Coordinated Atomic Actions: from Concept to Implementation

B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, and A.F. Zorzo

Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK

ABSTRACT

The Coordinated Atomic Action (or CA action) concept is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. It provides a conceptual framework for dealing with different kinds of concurrency and achieving fault tolerance by extending and integrating two complementary concepts — conversations and transactions. Conversations (enhanced with concurrent exception handling) are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

This paper explains the CA action concept in detail and then addresses related design issues such as multi-thread coordination, exception handling and resolution, coordinated access to shared objects and provision of software fault tolerance. Finally, brief details are given of a number of experimental prototype implementations and case studies.

Key Words — Atomicity, concurrency models, nested transactions, object-oriented programming, (software) fault tolerance.

1 Introduction

1.1 Background

Distributed computing often gives rise to complex concurrent and interacting activities. In some cases several concurrent activities may be working together, i.e. *cooperating*, to solve a given problem; in other cases the activities can be completely *independent* or may be essentially independent though needing to *compete* for shared common system resources. In practice, different kinds of concurrency might co-exist in a complex application which thus will require a general supporting mechanism for controlling and coordinating complex concurrent activities.

Due in no small measure to their complexity, concurrent and distributed systems are very prone to faults and errors. Various fault tolerance techniques for coping with hardware and software faults can provide a practical way of improving the dependability of such systems. Moreover, because certain faults can have an impact on, or arise from, the environment of a computing system [Campbell and Randell 1986], some forms of error recovery may require stepping outside the boundaries of a computer system (i.e. considering the computer system and its environment recursively as an entire system at a higher level of abstraction). However, in reality the majority of fault-tolerant computing systems do not attempt to tolerate software faults, or facilitate recovery from errors that affect both the computer system and its environment — rather they concentrate on the problems that arise from operational faults (typically hardware faults). For example, many software systems that use the concept of atomic (trans)actions to construct fault-tolerant distributed applications generally assume that user programs are correct.

Increasingly, software is constructed according to the object-oriented programming methodology which supports clean structuring, simplicity of design and software reuse. If used correctly, object-oriented programming can improve software dependability.

However, given the complexity of today's computing systems, it still seems inevitable that a complex system will contain some residual (typically software) design faults or “bugs”. Approaches and mechanisms, such as software fault tolerance and exception handling, are therefore required in order to cope with software bugs and abnormal run-time events. Practical techniques for dealing with software faults do exist, especially for sequential systems, and have been proved successful for a variety of applications [Lyu 1995]. The challenge is to extend these techniques to cope with the full potential complexity of a distributed system.

1.2 Motivation

In many real-world applications there are specific needs for cooperation and coordination among multiple concurrent activities. It is becoming ever clearer that the traditional (nested) transaction model does not provide satisfactory support for cooperation between concurrent activities. Researchers in the area of transactions and databases [Gray and Reuter 1993] are aware of such limitations and problems:

> “The transaction concept has emerged as the key structuring technique for distributed data and distributed computations. Originally developed and applied to database applications, the transaction model is now being used in new application areas ranging from process control to cooperative work. Not surprisingly, these more sophisticated applications require a refined and generalized transaction model. The concept must be made recursive, it must deal with concurrency within a transaction, it must relax the strict isolation among transactions, and it must deal more gracefully with failures.”
>
> *Jim Gray* (Foreword for [Elmagarmid 1993])

For process-oriented systems, such as process control or real-time control applications, the best-known approach to structuring dependable concurrent systems to facilitate error recovery is the *conversation* concept [Randell 1975], an approach that provides full support for cooperative concurrent activities, and which was extended to cover forward error recovery via coordinated exception handling in [Campbell and Randell 1986]. However, without special support for object interactions and consistent access to shared objects, it has proved difficult to use the conversation concept to control concurrency and facilitate error recovery in an object-oriented system [Gregory and Knight 1989][Xu, Randell et al 1995a].

In this paper we discuss the problem of structuring dependable distributed object-oriented systems using a general scheme, called *Coordinated Atomic Actions* (or CA actions), that extends and integrates two complementary concepts, conversations and transactions, to support different kinds of (independent, competitive, and cooperative) concurrency and to provide fault tolerance. The CA action concept (first introduced in [Xu, Randell et al 1995a]) thus encompasses strategies for dealing with hardware, software and environmental faults to provide coordinated error recovery between a set of interacting objects, both inside a set of computing systems, and in their external world.

1.3 Coordinated atomic actions: overview and example

A CA action is a mechanism for coordinating multi-thread interactions and ensuring consistent access to objects in the presence of competitive concurrency and potential faults. In order to support backward error recovery, a CA action must provide a recovery line which coordinates the recovery points of the objects and execution threads participating in the action so as to avoid the *domino effect* [Randell 1975]. To support forward error recovery, a CA action must provide an effective means of coordinating the use of exception handlers. An acceptance test can and ideally should be provided in order to determine whether the outcome of the CA action is successful. The various threads participating in a given CA action can enter the action asynchronously but their exits from the CA action must be synchronized, albeit perhaps just logically. If an error is detected inside a CA action, appropriate forward and/or backward recovery measures must be invoked cooperatively, in order to reach some mutually consistent conclusion.

Error recovery for participating threads of a CA action generally requires the use of explicit error coordination mechanisms within the CA action (e.g. the conversation scheme plus concurrent exception handling [Campbell and Randell 1986]); objects that are external to the CA action and can be shared with other actions must provide their own error coordination mechanisms. These external objects must behave atomically with respect to other CA actions so that they cannot be used as an implicit means of "smuggling" information [Kim 1982] into or out of a CA action.

Figure 1 shows a simple example in which two threads enter a CA action asynchronously through different entry points. Within the CA action the threads communicate with each other and cooperate in pursuit of some common goal. However, during the execution of the CA action, an exception e is raised by one of the threads. The exception is propagated to the other thread and both threads transfer control to their exception handlers H_1 and H_2 which attempt to perform forward error recovery. The effects of erroneous operations on *external objects* are repaired by putting the objects into new correct states so that the CA action is able to pass its acceptance test and exit with a successful outcome. (As an alternative to performing forward error recovery, the two participating threads could undo the effects of operations on the external objects, roll back and then try again, possible using diversely designed software alternates.)

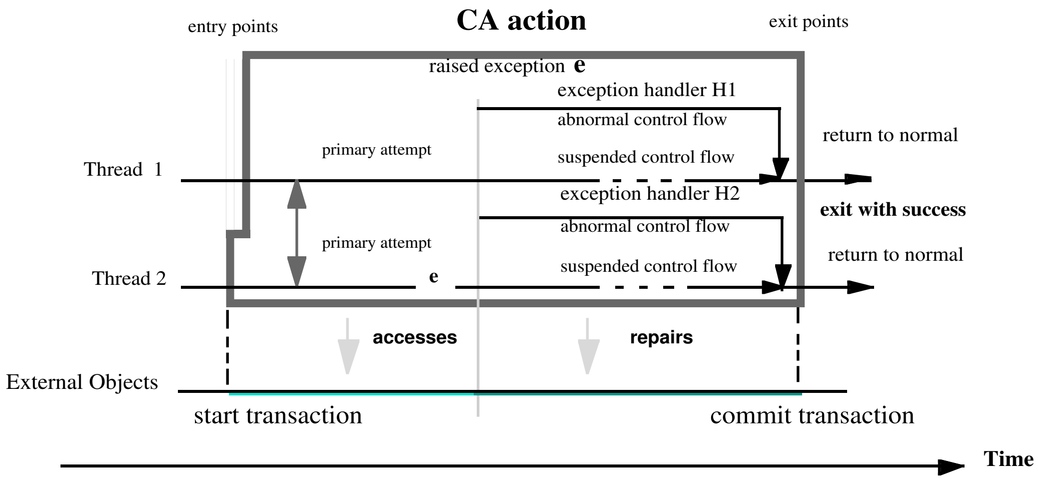

Figure 1 Coordinated error recovery performed by a CA action.

