Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery

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

Department of Computing Science
University of Newcastle upon Tyne, Newcastle upon Tyne, UK

ABSTRACT

This paper presents a scheme for coordinated error recovery between multiple interacting objects in a concurrent object-oriented system. A conceptual framework for fault tolerance is established based on a general object concurrency model that is supported by most concurrent object-oriented languages and systems. This framework integrates two complementary concepts — conversations and transactions. Conversations (associated with cooperative exception handling) are used to provide coordinated error recovery between concurrent interacting activities whilst transactions are used to maintain the consistency of shared resources in the presence of concurrent access. The serialisability property of transactions is exploited in order to help prevent unexpected information smuggling. The proposed framework is illustrated by means of a case study, and various linguistic and implementation issues are discussed.

Key Words — Atomicity, concurrent object-oriented programming, coordinated error recovery, fault-tolerant software structuring, transactions.

1 Introduction

The object-oriented (OO) paradigm supports clean structuring, simplicity of design and software reuse and it is thus likely, if used correctly, to increase software dependability. However, given the complexity of today's computing systems, it is inevitable that even OO systems may still contain residual (typically software) design faults or “bugs”. Complementary approaches and mechanisms, such as software fault tolerance and exception handling, are therefore required in order to cope with software bugs and run-time abnormal events. This is particularly the case with complex concurrent systems because these systems are very prone to errors.

Practical techniques for dealing with software faults do exist, especially for sequential systems, and have been proved successful for some applications (see for examples the collection of papers in [35]). Comprehensive surveys of software fault tolerance issues can be found in [27][31]. However, 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 an atomic (trans)action to provide a means of surviving hardware failures generally assume that user programs are correct [10][16][20][24][29].

In this paper we discuss the problem of providing fault tolerance in concurrent OO software systems and propose a general framework for fault tolerance that integrates two complementary concepts, conversations and transactions. Our framework encompasses strategies for dealing with hardware, software and environmental faults to provide coordinated error recovery between a set of interacting objects.

Our approach has the following novel and favourable characteristics:

- It relies on an OO concurrency model that is general enough to be able to represent the semantics of several different concurrent OO languages, and is thus more consistent with the realities of actual OO languages than existing concepts and approaches which are based on conventional process-oriented models.

- Coordinated error recovery between a set of interacting objects is established as a most general concept that is able to deal with complex interactions between application programs, external environments, and independently-designed shared objects.

- Conversation-type schemes, transactions, and cooperative exception handling are allowed to co-exist in various combinations and are integrated into a uniform framework so that the most effective scheme can be selected to match the given application’s requirements.

- The issue of unexpected information smuggling, (i.e. implicit information passing via means such as shared servers and resources) is treated carefully in order to ensure the effectiveness of attempts at concurrent error recovery. The serialisability property of transactions, provided by means of appropriate concurrency control protocols, is exploited in order to help prevent such information smuggling.

- Means are provided for cooperative exception handling, together with a resolution mechanism for dealing with the problems of concurrent detection of several different errors. These means are generally applicable to any set of objects whether or not of the same type.

The remainder of this paper is organized as follows. Section 2 contains a detailed discussion of fault tolerance issues in concurrent systems. Section 3 proposes a general framework for fault tolerance in concurrent OO systems that is intended to encompass both hardware and software fault tolerance strategies. Section 4 demonstrates the usefulness of this framework through a case study and Section 5 discusses various linguistic and implementation issues briefly. Finally, Section 6 makes comparisons with related work and provides a brief summary.

2 Fault Tolerance Issues in Concurrent Systems

Although techniques for tolerating hardware-related faults based on the use of atomic transactions controlling operations on objects are widely employed in distributed systems, there has been relatively little work on the use of coordinated error recovery amongst concurrent programs, especially for deliberately treating faults in the software itself. The design of fully dependable practical computing systems must incorporate techniques for treating both hardware and software faults and cope adequately with the problems caused by concurrency. In what follows, we will use the term “fault-tolerant software” to describe software with this property, and explore how to take advantage of OO structuring techniques in designing and implementing such software.

2.1 Conversations, Transactions and Exception Handling

The conversation scheme [25] is a canonical software fault tolerance technique for performing coordinated recovery in a set of communicating (and in general cooperating) processes. A conversation generally involves two or more processes, constitutes a two-dimensional enclosure of recoverable activities of multiple interacting processes and creates a “time-space boundary” that process interactions may not cross, as shown in Figure 1(a). The boundary of a conversation consists of a recovery line, a test line, and two side firewalls. A recovery line is a coordinated set of recovery points for interacting processes that are relying on backward error recovery. Such a recovery line is established on entry to the conversation before any process interaction occurs. A test line is a correlated set of the acceptance tests for the interacting processes. The two side firewalls define exclusive membership; that is, a process inside a conversation cannot interact with a process that is not in the conversation. The concept of a conversation permits only strict nesting.

If a process within a conversation raises an exception, then an appropriate error recovery mechanism must be invoked. A coordinated error recovery strategy between all the processes in the conversation is required [5]. Error handlers can use a mixture of forward and backward recovery techniques. For example, the state of a process may be rolled back to the recovery line or compensating actions may be performed to correct the erroneous state. Note that the incorporation of forward error recovery techniques into the conversation framework can deal with errors that affect the environment in which the system runs, since the real world cannot actually go backwards.

