Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems

A.F.Zorzo[1], A.Romanovsky, J.Xu, B.Randell, R.J.Stroud, and I.S.Welch
Department of Computing Science, University of Newcastle upon Tyne, UK

Abstract

Coordinated Atomic actions (CA actions) provide a scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. In this paper we show how CA actions can be applied to two different areas: safety-critical systems and fault-tolerant parallel systems. We have used a Production Cell case study to show how we can use CA actions to control a safety-critical system, where safety requirements play a fundamental role, and we have used an example based on the GAMMA paradigm to demonstrate how CA actions can be used to add fault tolerance to a parallel computation model. We discuss how CA actions provide these systems with dependability features, and describe our Java framework for constructing CA actions.

Keywords: coordinated atomic actions, dependability, Java, object-oriented systems, safety-critical systems

1 Introduction

The purpose of the research described in this paper is to demonstrate how CA actions can be used as a system structuring tool for designing dependable distributed systems using object-oriented languages by applying them to a Production Cell case study [1] and to a Distributed GAMMA model [2], and to explore some of the issues that arise in providing a distributed implementation of CA actions. In this section we give a brief introduction to CA actions and explain how they support the construction of dependable systems. In Sections 2 and 3 we present the case studies we use in this paper, and describe how we designed both systems using CA actions. In Section 4 we describe an object-oriented framework for implementing CA actions in Java. Finally Section 5 draws some conclusions.

1.1 Coordinated Atomic Actions

The Coordinated Atomic (CA) action concept [3] [4] is a unified approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object-oriented system. This paradigm provides a conceptual framework for supporting both cooperative and competitive concurrency and for achieving fault tolerance. It does this by extending and integrating two complementary concepts - conversations and transactions. CA actions have properties of both conversations and transactions.

[1]Lecturer at PUCRS/Brazil (on leave).

Conversations are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and of competitive concurrency.

Each CA action consists of a number of roles which are performed concurrently by some external activities (e.g. threads, processes). We refer to these external activities as the participants in the CA action. The participants cooperate within the scope of the CA action to perform some coordinated activity on a set of objects. A CA action starts when all roles have been activated and finishes when each role has completed its execution. Objects that are external to CA actions and can therefore be accessed concurrently by more than one CA action must support transactional semantics. In other words, the sequence of operations performed by a given CA action on a set of such objects must be atomic with respect to other CA actions. In this way, it is possible to guarantee good fault tolerance properties for CA actions and to prevent information smuggling between CA actions. The execution of a CA action thus updates the system state (represented by a set of external objects) atomically. In addition, actions can use local objects as the means by which the participants within an action can interact and coordinate their executions. These local objects are similar to the local variables of procedures, but because they can be used by several roles their consistency has to be provided (usually not by the CA action support but by the objects themselves which must guarantee some form of monitor semantics).

CA actions provide a basic framework for exception handling that can support a variety of fault tolerance mechanisms aimed at tolerating both hardware and software faults. The former can be tolerated using two phase commit protocols [5] and stable storage to ensure that the effects of CA actions are permanent. The latter can be addressed using fault masking and design diversity [6] [7].

During the execution of a CA action, one of the roles of the action may raise an exception. If that exception cannot be dealt with locally by the role, then it must be propagated to the other roles in the CA action. Since it is possible for several roles to raise an exception at more or less the same time, a process of exception resolution [8] is necessary in order to agree on the exception to be propagated and handled within the CA action. Once an agreed exception has been propagated to all of the roles involved in the CA action, then some form of error recovery mechanism must be invoked. It may still be possible to complete the performance of the CA action successfully using forward error recovery. Alternately, it may be possible to use backward error recovery to undo the effects of the CA action and start again. 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. Otherwise, a failure exception will be signalled to the external environment.

Figure 1 shows a simple example in which three threads enter a CA action synchronously. Within the CA action the threads communicate with each other and cooperate in pursuit of some common goal. During the execution of the CA action, two of the threads, *Thread1* and *Thread2,* enter a nested CA action in order to do something that *Thread3* is not allowed to interfere with. Note how a nested transaction is used to prevent information smuggling via the external object that all three threads can access.

