A Comparative Study of Fault-Tolerant Concurrent Mechanisms: Atomic Transactions, Conversations and Coordinated Atomic Actions

Delano M. Beder   *Cecília M.F. Rubira*

Relatório Técnico IC–99-21

Setembro de 1999

A Comparative Study of Fault-Tolerant Concurrent Mechanisms: Atomic Transactions, Conversations and Coordinated Atomic Actions

Delano M. Beder Cecília M.F. Rubira

Abstract

Distributed computing often gives rise to complex concurrent and interacting activities. In many cases, several concurrent activities may be working together, i.e., *cooperating*, to solve a given problem; in other cases the activities may be independent but sharing common system resources for which they should *compete*. In practice, different kinds of concurrency might coexist in a complex application which thus will require a general supporting mechanism for controlling and coordinating complex concurrent activities. Many difficulties and limitations occur when the object and action model is used to support cooperating activities. This paper discuss some of these difficulties and limitations and then presents an alternative model for constructing fault-tolerant distributed systems based on the concept of Coordinated Atomic (CA) actions [20] which provides uniform support for both cooperative and competitive concurrency.

1 Introduction

The *object and action* model is used to provide fault tolerance object-oriented systems [17]. Atomic transactions with the properties of serializability, failure atomicity and permanence of effect are used to control operations on shared objects. The model attempts to hide the effects of failures and concurrent processing from the application programmer, and is very effective for certain applications in which concurrent activities are relatively independent, only needing to share some common objects for which they have to *compete*. Atomic transactions [6] are used to tolerate (hardware) faults in competitive concurrent systems. Practical examples include some types of electronic funds transfer and airline reservation systems. However, limitations and difficulties with the *object and action* model are now becoming more evident as soon as it is used more widely. One of the most severe limitations is that the model does not provide appropriate support for cooperation between concurrent activities.

In many real-world applications, there are specific needs for cooperation and coordination among activities. For example, several concurrent activities may need to work together, i.e. *cooperate*, to solve a given problem. Since such a cooperative concurrency requires explicit interactions among activities, some way (in principle more sophisticated than traditional transaction concept) of enclosing interactions is required in order to control the overall system complexity and hence to facilitate error recovery. The best-known approach for structuring (cooperatively) concurrent systems to facilitate error recovery and fault tolerance is the conversation concept, an approach that provides full support for cooperative concurrent activities. Conversations (sometimes called atomic actions) [11] were proposed as a means of allowing designers to structure cooperative concurrent systems and to incorporate software fault tolerance in a disciplined way. Concurrent processes (threads) enter in a conversation and cooperate within its scope in such a way that no information flow is allowed to cross the conversation border. This obviously restricts system design but it makes possible to regard each conversation as a recovery region (beyond which erroneous information cannot be spread) and to attach fault tolerance features (application-dependent or provided by conversation support) to each individual conversation. Basically, these features provide error detection and recovery within conversations: when an error has been detected, the corresponding recovery starts.

Conversations can use backward error recovery, forward error recovery, or a combination of these [3, 8]. In any case, recovery has to be coordinated, and all conversation participants have to be involved in it. Backward error recovery does not depend much on the application and can be implemented in a way transparent (or provides, to a considerable degree, by the conversation support) because it uses the rollback of all conversation participants to recover the system. Forward recovery usually relies on an exception mechanism and may incorporate an additional mechanism to resolve multiple exceptions raised in several conversation participants [3] (see section 3.2.1 for a more detailed discussion). This recovery is application-dependent by nature and this is why only basic support and a general structuring mechanism are provided by conversations. Conversations can be nested; in this case, the execution of the nested conversation is indivisible and invisible for the containing and for the sibling conversations, and the nested conversations results cannot be seen (are not committed) until the containing conversation is completed. Unfortunately, conversations are mainly directed to process-oriented systems such as process control or real-time control applications, and concentrate on the issues of factoring and organizing the interactions between process without addressing the problem of consistency of shared objects accessed by these process.

The concept of *coordinated atomic action* [12, 20] (or CA actions) is the first attempt to integrate transactions and conversations into a single conceptual framework for coping with different kinds of concurrency and achieving fault tolerance. The CA action concept thus encompasses strategies for dealing with hardware and software faults to provide coordinated error recovery between a set of interacting objects.

This paper discuss the problems and limitations of the *object and action* model and then presents an alternative *object and CA action* model [21] which supports both cooperative and competitive concurrency. In section 3 the *object and CA action* model is introduced, an application example is described and some implementations of the coordinated action concept are discussed. In section 4, generalized transaction models are summarized, and problems and difficulties with these models are discussed. And finally, in section 5 our conclusions are presented.