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

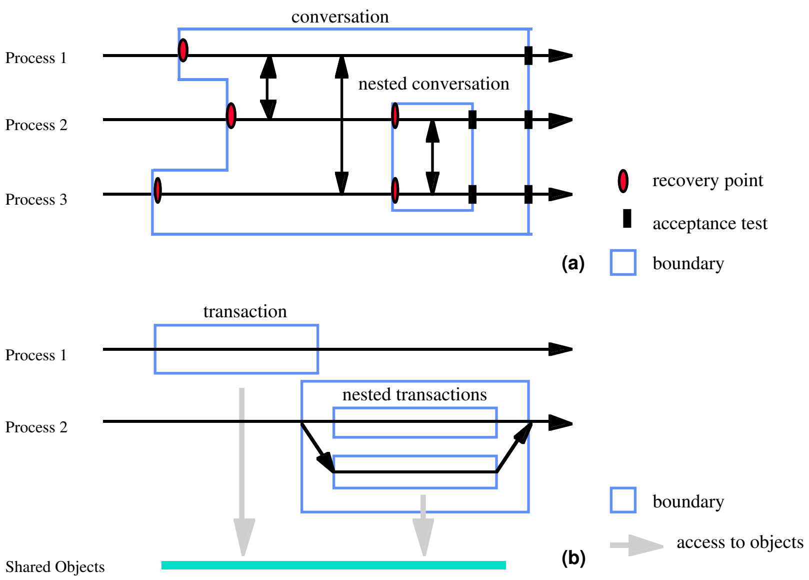

Figure 1. (a) Conversations and (b) Transactions.

A transaction is a logical user action that performs a sequence of basic operations on shared data or objects. An object, or an instance of an abstract data type, encapsulates some data and provides a set of operations for manipulating the data. A transaction can protect shared objects by providing the well known ACID properties — atomicity, consistency, isolation and durability — for all the operations carried out within the transaction [10] (see Figure 1(b), in which interactions with and via shared objects are assumed to take place but are not portrayed explicitly). Such shared objects are designed and exist independently of the user processes. Thus, shared objects are responsible for maintaining their own integrity in the face of concurrent updates by user processes. Nested transactions [24] extend the transaction paradigm by providing the independent failure property for sub-transactions. Therefore, concurrency, i.e. concurrent sub-transactions, may be supported within a transaction, as illustrated in Figure 1(b). However, unlike a conversation in which multiple processes may enter the conversation asynchronously, concurrency between processes is hidden inside a transaction; that is, just one process can enter the transaction and exit later.

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

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

2.2 Prevention of Information Smuggling

The original paper on conversations [25] clearly explained that process interactions could be performed via any means of communication between concurrent processes, such as explicit message passing, or merely reference to common variables and objects. Somewhat surprisingly, much of the subsequent work on conversations concentrated on the recovery line (e.g. using the recovery cache mechanism) and the issue of the test line (e.g. local and global acceptance tests). Little attention has been paid to the "side firewalls" that isolate the set of processes within a conversation from other activities. It is usually assumed that these side firewalls can readily be provided by some conventional protection mechanism. Unfortunately, this is not the case in practice. There are many means by which information may break through the side firewalls and thus defeat the effect of error recovery — this problem is known as "information smuggling".

For example, in an OO system a set of cooperatively defined concurrent objects forming a conversation may need to interact with one or more independently designed objects (e.g. various kinds of server) that provide their own mechanisms for error recovery and fault tolerance. If such service objects can be concurrently accessed by objects in other conversations then implicit information transfers can occur, thus causing unexpected information smuggling between conversations. The problem of controlling information smuggling has proved difficult. For example, servers may become in effect trapped within a conversation, and dynamic object creation/destruction may result in difficulties during backward recovery [12]. In Section 3 we solve this problem and describe a mechanism for coordinated error recovery that allows independently designed objects to be implicit participants in more than one conversation at once. We also discuss the use of forward error recovery to deal with objects for which backward error recovery is inappropriate or infeasible, including for example ones which are outside the computer system.

2.3 Coping with Complexity

Concurrent systems are often very complex, and ill-considered strategies for performing coordinated error recovery may greatly increase their complexity. The issue of complexity control is therefore central to the design of effective fault tolerance mechanisms. We, like other researchers, view it as crucial for practical reasons to support fault tolerance for selected critical components, rather than just for the system as a whole. A framework for fault tolerance that is based on (sub)components rather than the whole system will assist the application programmer in making appropriate tradeoffs between dependability, complexity, flexibility, and performance. This is also consistent with the idea of controlling complexity by structured system design.

Our starting points in considering the structuring of fault-tolerant systems are the concepts of idealized fault-tolerant components [19] and of recursive system structuring [26]. These concepts form the basis of the OO scheme for incorporating design diversity into programs that we described in [37]. In considering the recursive structuring of a system into a collection of components, we mainly concern ourselves with the ways in which a system can itself be sub-divided, i.e. its static structure. However, the pattern of interactions between components of a system is, as pointed out above, quite complex, and can be either explicit or implicit. Such patterns of interaction relate to the identification of the system's dynamic structure. The concept of an *atomic action* can be used to structure such interactions.

An atomic action is an activity between a group of components with the property that no interactions occur between that group and the rest of the system for the duration of the activity [4][19][21]. If a group of components are asynchronous (i.e. concurrently active) and interacting, atomic actions are useful in imposing constraints on the flow of information within the system. A conversation and a transaction are in fact two concrete instances of the notion of an atomic action. (In a database context, the term atomic action is sometimes used as an alternative to transaction.)