We have explored the CA action concept using various interesting examples of cooperative activities. Many of our early discussions made use of the "Hamlet metaphor" — something we recall Kristen Nygaard using years ago in a lecture on Simula. (The Hamlet metaphor involves such interesting complications as multiple concurrent performances of the same play in different theatres, a play within a play, and (through Rosencrantz and Guildenstern's entrances and exits) two separate interlaced sets of concurrent cooperative actions.) Among the more practical examples we have examined, and in several cases made simple experimental prototypes of, are banking applications (e.g. joint accounts and credit card authority checks, etc.), sales control systems, and cooperative activities in CAD/CAM and office automation. Computer supported cooperative work (CSCW) systems [Ellis, Gibbs et al 1991] are another area in which we believe the CA action concept could be applied.

In order to show how the CA action concept could be applied to a real-life example, we will briefly discuss a simple banking application. Consider a funds transfer system implemented as a set of interacting objects with a transaction processing system responsible for ensuring consistent access to those objects. The most typical objects in such a system would be *bank account* objects. An account object would have an internal state that recorded the current balance of the account and an associated set of operations, such as *credit*, *debit*, *check-balance* (that takes a value as argument and returns *TRUE* if the balance of the account is greater than or equal to the argument value, else it returns *FALSE*), *read-balance* etc. Assume that clients of the funds transfer system (say, *Mary* and *John*) can have various types of accounts contained in the system and can issue various kinds of transactions (or actions) to obtain the expected service.

Suppose *Mary* and *John* have two shared (joint) accounts, *acc-A* and *acc-B*. If one of them, say *John*, wishes to withdraw £1000 from these joint accounts, a cooperative action, rather than an independent action, will have to be taken if the pre-defined condition is that no single person can withdraw any money from a joint account. Figure 2 illustrates this example, assuming that the authority check can examine PINs, or signatures or both. After passing the authority checks, *Mary* and *John* check the balances of *acc-A* and *acc-B*, respectively. Because there is insufficient money in *acc-A*, they communicate privately within the CA action and agree to withdraw the money from *acc-B*. Before leaving the action, they communicate again to make sure that only the agreed amount of money is withdrawn.

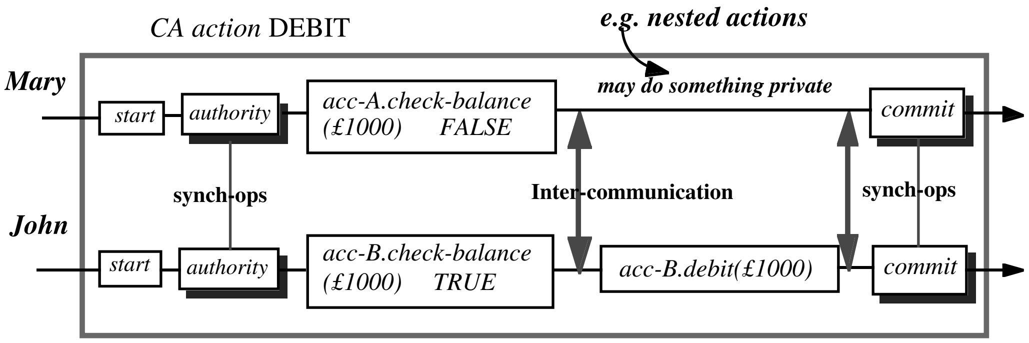

Figure 2 A cooperative action DEBIT.

From the viewpoint of the rest of the system, and in particular other users, the effects of the DEBIT CA action on the account objects are the same as the effects a single-threaded transaction would have. Yet DEBIT involves multiple concurrent threads cooperating in order to produce some mutually acceptable result (and possibly detecting and recovering from errors that occur while this is being done) — something that cannot be readily achieved by the use of conventional transaction-based systems.

1.4 Related work

The traditional domain for transaction processing is database systems [Gray 1978]. C.T. Davies pioneered the development of the atomic transaction concept [Davies 1973][Davies 1978]. He addressed many concepts concerned with concurrent systems, recovery and integrity within an overall scheme that he called *data processing spheres of control*. Spheres of control are intended to deal with various problems including coordinating multiple processes within recovery regions, sharing partial (uncommitted) data between processes, and controlling concurrency across machine boundaries. However, the descriptions of spheres of control provided little implementation advice for general applications, and early work on transactions, though influenced by Davies, was much less ambitious in its goals.

Basic transaction systems do not allow for subtransactions or support concurrency within a transaction. Nested transactions [Moss 1981] extend the flat transaction model by providing the independent failure property for subtransactions, and supporting competitive concurrency within a containing transaction. A number of generalized transaction models have been developed recently in order to overcome some of the limitations of traditional (flat or nested) transactions, such as lack of support for long-lived actions, cooperative activities and multidatabase systems. Much of this work is surveyed comprehensively in [Elmagarmid 1993].

Several systems have been developed that successfully combine transaction processing with the object-oriented programming methodology, e.g. Argus [Liskov 1988] and Arjuna [Parrington, Shrivastava et al 1995]. These systems offer support for nested transactions and provide a powerful linguistic base for developing reliable distributed programs. Competitive concurrency is well controlled and fully supported in such systems, but no support is provided for cooperative concurrency.

Conversations and their variations [Randell 1975][Campbell and Randell 1986][Randell and Xu 1995] provide efficient support for cooperative activities in process-oriented systems, and moreover pay much attention to the problem of error recovery and fault tolerance. The Arche language [Issarny 1993] extended some ideas behind the conversation concept to provide support for object replication and concurrent exception handling in a distributed object-oriented environment, using a construct called a multi-operation, which is a simplified form of the multi-function construct introduced in [Banatre, Banatre et al 1986].

Recently, there have been a number of proposals for "multiparty interaction mechanisms" [Jung and Smolka 1996], intended to coordinate interactions among multiple concurrent processes in distributed programming, e.g. Interacting Processes (or IP) [Francez and Forman 1996]. However, to the best of our knowledge, none of the proposals published to date provides a means for fault tolerance or for avoiding interference from unrelated processes and parties.

1.5 Organization of this paper

The remainder of this paper is organized as follows. Section 2 contains a description of our basic model and the CA action concept. Section 3 addresses related design issues. Section 4 summarizes the results of our work on various prototypes and implementations. Section 5 concludes the paper.

2 Basic concepts and conceptual model

In this section we introduce the basic concepts behind CA actions and present a conceptual model for CA actions that is intended to be neutral with respect to both programming language issues and implementation issues (although for convenience we present the model in object-oriented terms). Thus, we describe our view of concurrency in an object-oriented environment and explain how CA actions can be used as a mechanism for supporting both cooperative and competitive concurrency whilst tolerating both hardware and software faults. We also discuss issues such as nesting and parameterisation of CA actions.

2.1 Objects, threads and synchronization

There are many different ways of dealing with concurrency in object-oriented systems and we prefer to maintain a neutral position by concentrating on run-time mechanisms rather than linguistic constructs. Thus, we wish to avoid choosing any particular model of active objects or postulating any particular relationship between objects and threads. Instead, we feel that it is useful to maintain a distinction, at least conceptually, between threads as the agents of computation and objects as the subject of computation. In practice, the distinction between threads and objects can become rather blurred. For example, in some programming models, threads are just objects with a `run` method that is invoked asynchronously. But in our view, this is just a convenient way of structuring the code and not a conceptual unification.

In an object-oriented system, computation proceeds by invoking methods on objects. In a concurrent system, there are multiple threads of computation, in other words, multiple threads of method invocation. The existence of multiple threads immediately introduces the possibility of two or more method invocations being active within the same object at the same time. Without adequate synchronisation mechanisms, the result would be chaos.