2 Transactions and Conversations

In order to ensure consistency of shared objects, atomic transactions [2] control operations on such objects with the properties of serializability, failure atomicity and permanence of effect. The serializability property ensures that concurrent executions of programs are free from interference (i.e. a concurrent execution must be shown to be equivalent to some serial order of execution). The second property ensures that a computation can either be terminated normally (committed), producing the intended results, or be aborted, producing no results. The permanence of effect property ensures that state changes produced are correctly recorded on stable storage. A **flat** (i.e. non-nested) **transaction** corresponds to a single execution thread composed of a partially ordered set of operations on objects and permits no concurrency inside the action.

The **nested transaction** model [9] extends the traditional transaction paradigm by providing the independent failure property for sub-transactions and supports modular construction of applications. For error recovery purposes, sub-transactions are in principle able to abort independently from their parent. Moreover, by combining nesting with support for concurrent sub-transactions, each of which corresponds to one sub-thread of the execution, competitive concurrency at same level of nesting becomes possible. That is, two concurrent sub-transactions are activities independents that share common system resources for which they have to compete.

In the flat and nested transaction models, programming in a concurrent environment is performed similar to that in a sequential environment. The application programmer does not need to worry about possible interference between concurrent and competitive actions – the system will provide transparent mechanisms for concurrency control that ensure interference-free access to shared objects despite concurrent invocations. However, this concurrency transparency imposes a serious limitation on the applicability of the transaction models: no appropriate support for cooperative activities. Cooperative concurrency is application-specific in nature; any support for this kind of concurrency will generally require explicit cooperation and coordination in application programs. Because the traditional models do not address cooperation among concurrent actions, it is particularly difficult to organize a cooperative computation performed jointly by a group of concurrent threads; each thread is responsible only for one part of the entire computation.

The conversation [11] scheme (an extension to the recovery block scheme) provides a structured design for reliable concurrent programs. The boundary of a conversation consists of a recovery line, a test line, and two side walls. The boundary encloses the set of communicating (interacting) processes which are part of the conversation. The recovery line is the part of the boundary which defines the start of the conversation. It consists of a coordinated set of states (recovery point) for the interacting processes. At the start of a conversation, the state of each entry process is stored for use during recovery. The entry to a conversation need not be a synchronous event. The test line is a coordinated set of acceptance tests for the set of interacting objects. Each process is required to pass an acceptance test. A conversation is sucessful only if all processes pass their acceptance tests. If any acceptance test is failed, recovery is achieved by rolling back the conversation to the recovery line, restoring the process state to that on entry to the conversation, and executing the alternate blocks. Thus, processes in the conversation cooperate in error detection. The side walls of the conversation prohibit the passing of information to process not involved in the conversation (prevent information smuggling). A conversation consisting of three process is shown in figure 1.

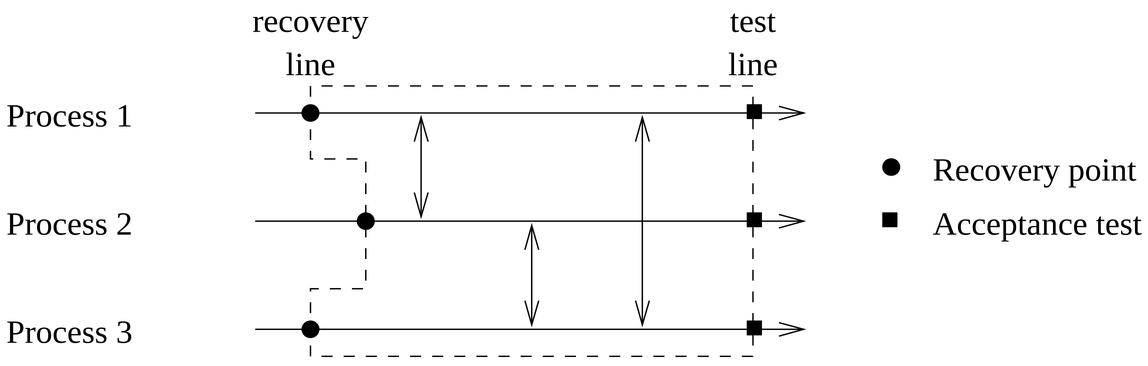

Figure 1: Conversation

3 Coordinated atomic actions

The coordinated atomic action concept was introduced [20] as a unified approach for structuring complex concurrent activities and supporting error recovery between multiple interaction objects in a distributed object-oriented system. This paradigm provides a conceptual framework for dealing with both kinds of concurrency (cooperative and competitive) and achieving fault tolerance by extending and integrating two complementary concepts – conversations and transactions. CA actions have properties of both conversations and transactions. Conversations (enhanced with concurrent exception handling) [3] are used to control cooperative concurrency and implement coordinated error recovery while transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

In this section we introduce the basic concepts behind CA actions and present a conceptual model for CA actions. We describe our view of concurrency in a object-oriented environment and explain how CA actions can be used as a mechanism for supporting both cooperative and competitive concurrency while tolerating both hardware and software faults. We also discuss issues such as nesting of CA actions.