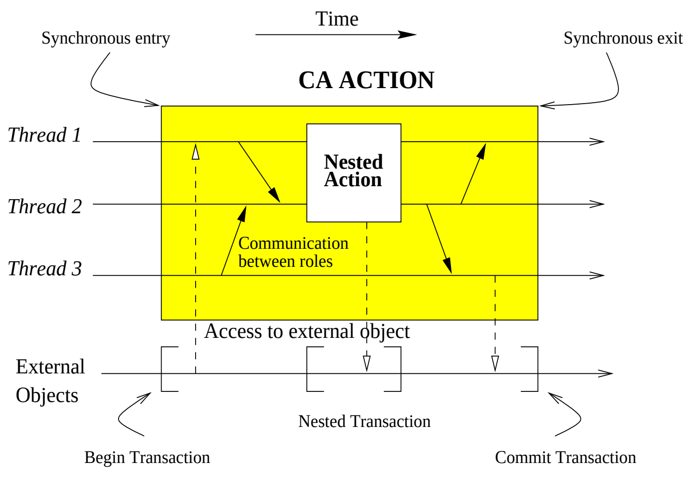

Figure 1: CA Action

1.2 Why Use CA Actions ?

CA actions can be used as a design structuring concept to provide support for the following aspects of dependability:

1) DAMAGE CONFINEMENT: If an error is not detected and limited to a certain extent then its effects may spread throughout the whole system inducing further errors. A CA action can confine the erroneous information flow by enclosing the interaction and cooperation between concurrent activities within its boundaries and by controlling access to external shared objects.

2) COMPLEXITY CONTROL: like the atomic action concept, CA actions can provide a general tool for structuring complex concurrent systems and allowing designers to reason about the dynamic behaviour of systems, thereby controlling complexity and confining damage.

3) FAULT TOLERANCE: For many critical applications, fault tolerance is often the only possible way of achieving the required reliability and safety. CA actions provide a unified framework for handling exceptional situations, into which various proven hardware fault tolerance techniques and existing software fault tolerance techniques, not only simple fault masking or backward error recovery, can be easily incorporated.

4) CRITICAL CONDITION VALIDATION: For many safety-critical systems, once an exceptional event occurs the system must be left in a well-defined safe state. CA actions provide a framework for coordinating execption handling and ensuring that either an acceptable degraded outcome is achieved or the effects of an action are undone, leaving the system in a previously achieved safe state.

5) NESTING: CA actions can be nested. Nested CA actions can provide support for finer damage confinement and enable layered exception handling (i.e. the raising of a failure exception from a nested CA action could invoke appropriate recovery measures in the enclosing action). A nested action also helps to control complexity by further enclosing

a group of basic operations which are part of the containing action.

2 Case Study 1 - Production Cell

2.1 Introduction

The Production Cell case study [1] is a model based on an actual industrial installation in a metal-processing plant in Karlsruhe, Germany. It was developed in the Forschungszentrum Informatik (FZI). The FZI Production Cell is composed of 6 devices, 13 actuators, and 14 sensors. Metal plates are conveyed to an elevating rotary table by a feed belt. A robot takes each plate from the elevating rotary table and places it into the press using its first arm. The robot's first arm withdraws from the press before the press forges the metal plate. After the plate has been forged, the robot's second arm takes the forged metal plate out of the press and puts it on a deposit belt. Finally, a travelling crane picks the metal plate up and takes it to the feed belt again making the system cyclic. In the description of the FZI Production Cell case study, controlling software has to be developed in order to satisfy 21 safety requirements.