Two levels of synchronisation are required. Firstly, objects must be responsible for ensuring their own integrity in the face of concurrent access. In other words, objects that can be accessed simultaneously from more than one thread need to provide some kind of monitor semantics, e.g. "*N* readers, 1 writer", in order to guarantee non-interference on a per object basis. But this is not sufficient if threads need to perform a set of operations on a set of objects without interference from other threads. This requires a stronger guarantee of non-interference such as serializability. CA actions are intended to be a mechanism for providing this guarantee. A CA action performs a set of operations on a group of objects atomically, and thus behaves like a transaction. However, the body of a CA action can be multi-threaded. In other words, CA actions also behave like conversations in that they allow a set of threads to come together in order to perform some action atomically. Thus, CA actions unify the concepts of transactions and conversations by providing a framework that supports both cooperative and competitive concurrency.

2.2 Properties of CA actions

A CA action provides a mechanism for performing a group of operations on a set of objects atomically. These operations are performed cooperatively by one or more roles executing in parallel within the CA action. The interface to a CA action specifies the objects that are to be manipulated by the CA action and the roles that are to manipulate these objects. In order to perform a CA action, a group of threads must come together and agree to perform each role in the CA action concurrently with one thread per role. (For more details, see section 2.5.)

The multiple threads within a CA action communicate with each other via local objects that are purely internal to the CA action and are used to support cooperative concurrency. In this way, the threads within a CA action can coordinate their concurrent activities and agree upon the set of operations they wish to perform upon the objects that are manipulated by the CA action. These objects are considered to be external to the CA action and can therefore be accessed competitively by other CA actions executing concurrently. To ensure correctness and prevent information smuggling, a CA action must behave like a transaction with respect to these external objects. Thus, the effects of any operations that the threads within a CA action perform on external objects are not visible to other threads or CA actions until that CA action terminates.

As discussed in Section 1.1, transactions are designed to deal with the problems of concurrency and hardware faults, and it is generally assumed that software faults are not an issue. In other words, if a transaction commits it is assumed to have produced the correct results and a subsequent transaction will not have to abort because of an error in the first transaction. In contrast, CA actions are intended to provide a more general framework for dealing with both hardware and software faults that combines mechanisms for forward and backward error recovery. Thus, it is recognised that the actions performed by the threads in a CA action may not be successful, and it is possible to use various techniques within the CA action to reduce the chance of failure.

The desired effect of performing a CA action is described by an acceptance test which is structured in terms of a normal outcome and a series of exceptional (or degraded) outcomes [Cristian 1989]. The effects of performing a CA action only become visible if the acceptance test is passed. This is analogous to transaction commit except that the acceptance test allows both a normal outcome and one or more exceptional outcomes, with each exceptional outcome signalling a specified exception to the surrounding environment. Conversely, if it is not possible to satisfy the acceptance test at the end of a performance, even by signalling one of the specified exceptions, the CA action is considered to have failed. This is analogous to transaction abort and it is therefore necessary to undo the potentially visible effects of the CA action in this case and signal an `abort` exception to the surrounding environment. If the CA action is unable to satisfy the "all or nothing property" necessary to guarantee atomicity (e.g. because the undo fails), then a `failure` exception must be signalled to the surrounding environment indicating that the CA action has failed to pass its acceptance test and that its effects have not been undone. The system has probably been left in an erroneous state and this must be dealt with by the enclosing CA action (if any).

Each of the threads involved in a given performance of a CA action should receive an indication of the result: a normal outcome, an exceptional outcome, an `abort` exception, or a `failure` exception. It is important that all of the threads should agree about the outcome. If the threads fail to reach agreement, then the CA action mechanism itself has failed and this failure must be dealt with at a higher level in the system. Some of the issues involved in ensuring the necessary thread coordination to guarantee these semantics are discussed in Section 3.1.

2.3 Fault tolerance properties

CA actions provide a basic framework that can support a variety of fault tolerance mechanisms. Hardware faults can be tolerated using two phase commit protocols and stable storage to ensure that the effects of CA actions are permanent. Software faults can be addressed using fault masking and design diversity. Environmental faults will typically be dealt with using forward recovery strategies that involve sending compensatory messages between the system and its environment, unless the objects in the system's environment are capable of backward recovery.

During the execution of a CA action, one of the threads that are involved in the action may raise an exception. If that exception cannot be dealt with locally by the thread, then it must be propagated to the other threads involved in the CA action. Since it is possible for several threads to raise an exception at more or less the same time, a process of exception resolution [Campbell and Randell 1986] is necessary in order to agree on the exception to be propagated and handled within the CA action (see Section 3.2).

Once an agreed exception has been propagated to all of the threads involved in the CA action, then some form of error recovery mechanism must be invoked. It may still be possible to complete the performance of the CA action successfully using forward error recovery. Conversely, it may be possible to use backward error recovery to undo the effects of the CA action and start again, perhaps using a different variant of each role in order to tolerate design faults (see Section 3.4). If it is not possible to achieve either a normal outcome or an exceptional outcome using these error recovery mechanisms, then the CA action should be aborted and its effects should be undone.

It is important that these fault tolerance measures are properly integrated with the effects that the CA action is having on objects, both internal and external to the CA action. For example, backwards error recovery must involve restoring the state of all the objects involved in the CA action, including external objects, whilst forward error recovery must ensure that all of the objects are left in an acceptable state. Otherwise, a failure exception will be signalled to the external environment.

2.4 Nesting

The CA action concept is intended to be recursive. In other words, nested CA actions can be used to structure the performance of an enclosing CA action. As with nested transactions, the effects of a nested CA action only become permanent when the top level enclosing CA action terminates. However, the effects of a nested CA action are visible to the enclosing CA action once the nested action has completed.

It is important to realise that the notion of a top-level action is relative to the current level of abstraction — for example, it could be that an apparently top-level CA action is in fact nested within an enclosing CA action in a wider system view. For this reason, it is perhaps better to think in terms of the effect of committing a CA action at a given level of nesting as being to pass the responsibility for recovering the effects of the action (if necessary) to the enclosing CA action (if any). As noted in Section 1, some forms of recovery may involve stepping outside the boundaries of the computer system.

Adding support for nested CA actions to the basic model introduces a number of complications, as well as benefits. The threads within a CA action coordinate their activities via a set of local objects that are internal to the CA action. However, with respect to a nested CA action, these shared objects are external objects and thus the rules for interacting with them must be different. If a nested CA action interacts with objects that are external to itself, then those interactions must appear to be atomic with respect to other nested CA actions and the enclosing CA action. In effect, this means that the nested CA action must enter into a nested transaction with any objects it accesses externally, even if such objects are not external to the enclosing CA action (see Figure 3).

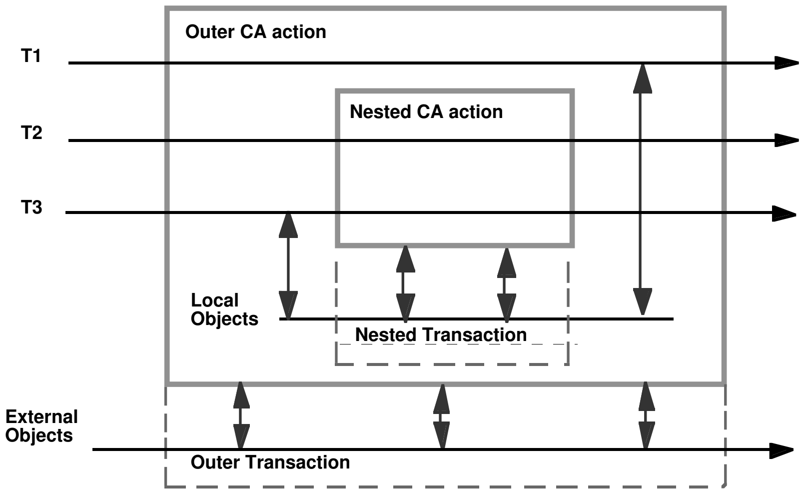

Figure 3 Nesting of CA actions.