3.1 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. However, we feel that it is useful to maintain a distinction, at least conceptually, between threads as the agents of computations and objects as the subject of computation. In an object-oriented system, computation proceeds by invoking methods (i.e. operations) on objects. In practice, the distinction between threads and objects can be rather blurred. For example, in some programming models, threads are just objects with a **run** method that is invoked asynchronously.

In a concurrent system, there are multiple threads of computation, in other words, multiple threads of method invocation. The existence of multiple threads introduces the possibility of two or more method invocations being active within the same object at the same time. The result would be chaos without adequate synchronization mechanisms.

Two levels of synchronization are required. Firstly, objects should 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. But this is not enough if threads need to perform a set of operations on a set of objects with no interference from other threads. This requires a guarantee of non-interference such as serializability. CA actions are intended to be a mechanism for providing this service. 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.

3.1.1 Kinds of Concurrency

For a given system, there may be a number of computations running concurrently, with each computation implemented as a set of threads. In order to perform effective concurrency control, it is helpful to identify different forms of concurrency. In [7], a set of concurrent processes is classified as being one of three categories, namely independent, competing or cooperating.

Concurrent threads are said to be *independent* if the sets of objects accessed by each thread are disjoint - a trivial case. *Competitive* concurrency arises when concurrent threads that are designed independently for different computations have access to a set of common objects but their access is not specially ordered (i.e., they have to compete for the shared objects). *Cooperative* concurrency arises when several threads are designed collectively and invoked concurrently to perform a pre-defined computation. Access to a set of common objects from these cooperative threads is specially ordered according to application-specific requirements. In reality, different kinds of concurrency may co-exist in a complex application and will require a general supporting mechanism.

3.2 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 participants 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. In order to perform a CA action, a group of threads must come together and agree to perform each activity in the CA action concurrently.

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, 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 should behave like a transaction with respect to these external objects. Thus, the effects of any operation that a thread within a CA action perform on external objects are not visible to other threads or CA actions until that CA action terminates. In other words, The CA action's execution looks like an atomic transaction from the outside world. One of the ways to implement this is to assume that there is a separate transactional support that provides these properties. A number of such transactional schemes are discussed in [2]. They offer the traditional interface, i.e., operations **start**, **abort** and **commit** transaction, which are called (either by the CA action support or by the CA action participants) at the appropriates points during the CA action execution.

As discussed in Section 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 hardware and software faults that combines mechanisms for forward and backward error recovery. In order to support backward error recovery, a CA action should provide a recovery line which coordinates the recovery points of the objects and the execution threads participants in the action so as to avoid the *domino effect* [11]. To support forward error recovery, a CA action should provide an effective means of coordinating the use of exception handlers.

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 (degraded) outcomes [5]. 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 signaling a specified exception to the surrounding environment. If it is not possible to satisfy the acceptance test at the end of a performance, even by signaling 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 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 signaled to the surrounding environment indicating that the CA action has failed to pass its acceptance test and its effects have not been undone.

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.

3.2.1 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 permanents. Software faults can be addressed using fault masking and design diversity.

During the execution of a CA action, each thread that is involved in the CA action may raise an exception. If a raised exception can not be dealt with locally by the thread, it should 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 is necessary in order to agree on the exception to be propagated and handled within the CA action. Campbell and Randell [3] have proposed using an exception resolution mechanism to deal with this problem applied for conversations (process-oriented concurrent systems). In order to deal with concurrent exceptions within a CA action, the general framework proposed [3] can be followed with some adjustments for distributed object-oriented systems. The work of Romanovsky *et al.* [15] discusses a number of difficulties and issues involved in the algorithm introduced in [3] and presents a new distributed object-oriented algorithm. This proposed algorithm is easier to implement and of lower complexity than the original solution. 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 fashion, an exception tree could be specified as a hierarchy of exceptions where exceptions are classes [15]. In an object-oriented system, if exceptions are typed, the exception tree could be deduced from the subtype hierarchy for exceptions.

