before the acceptance test is reached. CA actions must be coordinated so as to either produce a result agreeable to all the participating objects or (if at all possible) to restore all the objects changed by the CA action to their prior states. Thus, the default exception handler will typically simply use backward error recovery to terminate the current CA action attempt.

6.5 Linguistic and Implementation Issues

In general, support for CA actions can be provided by either embedding the support into a new language or by extending an existing language. The former approach can offer powerful linguistic constructs and provide a fine degree of control because of its tight integration with the underlying language. An example is the Argus language which provides language constructs for the creation of top-level and nested transactions [94]. But a new language may have difficulty in finding practical acceptance. Providing a set of library objects to support CA actions is the simplest approach to implementation — for example, the Arjuna system [149] uses this

approach to provide a transaction-based toolkit for writing reliable distributed programs in C++. However, the disadvantage of an approach based on the use of library classes is that it does not offer a good degree of control for coordinated actions because there is no linguistic link between the start and end of an action.

Linguistically, a CA action is like a multi-threaded procedure call and has some similarities to the proposal for a *multi-function* made in [17]. The programming language Arche [63] has a construct that supports N-version programming which is a simplified form of a multi-function called a multi-operation. However, a multi-operation call is a mechanism by which a single object can invoke the same operation on a set of objects that implement it in differing ways. In contrast, a CA action allows several different objects to cooperate in performing a task by coming together. Each participating object plays a different role in the CA action, in other words, each object executes a different operation. These roles should be declared as part of the specification of the CA action since the complete set of participating objects in a CA action must be known at run-time, so as to be able to ensure a synchronized exit from the action. Since a CA action is a mechanism by which a group of otherwise unrelated threads can rendezvous, the syntax and semantics for specifying a call to a CA action must make it possible to identify a particular instance of such an action because, unlike a conventional procedure call, a single invocation of a CA action is made up of several different calls. CA actions should be parameterized allowing them to be bound to different objects on each invocation. A further complication is the way in which variants of the different operations within the CA action should be specified. These language design considerations are the subject of on-going research — they are therefore not discussed further in this paper.

The most important implementation issue is the mechanism for coordinating the activity within a CA action. One approach would be to introduce a *CA action manager* object whose basic functions would be: (1) to register asynchronous entries of the participating objects; (2) to manage the transactions used to access external atomic objects; (3) to synchronize the exit of all participants; and (4) to enforce the correct nesting of CA actions.

On invocation of a CA action, i.e. when one or more objects begin to participate in the action, a globally unique identifier for the action must be generated. As each participating object enters the CA action, its identifier is passed onto the manager and recorded in the *Current-Participant-List* of the CA action. Whenever a CA action accesses an external atomic object (that hence is potentially visible to other CA actions executing concurrently), the manager must ensure that this access is recoverable, for example, the CA action must in effect start a transaction in order to access any atomic objects. If backward error recovery is being used, the manager is also responsible for establishing a recovery point for each participating object as the object enters the CA action. If the action completes successfully, any such recovery points are discarded; otherwise the previous states of the participating objects are restored and some recovery measures are invoked. The CA action may terminate with a failure exception despite the use of its own fault tolerance capabilities. Since CA actions can be nested, a failure exception of a sub-CA action will simply cause termination of the current attempt of the enclosing CA action. The outer CA action will then invoke appropriate recovery.

External atomic objects may be accessed concurrently by different CA actions and must have the semantics of atomic data types [169]. Either optimistic or pessimistic concurrency control policies can be used to implement atomic data types [169, 59]. The simplest approach is to lock all atomic objects exclusively for use only within a single CA action. This can be relaxed somewhat by allowing concurrent access to external atomic objects from several CA actions provided that none of them tries to modify such objects. Allowing concurrent updates to external atomic objects requires type-specific knowledge about the semantics of the atomic data type to prevent conflicts. However, note that concurrency control and error recovery for external atomic objects are the responsibility of those objects and not the CA actions that access them concurrently.

If exception handling is used to implement forward error recovery, a participating object may raise an exception during the execution of an operation within the CA action or if it fails its acceptance test at the end of the CA action. If an exception is raised, all the participating objects in the CA action should stop their normal computation and the process of exception resolution must be started. Any such exception must be first caught by the manager object which will then inform other participating objects that an exception has occurred so as to stop other normal computations. If several exceptions are raised concurrently, a resolution function is used to decide which single exception covers the entire set. Appropriate steps are then taken to handle that exception.

Finally, participating objects in a nested CA action may access external atomic objects that are already held by their parent CA action, but this must be done in a strictly controlled way in order to prevent information smuggling. A set of rules must be designed carefully, enforced and checked by the action manager. For example, once external atomic objects have been passed onto a nested CA action by the parent CA action, the parent action should not be allowed to access the external atomic objects until the nested action terminates.

7 Concluding Remarks

Although we believe that OO concepts provide a valuable perspective on the design of error recovery structures that support the provision of design fault tolerance in sophisticated programs, we have made only rather limited use of these ideas in this paper. Space constraints have precluded a more detailed discussion of the advantages that are provided by various object-oriented linguistic constructs (see however [125, 128, 136, 177], and also Paper III.G in this volume).