The OO paradigm fits closely with the idea of idealized components. In the recursive system structuring scheme, a component can conveniently be thought of as an object [19]. Like components, objects have a well-defined external interface that provides operations to manipulate an encapsulated internal state. Design redundancy can readily be supported — different implementations can be provided for the same interface and combined together to tolerate software design faults. In practice, design diversity can be incorporated into fault-tolerant OO software at different levels of granularity — diverse operations (or parts of an operation), or diverse objects of a specific class, or diverse objects from different classes [37].

3 Coordinated Error Recovery in Concurrent OO Software

The purpose of this section is to describe a framework for fault tolerance in concurrent OO programs that integrates conversations, transactions and exception handling, thus supporting the use of both forward and backward error recovery techniques to tolerate hardware and software design faults, and also environmental faults (i.e. faults that exist in or have affected the environment of the computing system).

3.1 Object-Oriented Concurrency

Computations are carried out in concurrent systems by the cooperation of several separate (or asynchronous) execution threads. Although compilers can restructure programs automatically to extract simple loop-level concurrency, most applications require that process or task-level concurrency must be specified by the programmer [1]. Thus languages for building concurrent systems should support the explicit specification of concurrency and facilitate the description of concurrent activities.

Features for supporting concurrency may be added as an extra layer on top of the OO features, or may be fully integrated with an OO language. We will concentrate on the latter because such solutions encompass the concepts of object and process into a single abstraction. There are essentially two basic techniques for achieving concurrency in the context of OO programming: *asynchronous operation execution* and *active objects* [1]. With the first technique, a new execution thread is generated to execute the body of an operation in response to an invocation request. Concurrency is provided at the level of individual operations (which may be associated with a single object or several different objects). Typical examples include Hybrid [23] and the Actor languages [38]. With the second technique, instead of generating a separate execution thread for each operation invocation, a permanent thread is associated with the whole object, so that the object is regarded as an active (but sequential) process. This technique is used in POOL and Concurrent Smalltalk [38], for example.

Our proposed framework for coordinated error recovery in concurrent OO languages is based on an abstract model of concurrent OO computation from which the concrete model used in a particular language may be derived as a special case. In our model, a concurrent OO system is defined as a collection of interacting objects. Concurrent execution threads correspond to executions of operations on a group of objects. What we are actually concerned with is concurrent executions of operation bodies and coordinated error recovery between a set of such executions. Consequently, there is no need to distinguish between active and passive objects at this level of abstraction. Since a general error recovery mechanism should make no assumptions about the synchronization mechanism that is being used, our model will not specify this mechanism. To avoid extra complexities, we assume in the model that an object must execute just one of its operations at a time. It is therefore conceptually correct by this model to consider objects, rather than individual operations, as participants of a coordinated activity.

Conversations are a mechanism for controlling concurrency and communication between objects that have been designed to interact with each other. In contrast, transactions are a mechanism for hiding concurrency between objects by guaranteeing serialisability for updates they make to objects that have simply been designed to be interacted with (typically termed *shared* objects). Such shared objects have been designed and implemented separately from the applications (i.e. objects) that make use of them, they thus have to be responsible for ensuring their own integrity in the face of concurrent updates and possible failures. However objects that have been designed to interact with each other are collectively responsible for their own integrity. For such objects it may well be possible to use forward error recovery since the designer will know what progress each of the set of objects is intended to make. Backward error recovery can be designed without the need of such knowledge and so is the typical form of recovery used for objects that are individually responsible for their own integrity.

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

An object can be "active" in the sense that it has a script of its own to execute or "passive" in that it relies on other objects to invoke its operations. Some objects, provided by the system or defined by the user, may be just used to store local information and cannot be shared. However, these differences are not specified in the model and will not be exploited for the development of our scheme for coordinated recovery.

3.2 Coordinated Atomic Actions

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

A CA action has the following basic properties:

- A CA action that relies on backward error recovery must provide a recovery line in which the recovery points of the objects participating in the action are properly coordinated so as to avoid the domino effect [25].
- CA actions must provide a test line consisting of a set of acceptance tests, one for each participating object, and a global test for the whole.
- All the objects accessed by a CA action must invoke appropriate forward and/or backward recovery measures cooperatively once an error is detected inside the action, in order to reach some mutually consistent conclusion.
- Error recovery for participating objects in a CA action requires the use of explicit error coordination mechanisms within the CA action; objects that are external to the CA action and can be shared with other actions must be atomic and provide their own error coordination mechanisms (in order to prevent information smuggling).
- Nesting of CA actions is permitted.

On entry to a CA action, a participating object establishes a recovery point if backward error recovery is required and, thereafter, may only communicate with other objects participating in the action and with external objects that are atomic. Note that the participating objects in a particular CA action may enter the action asynchronously. Accessing an external atomic object from within a CA action involves starting some kind of transaction. If all the current participants complete and pass the acceptance tests, then any recovery points taken on entry are discarded, transactions involving external atomic objects are committed and the CA action is exited. If, for any reason, some participating object fails to complete or to satisfy its acceptance test, appropriate recovery measures must be invoked. For this purpose, a CA action is organized as several *CA action attempts*. The first attempt is the normal activity that results from executions of the primary alternates of cooperative participating objects. Subsequent attempts either consist of the activity of the set of exception handlers, or of the activity of doing backward recovery followed by the next set of alternates. Transactions involving external atomic objects must be aborted during backward error recovery. The concept of a CA action thus suggests a quite general solution where both forward and backward recovery techniques can be used in a complementary or combined manner.