Note that because CA actions are multi-threaded and because not all the threads within a CA action are necessarily involved in a nested CA action, there is a very real possibility of competitive concurrency arising between the nested CA actions and the threads within an enclosing CA action. The use of nested transactions can guard against the possibility of interference and guarantee serializability. Meanwhile, the threads within the enclosing CA action are responsible for coordinating the actions performed by nested CA actions and thus ensuring cooperation towards a common goal. Thus, nesting allows CA actions to support both cooperative and competitive concurrency at different levels of abstraction. These issues are discussed further in Section 3.3.

2.5 Parameterisation, binding and instantiation

In a conventional programming language, a procedure is an abstraction mechanism describing a pattern of computation. Procedures are parameterised and describe how a result can be produced from a given input but they do not identify a particular input. Instead, the inputs to a procedure are supplied at the time that the procedure is called and the same procedure can be called many different times with different inputs each time.

A CA action is intended to provide an abstraction rather like a multi-threaded procedure call. The different threads within the body of the CA action are described by roles and these roles must be performed concurrently in order to perform the CA action. The roles can be parameterised so the inputs to the CA action are only determined at the moment of performance. However, the same CA action can be performed many times, with different inputs each time.

Despite this similarity between a procedure and a CA action, there is an important difference which arises from the multi-threaded nature of a CA action. In order to call a procedure, a thread must specify the name of the procedure and the values of any arguments to be passed to the procedure. But each such procedure call results in a different execution of the procedure with a different set of arguments.

Unfortunately, the same model does not apply to CA actions. Each call to a CA action from a thread must name the particular role within the CA action that the thread is to perform and specify any arguments that need to be passed to that role. But it is also necessary to identify the particular performance of the CA action because each performance involves more than one thread. A performance of a CA action will only take place if a thread can be found to play each role — in other words, a performance of a CA action only takes place when the appropriate number of threads have made a call to that CA action, each specifying a different role within the same performance. This implies the need for a mechanism to identify a particular performance of a CA action — such a mechanism is not necessary for procedures because only one thread is required to make a procedure call and consequently every procedure call made by a thread results in a distinct execution of that procedure.

To identify a particular performance of a given CA action it is necessary to use some kind of token. Any thread that has access to this token can participate in the performance providing it can also identify a unique role within the performance. Thus, the mechanism for calling a CA action must of necessity involve several steps. First, some agent is responsible for identifying the need to call a particular CA action and generating a token to represent a particular performance of that CA action. This involves some form of instantiation of a CA action template. Next, the agent is responsible for communicating the value of the token to the various threads that will perform the roles of the CA action. It is assumed that each thread knows (or is told) which role it is to perform. Finally, the various threads use the token in some kind of CA action call statement to indicate that they want to participate in a particular performance of the CA action. Each such call statement must name the particular role in the CA action to be performed by that thread and specify any arguments to be passed to that role.

In the general case there seems to be no alternative to this complicated mechanism, although it might be possible to reduce the number of steps by making some simplifying assumptions. For example, for some CA actions, it might be sufficient to name the action and not a particular instance of the action, allowing threads to fill the roles on a "first come, first served" basis. But this kind of non-determinism would only be appropriate for some applications and the need to name a particular instance of a call to a CA action seems unavoidable in general.

How does parameterisation fit into this model? There are two separate opportunities to parameterise a CA action: when a particular call to the action is instantiated, and when a thread asks to perform a particular role within that call. So it is reasonable to allow parameters to both the CA action as a whole and to each role within the CA action, with the values of each parameter being specified at the appropriate point during the instantiation and performance of a particular call to the CA action.

What is the relationship between the parameters to a CA action and the various objects that are accessed by the CA action, both internally and externally? Parameterisation seems to be a good way of naming the external objects that are accessed by a CA action. These objects can either be associated with the CA action as a whole or associated with a particular role within the CA action. Conversely, the objects that are used internally by the CA action are entirely local to a particular performance of a CA action. They are created at the start of the performance and destroyed at the end of the performance in the same way that the local variables used by a procedure call exist only for the duration of that procedure call. Thus, objects that are passed into CA actions as parameters remain external to the CA action and are subject to competitive concurrency whilst objects that are local to CA actions are not visible from outside the CA action and need only be subject to cooperative concurrency.

It could be argued that it might be useful to allow the state of local objects to persist between calls to a given CA action. But this poses a number of difficulties and is an unnecessary complication. For example, if the state of the local objects is a property of the CA action as a whole rather than a particular call to a CA action, what happens if two calls to the same CA action are active at the same time? Although it is tempting to view CA actions as objects rather than procedures, in other words entities that have a persistent state, it is not necessary to do so because any state associated with a particular instantiation of a CA action that needs to persist between successive calls to that action can be represented by an external object bound to the CA action as a parameter.

3 Design issues and problems to be solved

As we have indicated, the conceptual model for CA actions is intended to be neutral with respect to issues of language representation and implementation technology. But there are some obvious design issues that need to be solved in order to realise the concept of CA actions. In this section, we highlight some of these issues and explore some of the trade-offs that arise in their implementation. For example, a centralised implementation of CA actions is more straightforward than a distributed implementation but is susceptible to a single point of failure. Similarly, the use of optimistic concurrency control techniques may be more appropriate than pessimistic concurrency control techniques in certain situations. Our goal is to achieve a flexible implementation strategy for CA actions that supports the same basic conceptual model at all levels of abstraction but is capable of exploiting appropriate simplifying assumptions as appropriate. Thus we consider issues such as thread coordination, exception handling and resolution, coordinated access to external objects, and software fault tolerance and show how these can be implemented in different contexts and under different assumptions.

3.1 Thread coordination

In this section we concentrate mainly on the problems of synchronising the threads participating in a CA action. We consider synchronisation issues at the entry and exit points of a CA action and also during thread recovery. We also discuss the problem of thread desertion and the issue of global vs. local acceptance tests.

Particular implementations can provide different levels of synchrony. Threads can enter a CA action asynchronously but they have to be synchronised (at least logically) at the exit. Exit synchronisation involves agreeing on the success or failure of the action and then either committing the action or attempting to recover from any errors that are detected by the acceptance test. If the action is eventually able to complete successfully according to the acceptance test with either a normal or an exceptional outcome, then changes to external objects must be committed; otherwise, the effects of the CA action must be undone by aborting any changes made to external objects. Finally, the threads must leave the CA action in synchrony, possibly signalling an exception to the enclosing environment.

It is relatively easy to synchronise threads on exit from a CA action in centralised systems: this can be done by various existing concurrent programming mechanisms (monitors, semaphores, etc.) or by one of the logical synchronization schemes such as time-warping [Jefferson 1985]. Implementing synchronised exits is much more difficult in distributed decentralised systems. However, an underlying service such as causal order delivery or reliable group communication [Chang and Maximchuk 1984] can make this implementation much simpler.

With respect to exception handling, there are two distinct approaches: *blocking* and *pre-emptive*. In blocking schemes, each role terminates by reaching the end of the action or fails by raising an exception. However, the other roles involved in the action are informed of an exception only when they are completed (or, also detect an error) and so become ready to accept information about the state of other roles. In contrast, pre-emptive schemes do not wait but use some language feature to interrupt all roles when one of them has detected an error.