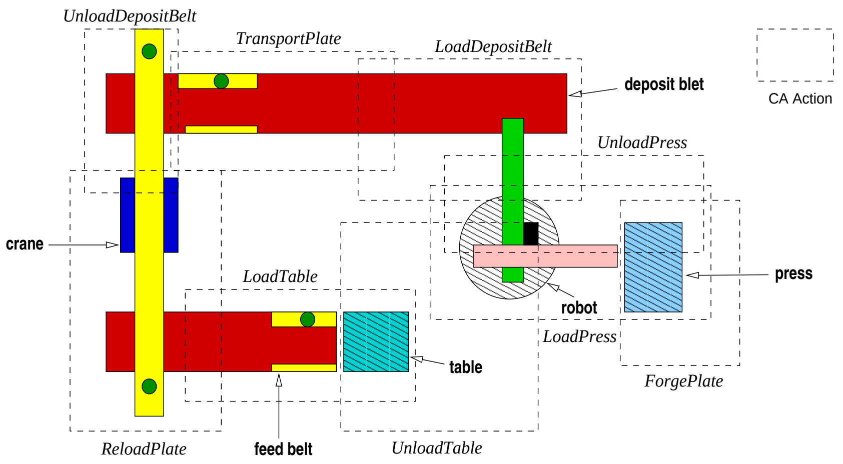

Figure 2: Production Cell and the set of CA actions

2.2 Design of the FZI Production Cell

Our design for the Production Cell separates the safety, functionality, and efficiency requirements between a set of CA actions that are used to control the system and a set of device/sensor controllers that determine the order in which the CA actions are executed. Because the safety requirements are the most important in the system, we satisfy them at the level of CA actions, while the other requirements are met by the device/sensor controllers, which can be programmed in several ways. Figure 2 shows the way in which CA actions are used to control the interactions between devices. Each CA action encloses a pair of devices that must interact in a coordinated fashion to satisfy the safety requirements. If two CA actions overlap, they cannot be performed in parallel because they both involve the same device.

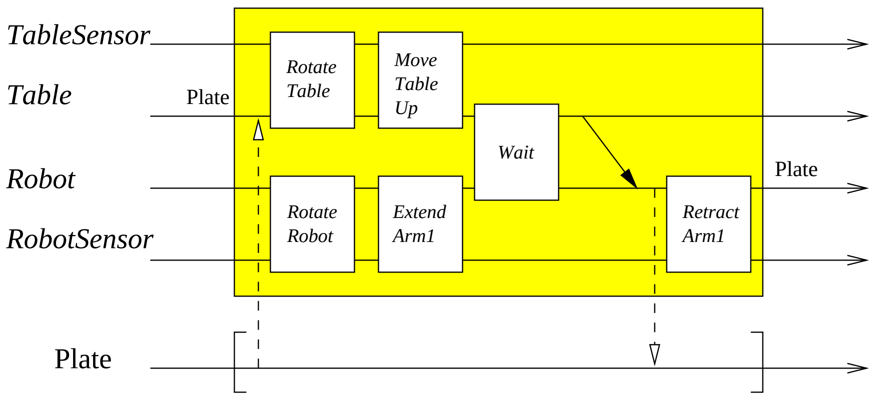

Figure 3: *UnloadTable* CA action

The CA actions that we have designed for the FZI Production Cell example typically have two special roles, one that takes a plate as an input argument, and the other that takes a plate as an output argument. The device corresponding to the role that has the plate as an input argument passes the plate to the role that has the plate as an output argument. Some actions also have sensor roles that check whether or not the devices are in the right position. For example, Figure 3 shows the interaction between the roles of the *UnloadTable* action using some simple nested, i.e. second-level, CA actions. *UnloadTable* has four roles: *TableSensor*, *Table*, *Robot*, and *RobotSensor*. The *Table* and *TableSensor* roles cooperate in order to put the table in the right position for the robot to grab a plate. This cooperation is done by the use of the nested CA actions. In parallel with this, the *Robot* and *RobotSensor* roles cooperate to bring the robot to a position it can grab the plate. When both the table and the robot are in the right position, then the plate can be passed from one device to the other. Note that if the robot finishes first then it waits for the table, and vice-versa.

For safety reasons, all the actions designed for the Production Cell are synchronous actions, i.e. they will only begin when all the participants in the action start to execute their respective roles. The same is true for the end of the action. Each participant is only able to enter a new action when all participants have finished their current roles in the action they were executing together.