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 them, but we are very pleased to acknowledge our indebtedness to all of them.

References for Chapter III (extracts)

- [3] P. E. Ammann and J. C. Knight, "Data Diversity: An approach to software fault tolerance", *IEEE Trans. Comput.*, 37 (4), pp.418-25, 1988.
- [6] T. Anderson, P. A. Barrett, D. N. Halliwell and M. R. Moulding, "Software Fault Tolerance: An evaluation", *IEEE Trans. Software Engineering*, SE-11 (12), pp.128-34, 1985.
- [7] T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems", *IEEE Trans. Soft. Eng.*, SE-9 (3), pp.355-64, 1983.
- [8] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and practice*, Prentice Hall, 1981.
- [13] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Execution", in *Int. Conf. Comput. Soft. and Applic.*, (New York), pp.149-55, 1977.
- [17] J. P. Banâtre, M. Banâtre and F. Ployette, "The Concept of Multi-Functions: A general structuring tool for distributed operating systems", in *Proc. 6th Int. Conf. Distributed Computing Systems*, pp.478-85, 1986.
- [19] G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points", in *Proc. 13th Int. Symp. on Fault-Tolerant Computing (FTCS-13)*, (Milan), IEEE Computer Society Press, 1983.
- [22] E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems", *Acta Informatica*, 16, pp.93-124, 1981.
- [27] A. Bondavalli, F. Di Giandomenico and J. Xu, "A Cost-Effective and Flexible Scheme for Software Fault Tolerance", *J. of Computer Systems Science and Engineering*, 8 (4), pp.234-44, October 1993.
- [31] R. H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems", *IEEE Trans. Software Engineering*, SE-12 (8), pp.811-26, 1986.
- [36] F. Cristian, "Exception Handling and Software Fault Tolerance", *IEEE Trans. on Computers*, C-31 (6), pp.531-40, 1982.
- [37] F. Cristian, "Exception Handling", in *Dependability of Resilient Computers* (T. Anderson, Ed.), pp.68-97, Blackwell Scientific Publications, 1989.
- [38] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast: From simple message diffusion to Byzantine agreement", in *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, (Ann Arbor, Michigan), pp.200-6, IEEE Computer Society Press, 1985.
- [54] J. Gray and A. Reuter, *Transaction Processing: Concepts and techniques*, Morgan Kaufmann, 1993.
- [55] S. T. Gregory and J. C. Knight, "A New Linguistic Approach to Backward Error Recovery", in *Proc. 15th Int. Symp. Fault-Tolerant Computing (FTCS-15)*, (Michigan), pp.404-9, IEEE Computer Society Press, 1985.

- [57] H. Hecht, "Fault-Tolerant Software for Real-Time Applications", *ACM Computing Surveys*, 8 (4), pp.391-407, 1976.
- [58] M. Hecht, J. Agron, H. Hecht and K. H. Kim, "A Distributed Fault-Tolerant Architecture for Nuclear Reactor and Other Critical Process Control Applications", in *Proc. 21st Int. Symp. Fault-Tolerant Computing (FTCS-21)*, (Montreal), pp.462-9, IEEE Computer Society Press, 1991.
- [59] M. Herlihy, "Apologizing Versus Asking Permission: Optimistic concurrency control for abstract data types", *ACM Trans. DataBase Systems*, 15 (1), pp.96-124, 1990.
- [60] 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*, 16, pp.177-93, 1974.
- [63] V. Issarny, "An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards reusable, robust distributed software", *Journal of Object-Oriented Programming*, 6 (6), pp.29-40, 1993.
- [64] P. Jalote and R. H. Campbell, "Atomic Actions for Fault Tolerance using CSP", *IEEE Trans. Soft. Eng.*, SE-12 (1), pp.59-68, 1986.
- [70] 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.
- [71] K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors", *IEEE Trans. Soft. Eng.*, SE-8 (3), pp.189-97, 1982.
- [72] K. H. Kim, "Distributed Execution of Recovery Blocks: An approach to uniform treatment of hardware and software faults", in *Proc. 4th Int. Conf. Distributed Comput. Sys.*, pp.526-32, 1984.
- [73] 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.*, C-38 (5), pp.626-36, May 1989.
- [74] K. H. Kim and J. C. Yoon, "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", in *Proc. 18th Int. Symp. Fault-Tolerant Computing (FTCS-18)*, (Tokyo), pp.50-5, IEEE Computer Society Press, 1988.
- [75] K. H. Kim and J. H. You, "A Highly Decentralized Implementation Model for the Programmer-Transparent Coordination (PTC) Scheme for Cooperative Recovery", in *Proc. 20th Int. Symp. Fault-Tolerant Computing (FTCS-20)*, (Newcastle), pp.282-9, IEEE Computer Society Press, 1990.
- [77] J. C. Knight, N. G. Leveson and L. D. S. Jean, "A Large Scale Experiment in N-Version Programming", in *Proc. 15th Int. Symp. Fault-Tolerant Computing (FTCS-15)*, (Michigan), pp.135-40, IEEE Computer Society Press, 1985.
- [79] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Trans. Soft. Eng.*, SE-13 (1), pp.23-31, 1987.