In blocking systems, error recovery and concurrent exception resolution are also much easier to provide than in pre-emptive ones because each role is ready for recovery and is in a consistent state when handlers are called. Moreover, there is no need to provide for the abortion of nested CA actions for these systems because such actions must have either completed successfully or have had any errors dealt with by the nested action's handlers [Campbell and Randell 1986] (Figure 4 illustrates this). Mechanisms such as timeouts and run-time error checks can increase the efficiency of blocking schemes and decrease the amount of time wasted by allowing early detection of either the error or the abnormal behaviour of the role that raised the exception and is waiting for the other roles. In pre-emptive schemes, there is inherently no wasted time but the feature required, namely pre-emptive thread interruption, is not readily available in many languages and systems. An appropriate approach should be chosen depending on the application, on the types of errors that are to be detected, on the failure assumptions, etc. But the general scheme should allow programmers to choose the most suitable approach.

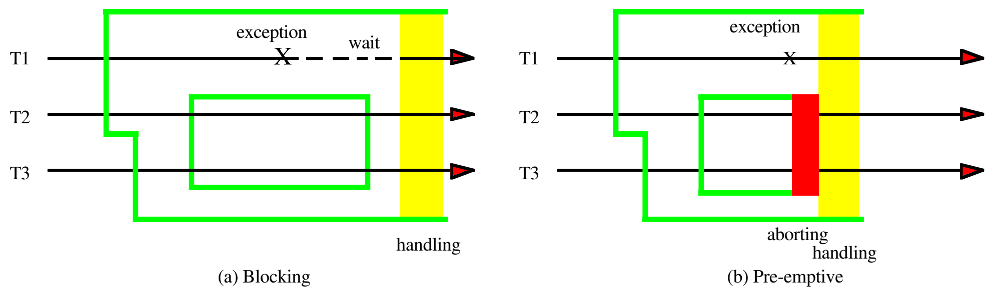

Figure 4 Two methods for treating nested actions when an exception is raised.

The deserter process problem [Kim 1982], which arises when a process that is intended to participate in a conversation fails to do so and causes the whole conversation to deadlock, exists in the CA action in a slightly modified form. It arises if one of the roles is either not activated (because no thread calls it) or never reaches the action exit. We term these two forms of thread desertion *entry* and *exit desertion* respectively. In order for these forms of thread desertion to be tolerated at the application level, it is necessary for the CA action mechanism to detect them and signal an appropriate exception. It is relatively simple for the CA action mechanism to detect exit desertion and then to abort the action pre-emptively. However, entry desertion cannot be dealt with within an action because the deserter thread has not entered the action and therefore cannot participate in the recovery. Entry desertion must be dealt with by the containing action. Thus, the action that detects the entry deserter should be aborted and an appropriate failure exception should be propagated up to the enclosing action with the hope of involving the deserter thread in the recovery at this level. Note that it is more difficult to deal with thread desertion in blocking schemes because they do not have features for interrupting the deserter thread.

The desired effect of performing a CA action is described by an acceptance test for the CA action as a whole, in other words, a global acceptance test in the sense of [Gregory and Knight 1985]. Many conversation schemes rule out global acceptance tests, preferring to use local acceptance tests for each participant in the conversation, but we believe that a global acceptance test is more appropriate for CA actions because it allows one to check: (i) the state of the CA action (when an action has its own state); (ii) the state of external objects (by calling side-effect-free methods in them); (iii) the states of the output parameters (when CA actions have them); and (iv) a complex joint condition on the states of the action, several roles, external objects and output parameters. However, in order to check a global acceptance test, it is necessary to synchronise all the threads participating in the CA action to ensure that the external objects are in a consistent state. A simple conjunction of local acceptance tests is easier to implement but not always adequate for CA actions.

3.2 Exception handling and resolution

During the execution of a CA action, it is possible for several of the participant threads to detect an error and raise an exception concurrently. Such errors could be the symptoms of a different, more serious fault, and thus need to be treated in a coordinated manner. Campbell and Randell proposed using an exception resolution mechanism to deal with this problem for conversations [Campbell and Randell 1986] and the same mechanism can be adapted for CA actions. An exception tree is used to impose a partial order on the exceptions that can be raised during a CA action and the handler for a higher level exception is also expected to be able to handle any lower level exception.

In an object-oriented system, if exceptions are typed, the exception tree could be deduced from the subtype hierarchy for exceptions. However, the tree must exist at run time so as to allow concurrent exceptions to be resolved. After such resolution, the same handlers are called in all participating roles, and forward or backward error recovery measures are employed in an attempt to recover from the error and complete the CA action successfully.

Exception handlers should be associated with each of the roles of the CA action so that when a thread enters the action, it enters the corresponding exception context. Some or all of these threads may further enter a nested CA action and such nesting of CA actions causes the nesting of exception contexts. It must thus be guaranteed that each role of the nested action is associated with an appropriate set of handlers. In practice, such association could be done either statically or dynamically. Once the association is provided, clear semantics of exception propagation can be easily enforced. Exceptions can be propagated along nested exception contexts, corresponding to the chain of nested CA actions. Here we simply adhere to the termination model — in any exceptional situation, the handlers take responsibility for coordinated error recovery and either complete the CA action successfully (possibly by signalling an exceptional outcome) or else attempt to abort the effects of the action, signalling either `abort` or `failure` depending on whether the undo succeeds or not. Figure 5 shows two possible forms of coordinated error recovery.

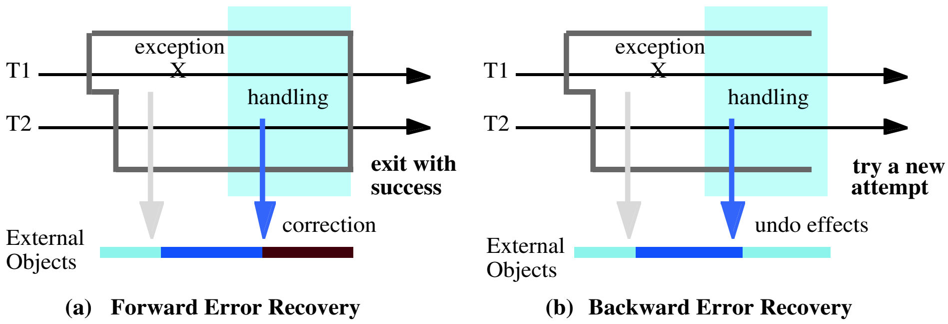

Figure 5 Different forms of coordinated error recovery.

There are a number of practical considerations regarding coping with concurrent exceptions within a CA action; detailed analysis and discussion are presented in [Romanovsky, Xu et al 1996], which describes a new algorithm that solves a number of difficulties with the resolution algorithm introduced in [Campbell and Randell 1986]. Figure 6 illustrates how the new algorithm works when two concurrent exceptions `E1` and `E2` are raised in a nested CA action that contains four participating threads. The formal details and an analysis of the algorithm can be found in [Romanovsky, Xu et al 1996].

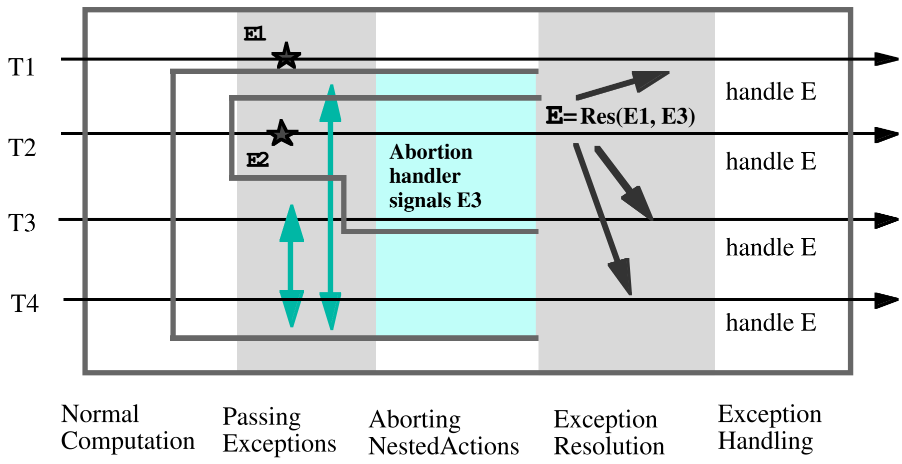

Figure 6 Concurrent exception handling and resolution.