2.3 Fault Tolerance in the FZI Production Cell

The safety requirements are the most important requirements for the FZI Production Cell case study. As we explained, we have introduced an action whenever there is an interaction between devices or a plate is forged. We assume that transient faults can happen at any time during the execution of an action. When a fault is detected, an exception is raised and the CA action informs all the other action roles that they have to interrupt their execution. When all the roles are ready to handle the fault, an exception is raised in all roles and they use backward error recovery to leave the system in the state it was when the action started.

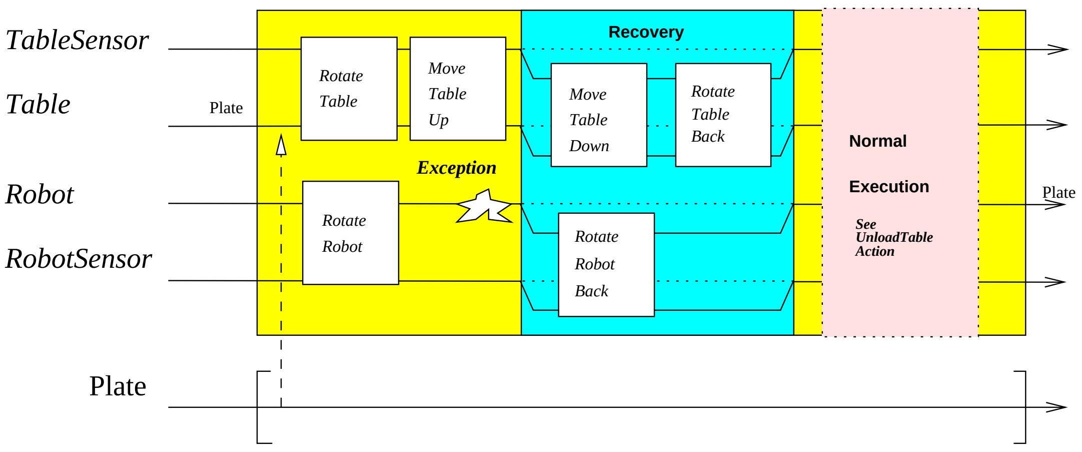

Figure 4: Backward Error Recovery in *UnloadTable* CA Action

In Figure 4 we show an exception being raised after the robot has rotated and before the robot has extended its arm (see Figure 3). This exception is caught by the CA action which then informs all the roles. When all roles are ready to handle the exception they enter a recovery phase during which they are rolled back to their initial state (the state the roles had when they started the action). When the recovery phase is finished the action restarts by reactivating all roles again.

3 Case Study 2 - Distributed GAMMA Model

3.1 Introduction

GAMMA is a model of parallel computation based on the idea of multiset transformations. It behaves in a similar way to a chemical reaction upon a collection of individual pieces of data [9]. Each step of a GAMMA computation involves selecting a set of values from the multiset and then combining them in some way to produce a new set of values. A distributed GAMMA model has also been proposed [2]. Its main novelties are distribution of multisets (each of them is presented as a set of local multisets), and distribution of chemical reactions. For example, the following GAMMA program performs the sum of a set of integers:

```
add = G((R,A)) (Multiset) where R(x,y) = true; A(x,y) = {x+y}
```

3.2 Design of the GAMMA System

To demonstrate how distributed GAMMA computations can be implemented using CA actions, we have used a simple example where numbers from distributed multisets are summed and the result is stored in a multiset. Our distributed GAMMA system is composed of a set of participants (located on different hosts), a scheduler (located on a separate computer) and a set of CA actions, called *GammaActions*. The design has two levels. The first level is concerned with information exchange between computers (participants and a scheduler). This is the level on which the execution of the *GammaActions* is scheduled (or the actions are glued together). At the second level of the design, each *GammaAction* performs a single step of the GAMMA computation by coordinating the interactions between roles and their access to external objects (multisets). On this level numbers are passed between different local multisets, summed, and the result is inserted into the multisets. As shown in Figure 5, each *GammaAction* has three roles: two producers (each of which supplies a number from its local multiset) and a consumer (which computes the sum of these numbers and stores the result in its local multiset).