Through the use of appropriate protocols it is possible to have a CA action whose participating objects are held in various of the different computers forming a distributed computing system. Indeed users in the environment of a computing system can also be viewed as objects participating in a CA action if they adhere to appropriate protocols — the practicality of this possibility is greatly enhanced by the fact that a CA action can provide a structure and strategy for forward error recovery. For example, the system could send compensatory messages to users in order to correct earlier messages that were later discovered to have been erroneous. In this way, a CA action can effectively deal with cooperative activities between application programs and environments that cannot be rolled back, using forward error recovery [19].

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

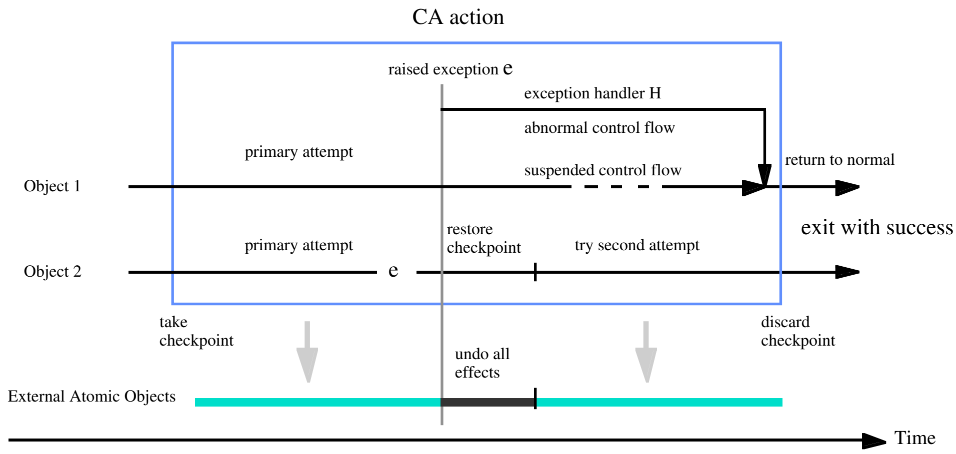

Figure 2. Combined forms of coordinated error recovery.

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

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

Finally, it is worth mentioning that both a conversation and a transaction are really restricted forms of a CA action. For example, when a CA action consists of just a single execution thread accessing one or more atomic objects, it will be in fact be just an ordinary transaction, as shown in Figure 3(a). If a CA action involves several explicit participants, i.e. multiple execution threads enter the action asynchronously, it will constitute a conversation of objects, as shown in Figure 3(b) and (c). Notice that if these participating objects communicate only through external atomic objects, the CA action is equivalent to what is called a "shared transaction" in the database world, but this is only a difference of terminology — it can be considered as a limited form of conversation. In its full generality however, a CA action also encompasses the provision of coordinated error recovery by objects that are directly invoking each other's operations, and the use of forward error recovery as well as backward error recovery.

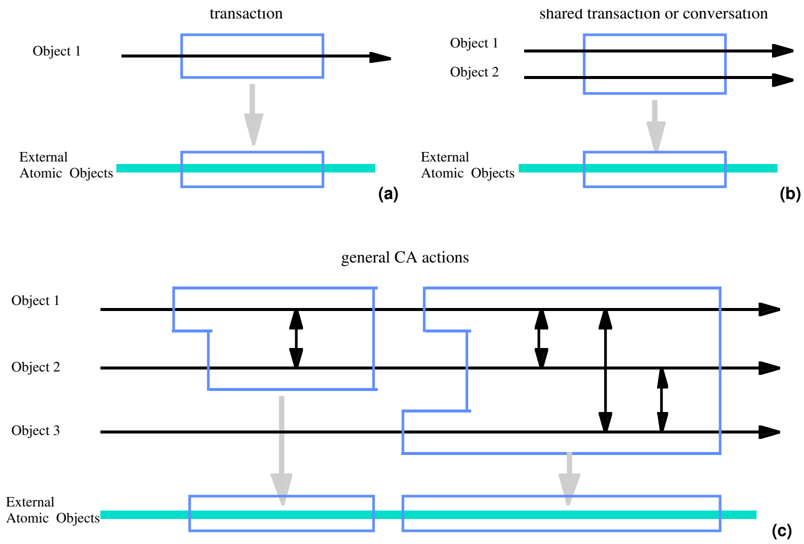

Figure 3. Examples of coordinated atomic actions.

3.3 Exception Handling in CA Actions

In OO systems it is appropriate for exceptions to be represented by instances of classes and therefore have a type [6][18]. This makes it possible to use inheritance to group exceptions together and to define a single handler to cope with a group of related exceptions.

Regardless of whether an exception is raised by one or several of the participating objects in a CA action, the fault tolerance measures must necessarily involve all of the objects that are participating in that CA action [5]. Thus, each participating object in the CA action should suffer the same exception. It is important that all the participating objects have exception handlers for each possible exception (though the use of a default exception handler provided by the underlying system is permitted). These handlers may either invoke appropriate recovery measures, or signal a further exception. Similarly, in the event of error recovery, transactions involving external atomic objects must be either aborted; or else forward error recovery mechanisms must be used to compensate for any erroneous updates they have made to external atomic objects.