3.3 Coordinated access to external objects

In this section, we will focus on how a CA action controls operations on external objects that may be accessed competitively by other actions. Figure 7 shows the structure of a possible CA action processing system implemented on top of an existing transaction system.

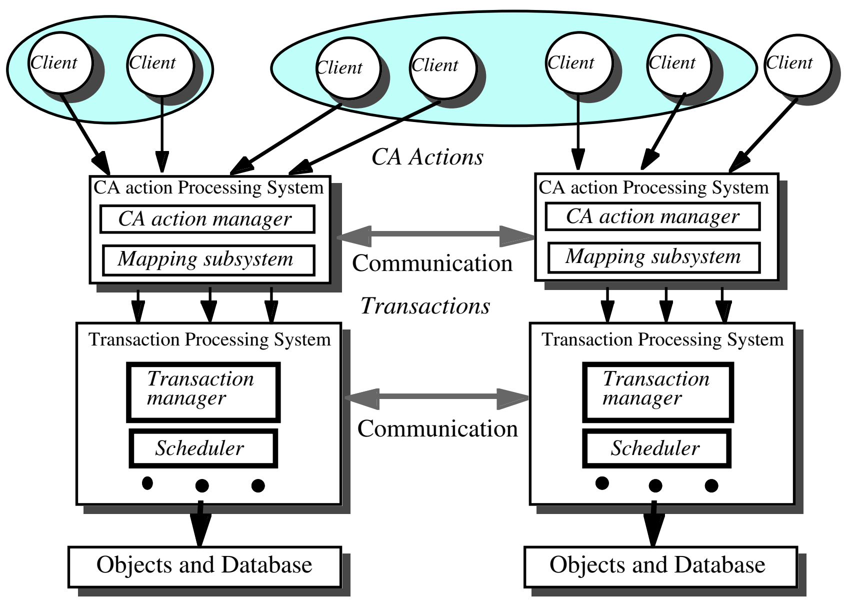

Figure 7 A distributed CA action processing system.

Given the widespread existence of transaction processing systems and mechanisms, adding a specially designed mapping mechanism onto an existing transaction mechanism in this way is more realistic from a practical point of view than designing a new mechanism for CA actions from scratch, even though the latter approach might allow a greater degree of concurrency.

It is important to note that the execution order of operations invoked in a CA action is determined by both the order specified by each participating thread and the communication (and synchronization) relationship between participating threads. These ordering relationships must be taken into account in determining the correct mapping from a set of nested CA actions onto a corresponding set of nested transactions.

Consider an example of nested CA actions, as shown in Figure 8. Two participating threads `T1` and `T2` are involved in two nested CA actions. `T1` contains two operations `op1` and `op2` on the external object *O* and `T2` contains two operations `op3` and `op4` on the same object. Since `T1` and `T2` may communicate each other, different interactions between them will lead to completely different nested transactions which permit different degrees of concurrency.

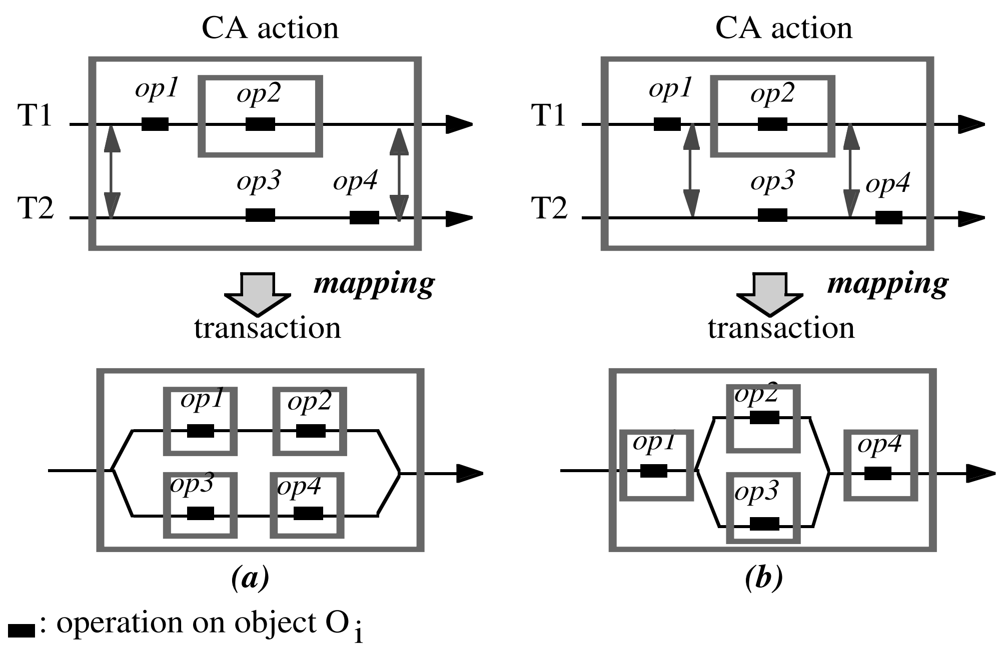

Figure 8 Mapping between nested CA actions and nested transactions.

The precise details of the way in which a given set of nested CA actions is mapped onto a set of nested transactions are still a subject of our research.

3.4 Design fault tolerance

Any provisions within a CA action for tolerating design faults, such as the use of design diversity, should not be visible from outside the action. For example, a CA action might make coordinated use of Recovery Blocks [Randell 1975]. As portrayed in Figure 9, this in effect involves the CA action's threads performing a sequence of one or more nested CA actions, depending on the errors encountered. Only if and when the CA action has completed (successfully or otherwise) will any effects of the final set of changes that it has made to external objects be made visible to other threads. The implementation, using the CA action mechanisms described earlier, of such an essentially sequential fault tolerance scheme should be straightforward.

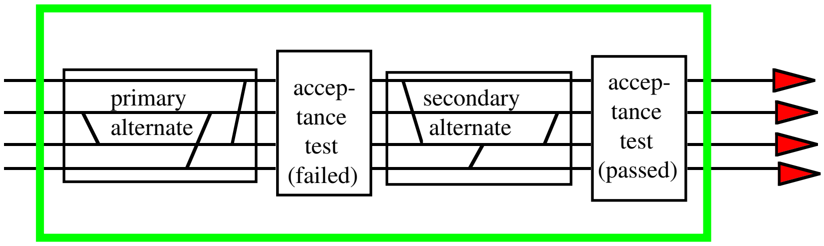

Figure 9 Design diversity: CA actions with Recovery Blocks.

Alternatively, coordinated use of N-version programming [Avizienis 1985] might be employed, as portrayed in Figure 10. The implementation of such a scheme is not quite so straightforward, because of the complications that arise through parallel accesses to objects that are external to the action (which may or may not themselves have been similarly replicated). These accesses will have to be synchronised and voted upon, as well as being subject to the normal transactional controls that are required for the use of objects that are external to a CA action.

The Recovery Block scheme and N-Version Programming are in fact just special cases of a more general scheme [Xu, Randell et al 1995b], whose components are a control algorithm, a set of diversely-designed variants, an adjudicator and an acceptance test. A CA action can employ such a scheme by using the control algorithm to govern the creation of any required parallel threads, the invocation of the variants, and the checking of the acceptance test. Additional mechanisms are required to create, synchronise, and merge the threads needed for parallel execution of variants, and control the synchronisation and voting on parallel accesses from these threads to external objects. Replica determinism is another problem that must be addressed. We are currently exploring the integration of such a scheme into an object-oriented linguistic framework.