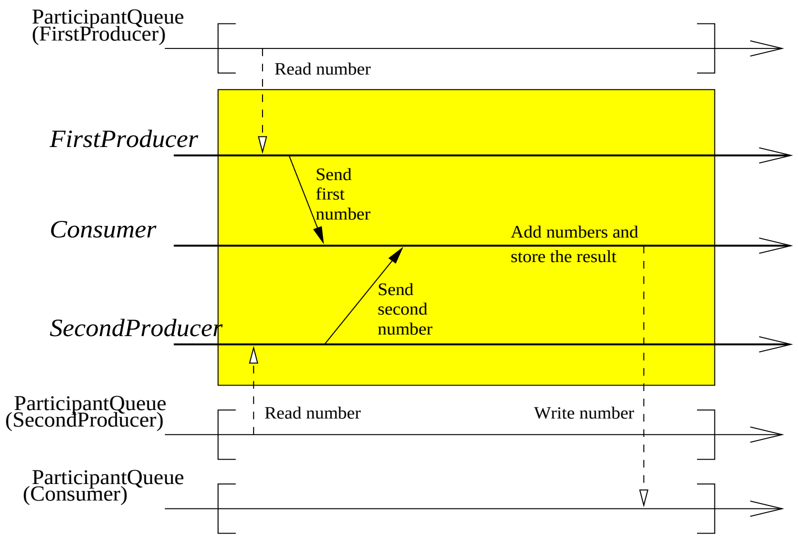
ParticipantQueue (FirstProducer)
Read number
FirstProducer
Send first number
Consumer
Add numbers and store the result
Send second number
SecondProducer
ParticipantQueue (SecondProducer)
Read number
Write number
ParticipantQueue (Consumer)

Figure 5: *GammaAction* CA Action

The participants in the *GammaActions* correspond to the computing resources available to perform the GAMMA computation. A participant starts when it is loaded into a client computer and establishes a connection with the scheduler. Each participant has a local multiset, i.e. a queue in which some part of the global multiset is kept. Each participant informs the scheduler when it receives a new number in its local multiset. The scheduler starts a new *GammaAction* whenever there are at least two new numbers available in local multisets. There can be as many *GammaActions* active concurrently as there are pairs in all local multisets at a given time (although in practice the degree of concurrency may be restricted for implementation reasons). Each participant creates a new thread to execute a role in a *GammaAction* and in this way it is possible for a participant to be involved in several *GammaActions* at the same time without violating the atomicity property (for example, if there are several numbers available in its local multiset). This allows a better parallelisation of the GAMMA computation.

3.3 Fault Tolerance in the GAMMA System

The original GAMMA paradigm [9] assumed that there are no faults in the system. We have chosen to include some fault tolerance in the system in order to demonstrate how CA actions can be used increase dependability and to make the system more realistic. We assume that faults can happen in the *GammaActions*. When a fault happens, a predefined exception `ReactionException` is raised in the thread executing a role in the action (see Figure 6). We assume that nodes and channels are reliable (that means that their fault tolerance, if required, is implemented transparently for our system by the underlying support - e.g. if CORBA was used, then a reliable version of CORBA has to be used).