It should be noticed that different participating objects in a CA action may raise different exceptions concurrently. For exception handling within the CA action, some exception resolution is thus needed that will combine these multiple exceptions into a single exception. The exception tree proposed by Campbell and Randell [5] is an appropriate mechanism for handling this situation. If several exceptions are raised in parallel, the multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions. Each CA action will have its own tree of exceptions.

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

4 Case Study

We now present a brief case study to illustrate the application of CA actions to a simple sales control system, based on the system considered in [1]. Although many necessary features of the system have been omitted in the interests of simplicity and brevity, the example should be sufficiently detailed to illustrate the mechanisms for coordinated error recovery and fault tolerance provided by a CA action.

The sales control system consists of a database, a set of control points and a set of sales points, as illustrated in Figure 4. Its main function is to maintain a database describing all the products to be sold so that many distributed sales points can obtain the correct prices of the items selected by the customers. Several control points provide interfaces that allow the human managers of the system to update the product information in the database. We assume that such updating is regarded as a very critical activity and consequently, to guard against fraud, the policy is that two human managers, one of whom is at a senior level, have to be involved in and agree to any such updating. Thus, it will be necessary to update the data cooperatively from the control points and this will require the use of coordinated actions. Such updates must also be atomic with respect to sales points that may be querying the database at the same time. Hence, an item is not really deleted or added to the database unless the corresponding action commits successfully.

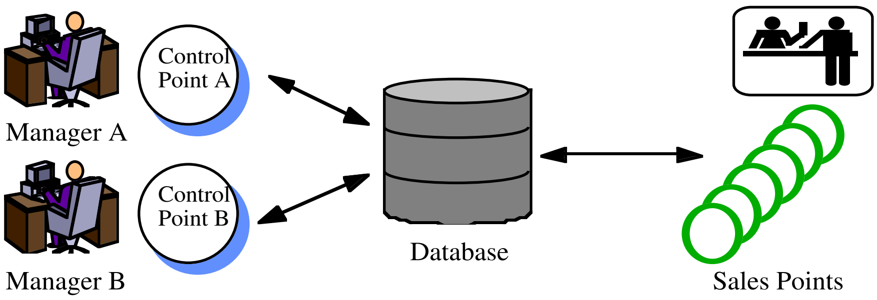

Figure 4. Components of a sales control system.

Several abstract data types are established for the above components. The `DataBase` class models the database of product information and provides operations that can be used from both sales points and control points. For example, an `add` operation can be used from a control point to add a new product descriptor to the database whilst the `retrieve` operation can be used by a sales point to discover the price of a product. Objects stored in the database can be accessed concurrently from different activities and must therefore be atomic. All sales points are modelled as a class named `SalesPoint`. Each sales point is an instance of the `SalesPoint` class and is allowed to retrieve the required data from the database. The control points are defined by two classes, `ManagerA` and `ManagerB` which have differing functions. Instances of the `ManagerA` class provide means for junior managers to update the product information in the database, whilst instances of the `ManagerB` class provide means for senior managers to monitor and, if necessary, corrects updates made by `ManagerA` objects. Thus, updates to the database require a coordinated CA action involving a `ManagerA` object and a `ManagerB` object, and the effect of these updates must be atomic with respect to concurrent price queries from `SalesPoint` objects.

Figure 5 shows the CA action `CoUpdate` that results from asynchronous invocations of `update(..)` and `monitor(..)` operations from two different manager objects. Note that fault tolerance in this system is achieved through the combined use of forward and backward error recovery. If an exception is raised within the CA action, coordinated error recovery will be performed. As illustrated in Figure 5, the exceptions e_1 and e_2 are raised concurrently by the `ManagerA` and `ManagerB` objects. Thus the primary attempt is abandoned and the transaction involving the external atomic object `Shared_DB` is aborted. Meanwhile, the exception resolution mechanism determines that the combined exception e_3 should be raised within the `CoUpdate` action. The `ManagerA` and `ManagerB` objects then execute the corresponding exception handlers for the `update(..)` and `monitor(..)` operations to do forward error recovery. Note that before the completion of forward recovery, `SalesPoint1` would still get the unchanged price. However, once the `CoUpdate` action terminates with successful forward recovery, updated prices will be available for all sales points.