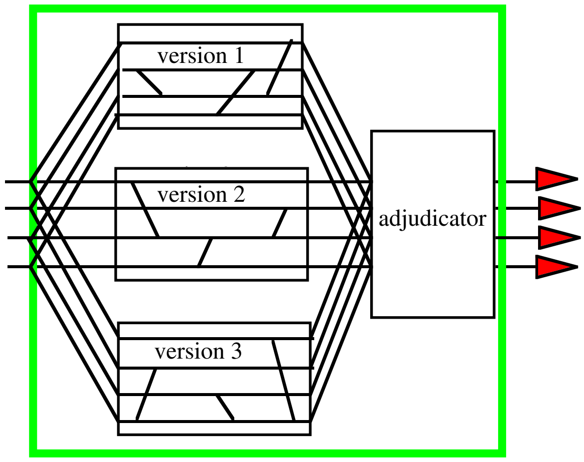

Figure 10 Design diversity: CA actions with N version programming.

4 Implementations and prototypes

In the two years since the original publication of the CA action proposal [Xu, Randell et al 1995a], we have explored various design and implementation issues for CA actions in a wide range of experiments. However, rather than devising a new programming construct for CA actions, we have realised CA actions using a combination of programming conventions backed up by sets of reusable components (e.g. action controller, resolution procedure, etc.). We have experimented with centralised (single-computer and distributed) CA action schemes in Ada and Java and have also implemented a decentralised (and distributed) manager for CA actions using Ada.

There are several reasons why we chose these two particular languages for our experiments. Ada95 [Ada 1995] is the only standard object-oriented language that has features for real-time, distributed programming. Moreover, it provides very elaborate support for exception handling and, in particular, for data- and process-oriented concurrent programming, which simplifies the programming of a CA action run-time manager. (We used the GNAT Ada95 compiler (public release 3.04) on SunOS 5.4 and took as our starting point the implementation of atomic actions described in [Wellings and Burns 1996]). Our other choice, Java [Java 1995], is an increasingly popular language from the C++ family that is an improvement on its predecessor in many respects: it has features for concurrent and distributed programming, better exception handling, clearer semantics, etc. Our implementations used JDK 1.0.2 (Sun Microsystems).

In total, we have produced nine different implementations of CA actions (five in Ada95 and four in Java): *Exp-A1* [Romanovsky, Randell et al 1997], *Exp-A2*, *Exp-A3*, *Exp-A4*, *Exp-J1*, *Exp-J2*, *Exp-J3*, *Exp-J4*, *Exp-A5*. Table 1 compares these implementations.

	Exp-A1	*Exp-A2*	*Exp-A3*	*Exp-A4*	*Exp-J1*	*Exp-J2*	*Exp-J3*	*Exp-J4*	*Exp-A5*
Centralised	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Exception Handling	B	P	B	B	B	B	P	P	P
Separated Role and Manager	No	No	Yes	No	No	Yes	No	Yes	N/A
Backward Error Recovery	No	No	No	Yes*	No	Yes	No	Yes	No
Multiple Exception Handling	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Nesting (conversation)	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
External Objects	No	No	No	No	Yes	Yes	Yes	Yes	Yes

* internal objects only

Table 1 Comparison of nine CA action implementations.

In all our implementations threads enter CA actions asynchronously but exit them synchronously. Some of our implementations use a blocking (B), some a pre-emptive (P) exception handling scheme. All implementations allow forward error recovery and propagate failure exceptions to the containing action if recovery is unsuccessful after an internal exception.

In our centralised implementations the action is controlled by a CA action manager (see [Xu, Randell et al, 1995a]) that provides some or all of the following services as appropriate: (i) *coordination service*: activating the roles, synchronising role exit and entry (when necessary); (ii) *exception service*: interrupting the roles if one of them raises an exception, resolving the exceptions, failure exception propagation; (iii) *information exchange service*: dealing with external objects, dealing with local objects and providing other mechanisms for inter-role cooperation; (iv) *fault tolerance service*: providing recovery points, execution of the acceptance test and control of the adjudicator and diverse set of roles (if design fault tolerance is employed). Generally speaking, the CA action manager controls references to external objects, role components, adjudicator, information about participants, and nested actions. Roles in these centralised implementations can be either glued together with the manager or be programmed separately. In the latter case, the roles and the manager can be presented as objects of different classes.

Section 3.1 explained why it is important to distinguish between implementations with blocking and pre-emptive exception handling. Both Ada95 and Java have features for thread interruption and we used them in some of our implementations.

As yet, we have not attempted to integrate our implementations of CA actions with a transaction system nor have we implemented full-scale transactional support for external objects. However, we have developed a simple implementation of recoverable objects in Java which supports flat concurrency control. This is why our prototype implementations only support nested actions that have conversation-like semantics (i.e. no access to external objects).

While implementing these various schemes we realised that there are many ways of packaging CA actions as programming abstractions. For example, CA actions could be implemented as packages, abstract data types, sets of procedures, classes, etc. Depending on the method chosen, different forms of distribution can be used and this affects how actions are designed, reused, and extended. However, in each case the aim is to achieve a good separation of concerns between application programmers who use CA actions, application programmers who design CA actions, and system programmers who are responsible for the implementation of the CA action mechanism itself [Xu, Randell et al 1996].

CA actions are packaged as classes in some of the centralised implementations (*Exp-A1*, *Exp-A2*, *Exp-A4*, *Exp-J1*, *Exp-J3*); each action class includes the action manager and all the roles are presented as methods. This makes it possible to use several different means of introducing diversity so as to tolerate design faults: class diversity can be used for the action design in all these implementations and for the role design in *Exp-J1* and *Exp-J3*; method diversity can be applied to the design of roles in all these implementations. The managers are essentially the reusable components. CA action classes can be extended and reused by adding new action roles or by overriding the old ones.

Distribution of roles is much more difficult to provide for the implementations in which CA actions are classes that incorporate both manager and role-methods because neither of the chosen programming languages support method distribution. That is why in *Exp-A3* and *Exp-A5* roles are not class methods but separate procedures that are included into separate distribution units. Another (much more object-oriented) approach is taken in the second Java-based scheme in which roles are objects and hence can be easily distributed. Unfortunately, both of these distributed CA action schemes make roles visible to the outside world, thus increasing the chances for information smuggling [Kim 1982] and for role misuse. We believe that some additional care should be taken to guarantee that roles can be called only within their CA action. For example, calling a role in *Exp-J2* and *Exp-J4* requires the use of an additional parameter as a sort of action identifier. An alternative approach would to call each role indirectly via a proxy object that enforced an access control policy.

Comparing *Exp-J1*, *Exp-J3* with *Exp-J2*, *Exp-J4* we conclude that it is simpler to provide software diversity with separated manager and role components (because their specifications can be given to different application programmers). Moreover, it is possible to distribute both roles and manager even though neither Java nor Ada95 support distribution at the method level — such distribution facilitates the implementation of provisions for tolerating hardware faults.

In all our implementations we rely on the sequential exception handling provided by the host languages. In [Romanovsky 1996] we formulate a general approach for dealing with concurrent exception resolution in languages such as Java and Ada95 which have both concurrency and local exception handling. This was used in *Exp-A1*, *Exp-A4*, *Exp-J1*, *Exp-J2*, *Exp-J3*, *Exp-J4*. Both Ada95 and Java support exception propagation via remote procedure calls; this is why exception resolution in all our centralised distributed CA action schemes is not very different from resolution in single-computer systems.

Further details of some of our experiments can be found in [Romanovsky 1996][Romanovsky, Randell et al 1997]; a paper detailing the set of experiments as a whole is now in preparation.

5 Conclusions and further work

Some two years on from the original publication of the CA action concept, we feel that we have a much fuller understanding of CA actions and the design issues involved in their implementation (as we hope has been evident from this paper). However, much remains to be done and we therefore plan to continue our research into CA actions as a major activity in the ESPRIT Long Term Research Project on “Design for Validation” (DeVa) together with a number of our partners on that project.