Once an agreed exception has been propagated to all of the threads involved in the CA action, then some form of error recovery mechanism should 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 participant in order to tolerate design faults. 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 a CA action is performing on objects, both internal and external to the CA action. For example, backwards error recovery must involve the restoration of the state of all the objects involved in the CA action, including external objects, while forward error recovery must ensure that all of the objects are left in an acceptable state. Otherwise, a failure exception will be signaled to the external environment.

Figure 2 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 H1 and H2 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 sucessful outcome. As an alternative to performing forward error recovery, the two participants threads could undo the effects of operations on external objects, roll back and then try again, possible using diversely designed software alternates.

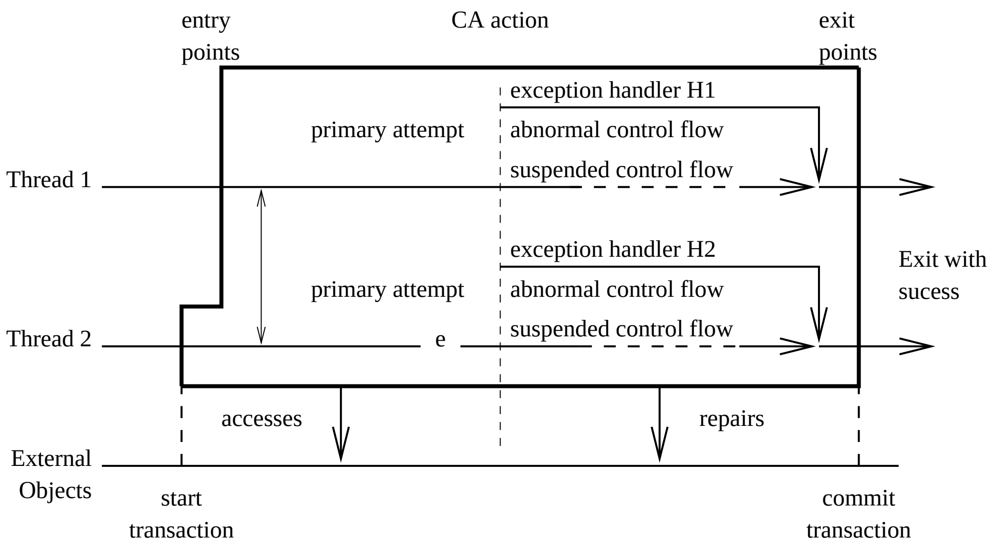

Figure 2: Coordinated error recovery performed by a CA action

3.2.2 Nesting

The CA action concept is intended to be recursive. In other words, nested CA actions can be used to structure the execution 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.

Adding support for nested CA actions to the basic model introduces a number of difficulties, as well as many benefits. 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 [9] with any objects it accesses externally, even if such objects are not external to the enclosing CA action (see figure 3).

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

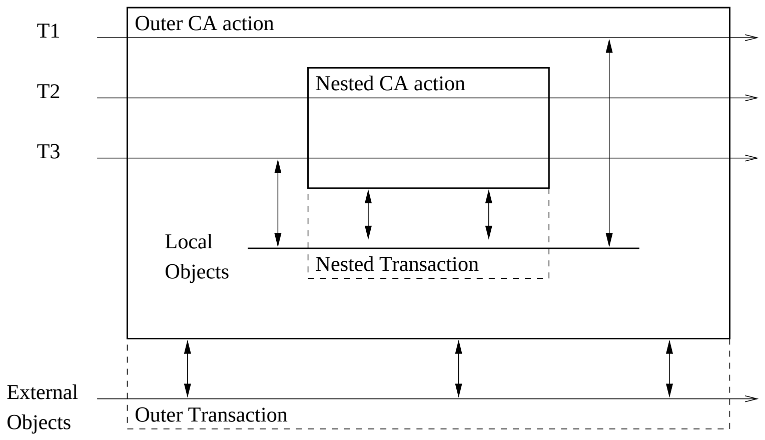

Figure 3: Nesting of CA actions

of interference and guarantee serializability. Meanwhile, the threads within the enclosing CA actions 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.

3.2.3 Thread coordination

Particular implementations can provide different levels of synchrony. Threads can enter a CA action asynchronously but they have to be synchronized (at least logically) at the exit. Exit synchronization involves agreeing on the success or failure of the action and then 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 a normal 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 signaling an exception to the enclosing environment. It is relatively easy to synchronize threads on exit from a CA action in centralized systems: this can be done by various existing concurrent programming mechanisms (monitors, semaphores, etc.). Implementing synchronized exits is much more difficult in distributed systems. However, an underlying service such as causal order delivery or reliable group communication can make this implementation much simpler.