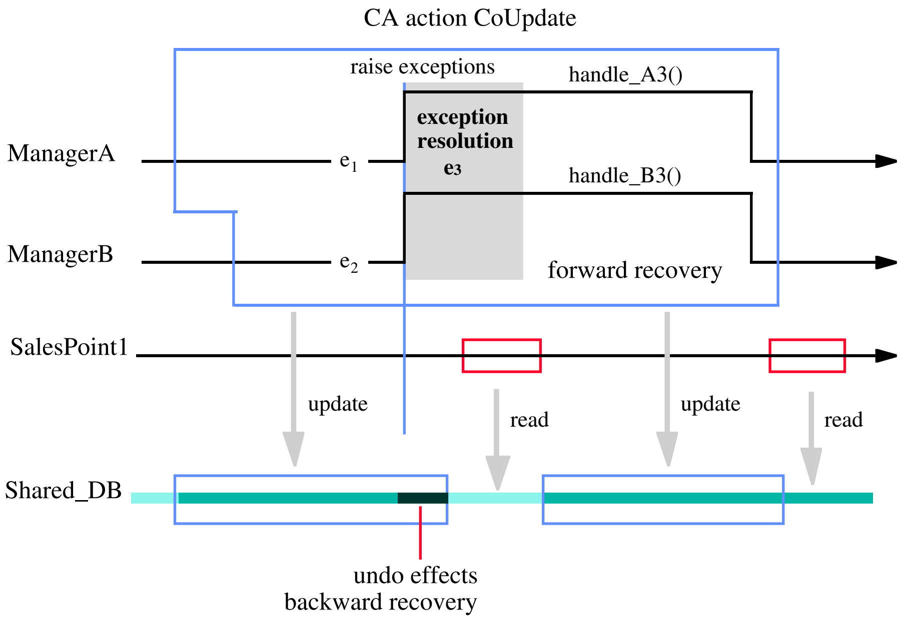

Figure 5. Coordinated atomic actions in the sales control.

5 Linguistic and Implementation Issues

The framework we have introduced for coordinated error recovery could be used to support CA actions in practical concurrent OO languages. However, language design issues are rather complex and in the limited confines of this paper we only outline some of the principal possibilities. Similarly, there are many ways of implementing the CA action concept and here we will only discuss a few major issues that must be faced by any implementation.

5.1 Linguistic Issues

In general, support for CA actions can be provided by either embedding the support into a new language or by extending an existing language. The former approach offers powerful linguistic constructs and provides a fine degree of control because of its tight integration with the underlying language. An example is the Argus language which provides language constructs for the creation of top-level and nested transactions [20]. But a new language may have difficulty in finding practical acceptance. Providing a set of library objects to support CA actions is the simplest approach to implementation — for example, the Arjuna system [29] uses this approach to provide a transaction-based toolkit for writing reliable distributed programs in C++. However, the disadvantage of an approach based on the use of library classes is that it does not offer a good degree of control for coordinated actions because there is no linguistic link between the start and end of an action. The language cannot prevent an action's thread of control from running outside the boundaries of the action unless a set of programming conventions is followed strictly. In the interests of software dependability, it would be better if the language implementation rather than the application programmer was responsible for enforcing such constraints. Thus, a good compromise would be to extend an existing, popular language by adding support for CA actions.

Linguistically, a CA action is like a multi-threaded procedure call and has some similarities to the proposal for a multi-function made in [3]. The programming language Arche [14] has a construct that supports N-version programming [2] called a multi-operation which is a simplified form of a multi-function. However, unlike a CA action, a multi-operation call is a mechanism by which a single object can invoke the same operation on a set of objects that implement it in different ways. A CA action allows several different objects to cooperate in performing a task by coming together. Each participating object plays a different role in the CA action, in other words, each object executes a different operation. These roles should be declared as part of the specification of the CA action since the complete set of participating objects in a CA action must be known at run-time to ensure a synchonized exit. Since a CA action is a mechanism by which a group of otherwise unrelated threads can rendezvous, the syntax and semantics for specifying a call to a CA action must make it possible to identify a particular instance of such an action because, unlike a conventional procedure call, a single invocation of a CA action is made up of several different calls. CA actions should be parameterized allowing them to be bound to different objects on each invocation. A further complication is the way in which variants of the different operations within the CA action should be specified. These language design considerations are the subject of on-going research — they are therefore not discussed further in this paper.

5.2 Implementation Issues

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

On invocation of a CA action, i.e. when one or more objects begin to participate in the action, a globally unique identifier for the action must be generated. As each participating object enters the CA action, its identifier is passed onto the manager and recorded in the *Current-Participant-List* of the CA action. Whenever a CA action accesses an external atomic object (that hence is potentially visible to other CA actions executing concurrently), the manager must ensure that this access is recoverable, for example, by ensuring that atomic objects are only accessed from within transactions.

If backward error recovery is being used, the manager is also responsible for establishing a recovery point for each participating object as the object enters the CA action. If the action completes successfully, any such recovery points are discarded; otherwise the previous states of the participating objects are restored and some recovery measures are invoked. The CA action may terminate with a failure exception despite the use of its own fault tolerance capabilities. Since CA actions can be nested, a failure exception of a sub-CA action will simply cause termination of the current attempt of the enclosing CA action. The outer CA action will then invoke appropriate recovery.

External atomic objects may be accessed concurrently by different CA actions and must have the semantics of atomic data types [36]. Both optimistic and pessimistic concurrency control policies can be used to implement atomic data types [13][36]. The simplest approach is to lock all atomic objects exclusively for use only within a single CA action. This can be relaxed somewhat by allowing concurrent access to external atomic objects from several CA actions provided that none of them tries to modify such objects. Allowing concurrent updates to external atomic objects requires type-specific knowledge about the semantics of the atomic data type to prevent conflicts. However, note that concurrency control and error recovery for external atomic objects is the responsibility of those objects and not the CA actions that access them concurrently.

When all the participating objects in a CA action reach the ends of their operations, the manager will coordinate the termination of the action. If the CA action is successful, all transactions involving atomic objects are committed and any recovery data for participating objects is discarded. If one or more of the participating objects fails its acceptance test, then there are two possibilities depending on whether forward or backward error recovery is used.

If backward recovery is used, all the objects in the action must participate in the process of recovery. The state of each participating object at its entry to the action is restored from a recovery point. All changes made by the action to external atomic objects must be undone by aborting the corresponding transaction.