Clearly, it is necessary to develop a more detailed linguistic formulation for CA actions even though, for practical reasons, our implementation experiments are likely to remain based on the use of existing languages buttressed by library routines and programming conventions. (We also hope to make use of recent work in DeVa on object-oriented reflection [Stroud and Wu 1995][Fabre and Perennou 1996], so as to achieve a greater logical separation of code involved in various aspects of the CA mechanism from application code.)

A proper formal model of CA actions is also needed — two promising starting points for such a model are the work of our colleagues at EPFL on CO-OPN2, a specification language for concurrent object-oriented systems [Biberstein, Buchs et al 1997], and work here at Newcastle on Petri Box Algebra [Koutny and Best 1997].

Last but not least, we would also like to extend the CA action concept so as to make it suitable for use in real-time systems. It is possible to draw an interesting parallel between the use of the present CA action scheme for structuring the behaviour of a system in the value domain, and the use of temporal firewalls [Kopetz 1996] for structuring system behaviour in the time domain. We therefore intend to investigate the ways in which these two concepts might complement each other in cooperation with our colleagues at the Technical University of Vienna and the University of York.

6 Acknowledgements

This work was supported by the ESPRIT Basic Research Project 6362 on "Predictably Dependable Computing Systems", and by the ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa), and has benefited greatly from discussions with a number of colleagues within these projects, in particular, Jean-Charles Fabre and Tanguy Pérennou of LAAS-CNRS, and Alan Burns, Stuart Mitchell, and Andy Wellings of the University of York.

References

[Ada 1995] "Ada. Language and Standard libraries" ISO/IEC 8652:1995(E), Technical Report, Intermetrics Inc., 1995.

[Avizienis 1985] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Soft. Eng.*, vol.SE-11, no.12, pp.1491-1501, 1985.

[Banatre, Banatre et al 1986] J.P. Banatre, M. Banatre and F. Ployette, "The Concept of Multi-Functions: A General Structuring Tool for Distributed Operating

Systems," in *Proc. 6th Distributed Computing Systems Conf.*, pp.478-485, 1986.

[Biberstein, Buchs et al 1997] O. Biberstein, D. Buchs and N. Guelfi, "CO-OPN2: A Specification Language for Distributed Systems Engineering," in *DeVa 1st Year Report*, York, 1977.

[Burns and Wellings 1995] A. Burns and A. Wellings. *Concurrency in Ada*, Cambridge University Press, 1995.

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

[Chang and Maximchuk 1984] J. Chang and N.F. Maximchuk, "Reliable Broadcast Protocols," *ACM Trans. Comput. Syst.*, vol.2, no.3, pp.251-273, 1984.

[Cristian 1989] F. Cristian, "Exception Handling," in *Dependability of Resilient Computers* (ed. T. Anderson), pp.68-97, Blackwell Scientific Publications, 1989.

[Davies 1973] C.T. Davies, "Recovery Semantics for a DB/DC system," in *Proc. ACM Nat. Conf*, pp.136-141, 1973.

[Davies 1978] C.T. Davies, "Data Processing Spheres of Control," *IBM System Journal*, vol.17, no.2, pp.179-198, 1978.

[Ellis, Gibbs et al 1991] C.A. Ellis, S.J. Gibbs and G.L. Rein, "Groupware, Some Issues and Experience," *Communications of the ACM*, vol.34, no.1, pp. 38-58, 1991.

[Elmagarmid 1993] A.K. Elmagarmid (ed.). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publ., 1993.

[Fabre and Perennou 1996] J.C. Fabre and T. Perennou, "FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications," in *Proc. 2nd European Dependable Computing Conference* (EDCC2), Taormina, 1996.

[Francez and Forman 1996] N. Francez and I.R. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*, Addison-Wesley, 1996.

[Gray 1978] J.N. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer *et al* eds., pp.393-481, Springer-Verlag, 1978.

[Gray and Reuter 1993] J.N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publ., 1993.

[Gregory and Knight 1985] S.T. Gregory and J.C. Knight, "A New Linguistic Approach to Backward Error Recovery," in *Proc. 15th Int. Symp. on Fault-Tolerant Computing*, 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 *Proc. 19th Int. Symp. on Fault-Tolerant Computing*, pp.506-511, Chicago, 1989.

[Issarny 1993] 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, 1993.

[Java 1995] "The Java Language Specification. Version 1.0 Beta," Sun Microsystems Inc., 1995.

[Jefferson 1985] D.R. Jefferson, "Virtual Time," *ACM Trans. Prog. Languages*, vol.7, no.3, pp.404-425, 1985.

[Jung and Smolka 1996] Y-J. Joung and S. A. Smolka, "A Comprehensive Study of the Complexity of Multiparty Interaction," *Journal of the ACM*, vol.43, no.1, pp.75-115, 1996.

[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.

[Kopetz 1996] H. Kopetz, "Temporal Firewalls," presented at *DeVa 1st Selective Open Workshop*, Schloss Reisensburg, Sept. 1996.

[Koutny and Best 1997] M. Koutny and E. Best, "Operational and Denotational Semantics for the Box Algebra," to appear in *Theoretical Computer Science*, 70 pages, 1997.

[Liskov 1988] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no.3, pp.300-312, 1988.

[Lyu 1995] M. Lyu (ed.).*Software Fault Tolerance*, (Trends in Software series), John Wiley & Sons, 1995.

[Moss 1981] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," *Ph.D. Thesis (Tech. Report 260)*, MIT Lab. for Computer Science, Cambridge, MA., 1981.

[Parrington, Shrivastava et al 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems Journal*, vol.8, no.3, 1995.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Soft. Eng.*, vol.SE-1, no.2, pp.220-232, 1975.

[Randell and Xu 1995] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," *Software Fault Tolerance*, M. Lyu Ed. (Trends in Software series) pp.1-22, John Wiley & Sons, 1995.

[Romanovsky, Xu et al 1996] A. Romanovsky, J. Xu and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems," in *Proc. 16th IEEE Intl. Conf. Distributed Computing Systems*, pp.545-552, Hong Kong, 1996.

[Romanovsky 1996] A. Romanovsky, "Atomic Actions Based on Distributed/Concurrent Exception Resolution," Technical Report, Computing Dept., University of Newcastle upon Tyne., no.560, 1996.

[Romanovsky, Randell et al 1997] A. Romanovsky, B. Randell, R. Stroud, J. Xu and A. Zorzo, "Implementing Synchronous Coordinated Atomic Actions Based on Forward Error Recovery," *Journal of Systems Architecture*, vol.2, July 1997.

[Shrivastava and Wheater 1990] S.K. Shrivastava and S.M. Wheater, "Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions," in *Proc. 10th Intl. Conf. Distributed Computing Systems*, pp.203-210, Paris, 1991.

[Stroud and Wu 1995] R.J. Stroud and Z. Wu, "Using Metaobject Protocols to Implement Atomic Data Types," in *Proc. ECOOP'95*, LCNS 952, pp. 165-189, 1995.

[Wellings and Burns 1996] A.J. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95", Technical Report, Department of Computer Science, University of York, no.YCS-263, 1996.

[Xu, Randell et al 1995a] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud and Z. Wu, "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," In *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, pp.499-508, Pasadena, June 1995.

[Xu, Randell et al. 1995b] 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. Pradhan and D. Avreski (Eds.), pp.226-233, IEEE CS Press, 1995.

[Xu, Randell et al 1996] J. Xu, B. Randell and A.F. Zorzo, "Implementing Software-Fault Tolerance in C++ and Open C++," in *Proc. 1996 Intl. Workshop Computer-Aided Design, Test, and Evaluation for Dependability (CADTED96)*, Beijing, pp.224-229, July 1996.