With respect to exception handling, there are two distinct approaches: blocking and pre-emptive (see figure 4). In blocking schemes, each participant terminates by reaching the end of the action or fails by raising an exception. However, the other participants involved in the action are informed of an exception only when they are completed (or, also detect an error). In contrast, pre-emptive schemes do not wait but use some language features to interrupt all participants 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 participant is ready for recovery

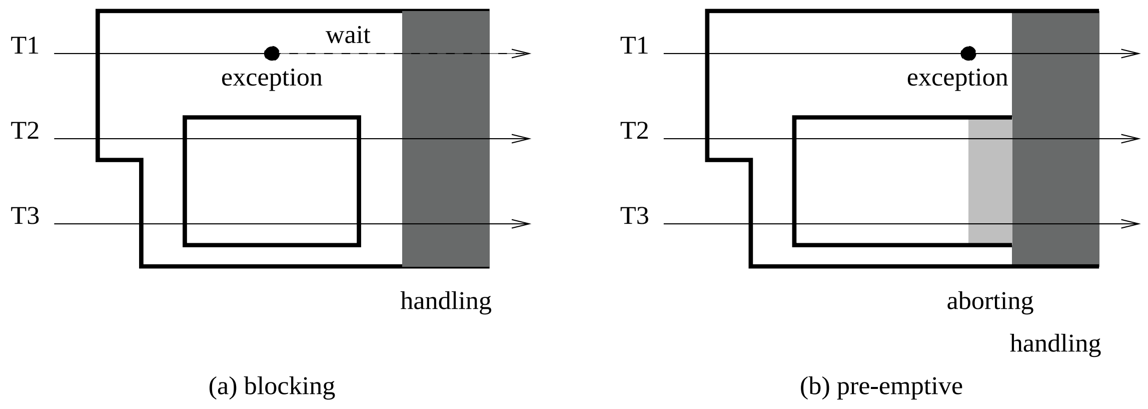

Figure 4: Blocking and pre-emptives schemes

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 [3]. Mechanisms such as timeouts 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 behavior of the participant that raised the exception and is waiting for the other participants. In pre-emptives schemes, there is inherently no wasted time but the feature required for the asynchronous transfer of control, namely pre-emptive thread interruption, is not readily available in many languages and systems. Even when they are available, they are usually very expensive [14]. Moreover, they usually have complex semantics. Then, it is more difficult to analyze, to understand and to prove programs which use these features. Another limitation with pre-emptives schemes is that the abortion of nested actions is difficult to program. In particular, even if application programmers implement such abortion handlers, a very sophisticated protocol (e.g. the one in [15]) needs to be applied to raise abortion exceptions in all nested actions (recursively and in the proper order); and, obviously, there are no languages which allow this. An appropriate approach should be chosen depending on the application, on the types of errors that have to be detected, on the failure assumptions, etc. But the general scheme should allow programmers to choose the most suitable approach.

3.3 Description of an application example

We present in this section an banking application that can be used structured using the coordinated atomic action concept.

Consider a distributed funds transfer system that consists of interacting objects and a transaction process system that ensures consistent accesses to the objects. The most typical objects in such a system are **bank account** objects. An account object has an internal state that records the current balance and an associated set of operations, such as **credit**, **debit**, **read-balance**, **check-balance**, etc. These operations have ordinary meanings: e.g. the check-balance operation 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**.

Different concurrent activities can occur in this banking application. Suppose **Mary** and **John** have two shared accounts, **acc-A** and **acc-B**. Because the system is distributed, it is possible for **Mary** and **John** to submit actions to the system at the same time, perhaps from different nodes. For example, when **Mary** issues the transaction to check the sum of money, **John** may be submitting a transaction to credit the two accounts with more money. Such concurrency is competitive in nature.

However, for a joint account a variety of authority and operation conditions could be defined by clients and must be verified when an action is actually issued. For example, if one of them, say **John**, wishes to withdraw US$ 100 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. There is a third person who examines the clients' PIN (Personal Identification Number) and then authorize (or not) the withdraw. 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**. After that, John withdraw the value required within a nested CA action. This withdraw is done within a nested CA action, because exceptions can be raised in course of this operation. Then, only the computation of this nested CA action is lost and the clients authorization process need not be made again. Before leaving the action, they communicate again to make sure that only the agreement amount of money is withdrawn.

From the viewpoint of the rest of the system, and in particular other users, the effects of the DEBIT CA action (figure 5) 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 can not be readily achieved by the use of conventional transaction-based systems.