- [92] P. A. Lee, "A Reconsideration of the Recovery Block Scheme", *Computer Journal*, 21 (4), pp.306-10, 1978.
- [93] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and practice*, Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, Vienna, 1990.
- [94] B. Liskov, "Distributed Programming in Argus", *Comm. ACM*, 31 (3), pp.300-12, 1988.
- [96] D. B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions", *ACM SIGPLAN Notices*, 12 (3), pp.128-37, 1977.
- [104] P. M. Melliar-Smith and B. Randell, "Software Reliability: The role of programmed exception handling", in *Proc. Conf. on Language Design For Reliable Software (ACM SIGPLAN Notices, vol. 12, no. 3, March 1977)*, (Raleigh), pp.95-100, ACM, 1977.
- [105] P. M. Merlin and B. Randell, "State Restoration in Distributed Systems", in *Proc. 8th Int. Symp. Fault-Tolerant Computing (FTCS-8)*, (Toulouse), pp.129-34, IEEE Computer Society Press, 1978.
- [109] J. E. B. Moss, *Nested Transactions: An approach to reliable distributed computing*, MIT Press, 1985.
- [124] P. Ramanathan and K. G. Shin, "Checkpointing and Rollback Recovery in a Distributed System using a Common Time Base", in *Proc. 7th Symp. Rel. Distrib. Syst.*, (Columbus), pp.13-21, 1988.
- [125] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Trans. on Software Engineering*, SE-1 (2), pp.220-32, 1975.
- [126] B. Randell, "Fault Tolerance and System Structuring", in *Proc. 4th Jerusalem Conf. on Information Technology*, (Jerusalem), pp.182-91, 1984.
- [128] B. Randell and J. Xu, "Object-Oriented Software Fault Tolerance: Framework, reuse and design diversity", in *1st PDCS2 Open Workshop*, (Toulouse, France), pp.165-84, 1993.
- [129] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept", in *Software Fault Tolerance* (M. Lyu, Ed.), Trends in Software, pp.1-22, J. Wiley, 1994.
- [136] C. M. F. Rubira-Calsavara and R. J. Stroud, "Forward and Backward Error Recovery in C++", *Object-Oriented Systems*, 1 (1), pp.61-85, 1994.
- [137] D. L. Russell, "State Restoration in Systems of Communicating Processes", *IEEE Trans. Soft. Eng.*, SE-6 (2), pp.183-94, 1980.
- [138] D. L. Russell and M. J. Tiedeman, "Multiprocess Recovery using Conversations", in *Proc. 9th Int. Symp. Fault-Tolerant Computing (FTCS-9)*, pp.106-9, IEEE Computer Society Press, 1979.
- [144] R. K. Scott, J. W. Gault and D. F. Mcallister, "The Consensus Recovery Block", in *Proc. Total Sys. Reli. Symp.*, pp.74-85, 1985.

- [147] K. G. Shin and Y.-H. Lee, "Evaluation of Error Recovery Blocks used for Cooperating Processes", *IEEE Trans. Soft. Eng.*, SE-10 (6), pp.692-700, 1984.
- [148] S. K. Shrivastava and J.-P. Banâtre, "Reliable Resource Allocation Between Unreliable Processes", *IEEE Trans. Soft. Eng.*, SE-4 (3), pp.230-41, 1978.
- [149] S. K. Shrivastava, G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, 8 (1), pp.66-73, January 1991.
- [160] G. Sullivan and G. Masson, "Using Certification Trails to Achieve Software Fault Tolerance", in *Proc. 20th Int. Symp. Fault-Tolerant Computing (FTCS-20)*, (Newcastle), pp.423-31, IEEE Computer Society Press, 1990.
- [169] W. E. Weihl and B. Liskov, "Implementation of Resilient, Atomic Data Types", *ACM Trans. Programming Languages and Systems*, 7 (2), pp.244-69, 1985.
- [172] W. Wood, "A Decentralised Recovery Control Protocol", in *Proc. 11th Int. Symp. Fault-Tolerant Computing (FTCS-11)*, pp.159-64, IEEE Computer Society Press, 1981.
- [175] J. Xu, A. Bondavalli and F. Di Giandomenico, *Software Fault Tolerance: Dynamic combination of dependability and efficiency*, Univ. of Newcastle upon Tyne, Tech. Report, N°442, 1993.
- [176] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu, "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery", in *Proc. 25th Int. Symp. Fault-Tolerant Computing (FTCS-25)*, (Los Angeles), IEEE Computer Society Press, 1995.
- [177] J. Xu, B. Randell, C. M. F. Rubira and R. J. Stroud, "Toward an Object-Oriented Approach to Software Fault Tolerance", in *Fault-Tolerant Parallel and Distributed Systems* (D. R. Avresky, Ed.), IEEE Computer Society Press, 1994.