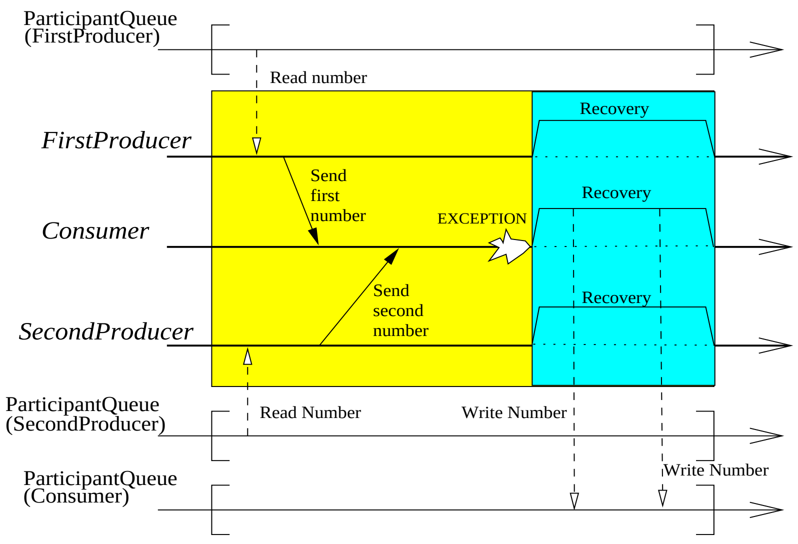

Figure 6: Forward Error Recovery in *GammaAction*

In accordance with the CA action concept we attach exception handlers to each role. After an exception `ReactionException` has been raised, the CA action support mechanism interrupts all the roles in the action and calls the handler for this exception in each of them (see Figure 6). Our design decision is to use forward error recovery in the action in the following way: when the reaction fails, the consumer keeps both numbers by inserting them into its local multiset whilst the producers complete the action as if nothing has happened. Thus, if a fault happens during the action execution, the consumer recovers the system, but in this case there are two new numbers in the consumer's local multiset. We use a special outcome of action *GammaAction* to inform the the scheduler about these new numbers.

GammaActions are atomic with respect to faults in the chemical reaction: the exception handlers guarantee "nothing" semantics for the global multiset (although the local multisets are modified during this recovery).

4 A Framework for Implementing CA Actions

4.1 Overview

We have developed a distributed implementation of CA actions in which each CA action is represented by a set of components: one manager object, a set of role 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 [3]. We have implemented both

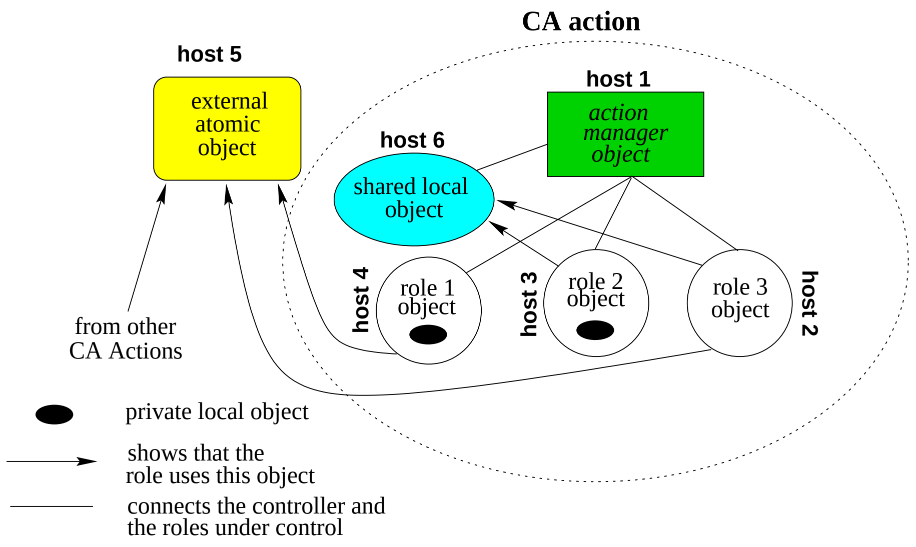

Figure 7: Distribution of CA action components

of the case studies described in Sections 2 and 3 using the Java language [10], an object-oriented language that has been widely used to develop distributed applications over the Internet. We use the Java Remote Method Invocation (RMI) API in order to distribute the components through a network. Figure 7 shows how the components of a CA action are distributed in our approach.

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

4.2 CA ActionManager

To program a new CA action using our framework, the first step is to define a new class that extends the `CAActionManager` class. The `CAActionManager` class contains the methods to deal with entry and exit synchronisation, action exception handling, controlling access to external objects, and accessing the roles that compose the action. Thus, the `CAActionManager` class provides a basic framework for coordinating the participants in a CA action and deals with the application-independent aspects of error handling. Extensions of `CAActionManager`, such as the `GammaAction` shown below (see Figure 8), are responsible for declaring the shared local objects used for coordinating the roles within a particular CA action and also for specifying the application-dependent aspects of error handling (notably, the exception resolution procedure).