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

Finally, participating objects in a nested CA action may access external atomic objects that have already been accessed from within a transaction belonging to their parent CA action, but this must be done in a strictly controlled way in order to prevent information smuggling. A set of rules must be enforced and checked by the action manager. Once these external atomic objects are held by the nested action, the parent action will not be able to access them until the nested action terminates.

6 Conclusions

We have introduced a framework for the provision of general fault tolerance in concurrent OO systems that provides coordinated error recovery within an atomic action. Conversations and transactions are integrated into this single abstract structure so as (i) to protect both concurrent objects and globally-accessible shared data and (ii) to cope with both hardware-related failures and software design faults. The proposed framework allows the controlled usage of both backward and forward error recovery techniques (e.g. involving compensatory messages to external activities that may have been affected by erroneous output from the system). This could be very valuable for systems that interact with environmental objects that cannot be simply backed up.

To be adequate for recovery in real concurrent systems, the proposed approach has been designed with the general characteristics of most concurrent OO languages in mind — explicit concurrency between objects within a CA action; implicit concurrency via atomic objects shared between CA actions. Such facilities can provide, we believe, a more practical resolution of the conflicts between basic concepts of fault tolerance and the realities of actual concurrent languages than has been available to date.

6.1 Related Work

The concept of a conversation [25] was aimed at controlling the domino effect and coping with design faults in concurrent processes. It was later improved and extended with various syntactic proposals, such as conversations based on monitors [17], *FT-actions* [15] and *Dialogs* and *Colloquys* [11]. All of these proposals were limited to discussing general ideas and were designed more or less separately from the other facilities of actual programming languages [12]. None of these proposals is based on an OO language model or could be used in a concurrent OO system.

The notion of an atomic action was originally introduced in the context of database systems, it was then explored as a method of process structuring [4][21]. In [5], the conversation scheme was extended to form a general framework for fault tolerance in concurrent process systems that allows the construction of systems employing both forward and backward error recovery supported by nested atomic actions. The work in [15] provided a syntax for this framework based on the CSP language. Though these proposals are very process-oriented, they were extended in [14] to the definition of programming notations in the Arche language — a distributed OO language. Compared with our work, the Arche approach only allows a limited form of coordination in the form of an N-version programming construct for groups of objects called a multi-operation.

Transactions are now a well-known paradigm for the construction of reliable distributed applications [10]. Nested transactions [24] extend the transaction notion by providing the independent failure property for sub-transactions. A recent proposal for extending the SQL 2 database query language to support real-time transactions [9] introduces a means of naming and initiating transactions together with a scheme of pre- and post-conditions by means of which concurrent transactions can be synchronized and user-defined correctness criteria specified.

Many systems have been developed that successfully combine transaction processing with the OO programming methodology — for example Argus [20] and Arjuna [29]. But such research is mainly directed towards data consistency problems and hardware-related failures. Work exists in the distributed computing area on tolerance to failures in concurrent processes that may share data, such as many checkpointing-based schemes for supporting process resiliency [16]. Although most such schemes are similar to conversations (some of them in fact used the idea of a conversation to deal with the domino effect), they are usually based on the assumptions that process failures are only caused by node failures and nodes are fail-silent.

While transactions require techniques for protecting concurrent processes like the conversation scheme in order to be effective in actual systems, such coordinated actions need in turn to treat implicit interactions deliberately, especially between shared resources. Many dual aspects of conversations and atomic transactions are discussed in [30] together with some differences between the two mechanisms. In [32], a proposal is made for dividing a heterogeneous system into two subsystems using conversations and transactions respectively. Within the context of OO languages and systems, our work instead offers a means of integrating the two mechanisms, thereby reducing the complexity of the design of fault-tolerant software for a concurrent system.

6.2 The Way Forward

The work that led to the scheme presented here is a continuation of long-term research into the impact of OO structuring mechanisms on software fault tolerance at Newcastle. One of the starting points for this research was the idea of developing a library of reusable components that could support the construction of fault-tolerant applications without requiring modifications to either the programming language or its underlying run-time system. The idea was to separate the functionality of a fault-tolerant application from the mechanisms it uses to achieve fault tolerance, using a variety of OO mechanisms to achieve this separation. Towards this end, we have specified a set of pre-defined classes that could be used to provide a general framework for fault tolerance in [37] and shown how to implement both forward and backward error recovery in C++ within the context of sequential programs in [28]. This work demonstrates how various forms of object diversity can be programmed in OO languages and shows how to build reusable OO components that encapsulate particular fault tolerance mechanisms using OO mechanisms such as inheritance, delegation, type parameterization and reflection. Our goal now is to develop a further similar set of mechanisms for supporting CA actions in a concurrent OO language.

We believe that the techniques of reflection [22] and meta-level programming based on the use of a meta-object protocol will allow us to achieve a separation of concerns between the functional part of an application and the non-functional part (i.e. the part concerned with fault tolerance measures) by extending the semantics of the underlying programming language transparently without unduly complicating the application-level program [33]. We have been experimenting with a reflective implementation of C++ called Open C++ [7] that provides a limited form of computational reflection. Collaborative work between LAAS and Newcastle within the PDCS project has developed several case studies and prototypes using Open C++ to implement fault-tolerant applications [8]. The use of meta-object protocols to implement atomic objects is presented in [34]. We intend to implement the semantics of a CA action using a CA action manager created at the meta level — this topic is however well beyond the intended scope of this paper.