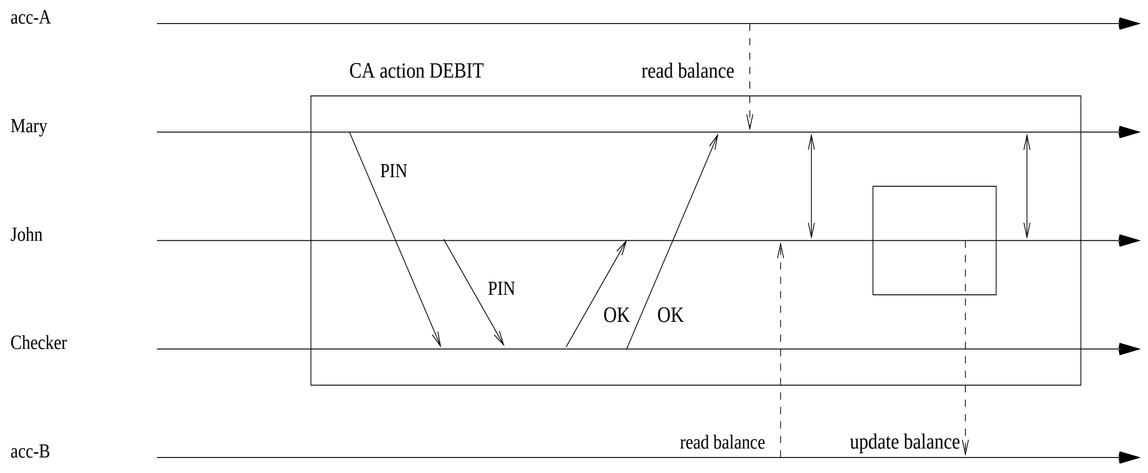

Figure 5: A cooperative action DEBIT

3.4 Implementations

Since the original publication of the CA action proposal [20], several implementations of CA actions have been produced. Further details of some of these implementations can be found in [13, 14]. In all their implementations threads enter CA actions asynchronously but exit them synchronously. Some of these implementations use a blocking, some a pre-emptive 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 all their implementations they rely on the sequential exception handling provided by the host languages. In [13] they 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.

Based on these experiments, Zorzo *et al.* [22] have developed a framework for implementing CA actions. Each CA action is represented by a set of components: one manager object, a set of participant objects, a set of local objects, and a set of external objects. The CA action mechanism is responsible for managing synchronous action entry and exit, global exception handling, recovery, consistency and atomicity of external and local objects, and so on.

To implement the above framework they provide the programmer with four different classes which can be used to program CA actions in Java: **CAActionManager**, **CAActionRole**, **ExternalObject**, and **SharedLocalObject**

CAActionManager. The **CAActionManager** class contains the methods to deal with entry and exit synchronization, action exception handling, controlling access to external objects, and accessing participants that compose the action. Thus, the **CAActionManager** provides a basic framework for coordinating the participants in a CA action and deals with the application-independent aspects of error handling. Extensions of this class are responsible for specifying the application-dependent aspects of error handling (notably, the exception resolution procedure).

CAActionRole. The extensions to the **CAActionRole** class contain the main code for one of the participants that compose the CA action. Only objects whose type is derived from **CAActionRole** can participate in a CA action. When deriving a new class from the CAActionRole class, the programmer has to implement at least one method: the private **execute ()** method that will contain the main code of that participant.

ExternalObject. Every new class extended from **ExternalObject** class is provided with transactional semantics, i.e. **Begin**, **Commit**, and **Abort** methods.

SharedLocalObject. Shared local objects are the objects used by the participants in order to exchange information with each other.

We think this framework presents some drawbacks. First, since only the **execute** method will be executed by threads, the same object can not take part in different CA actions simultaneously. Second, they didn't take into consideration design diversity. Third, the **CAActionManager** class provides a very simple exception handling mechanism. Programmers extending this class are responsible for specifying the application-dependent aspects of error handling (notably, the exception resolution procedure).

A reflective framework for implementing CA actions has been designed. A document describing this reflective framework has recently been done [1]. This framework separates objects into well-defined levels and the same object can take part in different CA actions simultaneously. At base level are the objects of the application, as his developer has implemented them. At meta-level are the objects that implement non-functional services such as distribution, security and fault tolerance. These services can be combined in such a way that a base-level object can be directly associated with one meta-object that encapsulates several non-functional services. The behavior of the base-level classes on this framework and the classes on Zorzo's framework are analogous but each object is associated with a meta-object. These meta-objects implement several services such thread synchronization, exception resolution, design diversity and so on.