Acknowledgements

This work was supported by the Commission of the European Communities, in the framework of the ESPRIT Basic Research Projects 3092 and 6362 "Predictably Dependable Computing Systems", and has benefited greatly from discussions with a number of colleagues in the PDCS2 project. Alexander Romanovsky is supported by the postdoctoral fellowship of the Royal Society. Cecilia M. F. Rubira is supported by Conselho Nacional de Desenvolvimento Científico e Tecnológico — CNPq (Brazil), grant n. 200198/90.4. Zhixue Wu is supported by the Research Committee Fellowship of the University of Newcastle upon Tyne.

References

[1] C. Atkinson. *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley, 1991.

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

[3] J.P. Banatre, M. Banatre and F. Ployette, "The Concept of Multi-Functions: A General Structuring Tool for Distributed Operating Systems," in *Proc. Sixth Distributed Computing Systems Conf.*, pp.478-485, 1986.

[4] E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems," *Acta Informatica*, vol. 16, pp.93-124, 1981.

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

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

[7] S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. ECOOP'93*, pp.482-501, 1993.

[8] J. Fabre, V. Nicomette, T. Perennou and Z. Wu, "Implementing Fault Tolerant Application Using Reflective Object-Oriented Programming," *PDCS2 2nd Year Report*, pp.291-314, Newcastle, 1994.

[9] P.J. Fortier, V.F. Wolfe and J.J. Prichard, "Flexible Real-Time SQL Transactions," in *Proc. 15th Real Time Systems Symposium*, pp.276-280, 1994.

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

[11] S.T. Gregory and J.C. Knight, "A New Linguistic Approach to Backward Error Recovery," in *Proc. FTCS-15*, pp.404-409, Michigan, 1985.

[12] S.T. Gregory and J.C. Knight, "On the Provision of Backward Error Recovery in Production Programming Languages," in *Proc. FTCS-19*, pp. 506-511, Chicago, 1989.

[13] M. Herlihy, "Apologizing versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Trans. Database Systems*, vol.15, no.1, pp.96-124, 1990.

[14] 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.

[15] P. Jalote and R.H. Campbell, "Atomic Actions for Software Fault Tolerance Using CSP," *IEEE Trans. Soft. Eng.*, vol. SE-12, no.1, pp.59-68, 1986.

[16] P. Jalote, "Fault Tolerant Processes," *Distributed Computing*, no.3, pp.187-195, 1986.

[17] 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.

[18] A. Koenig and Stroustrup, "Exception Handling in C++," *Journal of Object-Oriented Programming*, vol.3, no.7-8, pp.16-33, 1990.

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

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

[21] D.B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions," *ACM SIGPLAN Notices*, vol.12, no.3, pp.128-137, 1977.

[22] P. Maes, "Concepts and Experiments in Computational Reflection," in *OOPSLA 87*, *ACM SIGPLAN Notices*, vol.22, no.12, pp.147-155, 1987.

[23] D. Mandrioli and B. Meyer, (ed.). *Advances in Object-Oriented Software Engineering*, Prentice Hall, 1992.

[24] J.E.B. Moss. *Nested Transaction: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

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

[26] B. Randell, "Fault Tolerance and System Structuring," in *Proc. 4th Jerusalem Conf. on Information Technology*, pp. 182-191, Jerusalem, 1984.

[27] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," in *Software Fault Tolerance* (ed. M. Lyu), John Wiley & Sons Ltd, pp.1-22, Sept. 1994.

[28] C.M.F. Rubira-Calsavara and R.J. Stroud, "Forward and Backward Error Recovery in C++," *Object-Oriented Systems*, vol.1, no.1, pp.61-85, 1994.

[29] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol.8, no.1, pp.66-73, 1991.

[30] S.K. Shrivastava, L.V. Mancini and B. Randell, "The Duality of Fault-Tolerant System Structures," *Software - Practice and Experience*, vol.23, no.7, pp.773-798, 1993.

[31] L. Strigini, "Software Fault-Tolerance," *PDCS1 1st Year Report*, vol.2, Toulouse, 1990.

[32] L. Strigini, F. Di Giandomenico, and A. Romanovsky, "Recovery in Heterogeneous Systems," *PDCS2 2nd Year Report*, pp.115-166, Newcastle, 1994.

[33] R.J. Stroud, "Transparency and Reflection in Distributed Systems," *ACM Operating System Review*, vol.22, no.2, pp.99-103, 1993.

[34] R.J. Stroud and Z. Wu, "Using Meta-Object Protocols to Implement Atomic Data Types," submitted to *ECOOP95*, 1994.

[35] U. Voges, (ed.). *Application of Design Diversity in Computerized Control Systems*, Springer Verlag, 1988.

[36] W.E. Weihl and B. Liskov, "Implementation of Resilient, Atomic Data Types," *ACM Trans. Programming Languages and Systems*, vol.7, no.2, pp.244-269, 1985.

[37] J. Xu, B. Randell, C.M.F. Rubira-Calsavara and R.J. Stroud, "Toward an Object-Oriented Approach to Software Fault Tolerance," in *Fault-Tolerant Parallel and Distributed Systems* (ed. D.R. Avresky), IEEE Computer Society Press, Dec. 1994.

[38] A. Yonezawa and M. Tokoro, (ed.). *Object-Oriented Concurrent Programming*, MIT Press, 1987.