4 Transaction-based approaches for cooperative concurrency

The *object and action* model [17] is widely used to provide fault tolerance in distributed-oriented systems. Several systems have been developed that successfully combine transaction processing with the object paradigm, e.g. Arjuna [16]. 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. In this section we will examine the generalized transaction models that attempt to support a certain degree of cooperative activities.

Many variations of the basic transaction model have been proposed. These proposals attempt to overcome the limitations of the traditional transaction model and most of them provide support for cooperative activities to some extent.

Generally these models are based on the concept of nested transactions. At the top-level, a generalized transaction has all the properties of traditional transactions, that is, serializability, failure atomicity and permanence of effect. However, concurrent sub-transactions may be allowed to cooperate somehow. With respect to the allowed degrees of cooperation between sub-transactions, there are at least three major approaches for extending traditional transactions. The first is to relax strict isolation [19]. Sub-transactions are still serialized but uncommitted results may be shared among them. A sub-transaction that uses uncommitted data will depend on the sub-transaction that produced the data. Such a sub-transaction cannot commit or abort independently and may, once terminated, be required to wait for the commitment of any sub-transaction on which it depends before committing. However, since the serializability property is retained, only limited cooperation between sub-transactions is allowed.

The second approach is to allow the execution order of sub-transactions to be specified as part of the top-level transaction. There is also some work concerned with the execution order of both top-level transactions and their sub-transactions [18]. The mechanism for ensuring the correctness conditions defined by serializability should be extended to take into account the constraints imposed by the user-specified execution order. That is, locks on objects can be transfered from one of its constituent actions to another, while preventing non-nested actions from acquiring conflicting locks on them. Nevertheless if a sub-transaction aborts, the other sub-transactions (that used uncommitted data produced by the first sub-transaction) are not aborted. Since the failure atomicity property is not assured, such a cooperation is still restricted.

	Generalized transactions	Nested transactions
Action on	A set of objects	A set of objects
Inter-concurrency	Competitive sharing with other sub-transactions	Competitive sharing with other sub-transactions
Intra-concurrency	Cooperating sub-transactions	Competing sub-transactions
Composed of	Multiple concurrent but cooperating sub-transactions, operations involved in these sub-transactions can be specially ordered	Multiple concurrent and competing sub-transactions, each composed of a sequence of partially ordered operations on the set of objects
Correctness	User-defined, no isolation	Serializability

Table 1: Nested and generalized transactions [21]

By combining the two approaches discussed earlier, a greater degree of cooperation between sub-transactions can be achieved. Nodine and Zdonik [10] proposed the substitution of the notion of user-defined correctness for the notion of correctness defined by serializability. Because isolation is not required, correctness conditions on the execution order of operations involved in different (but cooperative) sub-transactions could be defined for special application purposes. Table 1 (from Romanovsky *et al.* [21]) compares some common properties of generalized transactions with those of the basic nested transactions.

The work of Caughey *et al.* [4] proposes transaction services to allow multiple threads to be active within a transaction. Both system state and function are encapsulated within objects, and threads progress by invoking objects. Invocation may be either synchronous or asynchronous. With synchronous invocation the invoking thread obeys call-return semantics. An asynchronous invocation can be described as the creation of a new thread whose task is to execute the required function. The creating thread continues independent of the created thread. Upon completion of the invoked function the created thread terminates. However, when asynchronous invocation is allowed explicit synchronization is required between threads and transactions in order to guarantee checked behavior. That is, it is guaranteed that whenever the transaction ends there can be no thread active within the transaction which has not completed its processing. Checked behavior is guaranteed by requiring that, within a transaction, i) every thread created must synchronize with its creator (thread) immediately before termination; ii) a thread cannot synchronize with its creator until the threads it created have performed this synchronization; iii) a transaction may not execute its commit protocol until the threads created by the initiating thread have performed this synchronization. In our opinion, this proposal is similar to conversation concept, where threads can be enter asynchronously but have to be synchronized at the exit.

4.1 Problems and limitations with transaction-based approaches

Generalized transaction models start from the concept of traditional transactions and suffer inevitably from some of the original limitations of the traditional model. On the one hand, these models violate the atomicity property of sub-transactions, leading to inconsistencies between the top-level atomic transactions and sub-transactions, therefore they could offer ambiguous semantics for a transaction. Moreover, violation of the atomicity property will become difficult the operation of built-in recovery mechanisms and have a negative impact on the system performance. On the other hand, the various forms of cooperation among supported by these models are still restricted since none of them permits true concurrent programming – the boundaries of sub-transactions can only be opened up to a limited extent and explicit communications across the boundaries cannot be allowed (without introducing semantic contradictions to the transaction notion). In the best case, the application programmer can use the specification mechanism provided by a model to define some operation conditions for a set of related sub-transactions and the system then in effect carries out some kind of cooperation by enforcing the specified conditions. However, because no integrated mechanism allowing explicit communications between sub-transactions can be provided to the application programmer, achieving close and fine cooperation becomes a particularly difficult task.

In contrast, the CA action concept is based on a totally different philosophy. A CA action allows different concurrent threads to cooperate in performing joint task. Explicit communication and coordination among threads are allowed completely but should be enclosed within the boundaries of a CA action. By the use of CA actions, most of the previously-identified limitations in generalized transaction models can be effectively overcome.

5 Concluding remarks

Real-world applications contain many examples of cooperative concurrent activities, but the widely-used *object and action* model does not provide appropriate support for cooperative concurrency. Generalized transaction schemes provide only limited support for cooperation by somehow opening up sub-transactions.

In comparison with generalized transaction models for handling cooperative activities, the CA action scheme has many favorable characteristics. First, the atomicity property for any (the containing or nested) CA action is well retained. At any level of nesting, CA actions must be isolated from each other, and in case of failure, they must be recovered without any side effect on other CA actions or on the shared objects. Secondly, since the application programmer is responsible for designing cooperation and coordination, at least in principle, such application-specific cooperation will potentially permit a greater degree of concurrency. Thirdly, the *objects and action* model assumes that the transaction itself is correct (software-fault-free). Such an assumption may be questionable, especially in a distributed and heterogeneous environment. The *object and CA action* model requires explicit concurrent programming, and so software design fault might be of greater concern. Associated mechanisms for program error detection, both forward and backward error recovery, and software redundancy [8, 11] merit further investigation and must be fitted into the CA action framework properly.

References

[1] D.M. Beder and C.M.F. Rubira. Uma Abordagem Reflexiva baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Distribuídas Confiáveis. In *VIII Simpósio de Computação Tolerante a Falhas*, Campinas, Brazil, 1999.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addisson-Wesley Publishing Company, 1987.

[3] R.H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, Agosto 1986.

[4] S.J. Caughey, M.C. Little, and S.K. Shrivastava. Checked Transactions in an Asynchronous Message Passing Environment. In *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 222–229, 1998.

[5] F. Cristian. *Dependability of Resilient Computers*, chapter Exception Handling, pages 68–97. Blackwell Scientific Publications, 1989.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, San Mateo, California, 1993.

[7] C.A.R. Hoare. Parallel programming: an axiomatic approach. *LNCS-46, Springer-Verlag*, 1976.

[8] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2 edition, 1990.

[9] J.E.B. Moss. *Nested Transactions: an Approach to Reliable Distributed Computing.* (tech. report 260), MIT Lab. for Computer Science, Cambridge, MA., 1981.

[10] M. Nodine and S. Zdonik. Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications. In *IEEE Int. Conf. On Very Large Data Bases*, pages 83–94, 1984.

[11] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[12] B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, and A.F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[13] A. Romanovsky. Atomic Actions Based on Distributed/Concurrent Exception Resolution. Technical Report 560, Department of Computing Science, University of Newcastle upon Tyne, UK, 1996.

[14] A. Romanovsky, B. Randell, R.J. Stroud, J. Xu, and A. Zorzo. Implementing Synchronous Coordinated Atomic Actions Based on Forward Error Recovery. Technical Report 561, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[15] A. Romanovsky, J. Xu, and B. Randell. Exception Handling and Resolution in Distributed Object-Oriented Systems. In *Proc. the 16th Int. Conference on Distributed Computing Systems*, pages 545–553, Hong Kong, 1996.

[16] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An Overview of the Arjuna Programming System. *IEEE Software*, 8(1):66–73, January 1991.

[17] S.K. Shrivastava, L.V. Mancini, and B. Randell. The Duality of Fault-Tolerant System Structures. *Software - Practice and Experience*, 23(7):773–798, July 1993.

[18] S.K. Shrivastava and S.M. Wheater. Implementing Fault-Tolerant Distributed Applications using Objects and Multi-Coloured Actions. In *IEEE 10th Intl. Conf. Distributed Computing Systems*, pages 203–210, Paris, 1991.

[19] P. Taylor, V. Cahill, and M. Mock. Combining Object-Oriented Systems and Open Transaction Processing. *The Computer Journal*, 37(6), 1994.

[20] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 499–509, Pasadena, California, 1995.

[21] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, A. Burns, S. Mitchell, and A. Wellings. Cooperative and Competitive Concurrency in Fault-Tolerant Distributed Systems. In *DeVa 1st Year Report*, 1997.

[22] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, and I.S. Welch. Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems. Technical Report 619, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.