By default, the `CAActionManager` class provides a very simple exception handling mechanism. The `exceptionHandling(Exception e)` method is intended to deal with an exception that cannot be handled locally by the roles; it should be redefined by the application programmer. This is the only method that can be redefined by the programmer extending the `CAActionManager` class.

```
class GammaAction extends CAActionManager {
  Channel firstProdChannel;  // Shared local objects used
  Channel secondProdChannel; // by the roles
  public GammaAction() {
    firstProdChannel  = new Channel(this,"first");
    secondProdChannel = new Channel(this,"second");
    SharedLocalObject list[] = {firstProdChannel, secondProdChannel};
    super(list);
  }
  private void exceptionHandling(Exception e) throws ...  {
    // implementation of GammaAction exception handling
  }
}
```

Figure 8: `GammaAction` class definition

Figure 8 shows the implementation of the `GammaAction` class. This new class has two `Channel` objects (`Channel` objects are `SharedLocalObjects`, used for role communication). The `CAActionManager` is informed about these objects, so that the roles belonging to this action can gain access to them later via the `getSharedObject()` method.

4.3 CAActionRole

After a new `CAActionManager` class has been defined, the programmer of the CA action must define the roles that will compose the CA action. This is done by extending the second most important class in our framework: the `CAActionRole` class. Each new class derived from `CAActionRole` contains the main code for one of the roles that compose the CA action. For example, in the GAMMA system, the `GammaAction` is composed of three roles: `Consumer`, `FirstProducer`, and `SecondProducer`, so the programmer must create three new classes by extending the `CAActionRole` class. 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 role. For example, Figure 9 shows the definition of the `Consumer` role for the `GammaAction` example:

```
class Consumer       extends CAActionRole {
   Channel firstProd, secondProd;
   public Consumer(CAActionManager caManager) {
     super(caManager);
   }
   private void execute(ExternalObject objs[]) throws ...{
     Integer num1, num2;
     try {
        firstProd =(Channel)caManager.getSharedObject(this,“first“);
        secondProd=(Channel)caManager.getSharedObject(this,“second“);
        Multiset multiset = (Multiset) objs[0];

        num1 = (Integer) firstProd.receive(caManager);
        num2 = (Integer) secondProd.receive(caManager);
        multiset.Put(num1.intValue() + num2.intValue());
     } catch (Exception e) {
        // initiate exception resolution via CAActionManager object
        // recover, otherwise throw new RoleCAActionException();
    }
  }
}
```

Figure 9: `Consumer` role class definition

In order to bind role objects to particular instances of CA actions, it is necessary to specify a reference to the corresponding `CAActionManager` object whenever an instance of a role object is created. The `CAActionRole` constructor attaches the role to the `CAActionManager` object using the `attachRole()` method of the `CAActionManager` class. Figure 10 shows how an instance of the `GammaAction` CA action would be initialised:

We have chosen to treat individual roles within an action as units of distribution because this allows them to be executed in the locations at which information is produced or consumed. Although several other approaches (e.g. [11] [12]) view CA actions as packages (modules, objects, etc.), such packages cannot be split into parts and distributed. The general idea of attaching exception handling to roles suits role distribution very well, in spite of the fact that these handlers are controlled by the application-independent action

```
// CA action manager declaration
GammaAction gammaAction = new GammaAction();

// roles declaration
Consumer       consumer       = new Consumer(gammaAction);
FirstProducer  firstProducer  = new FirstProducer(gammaAction);
SecondProducer secondProducer = new SecondProducer(gammaAction);
```

Figure 10: Objects declaration for the GAMMA case study

controller (global exception resolution and coordinated action exit are not local decisions). The exception handling which a role provides is application-specific and should be performed in its context. Moreover, because of this, roles deal with external and local object recovery.

4.4 ExternalObjects

The third class in our framework is the `ExternalObject` class. Every new class extended from this class is provided with transactional semantics, i.e. `Begin`, `Commit`, and `Abort` methods, but must provide its own definitions of `CommitState` and `AbortState` methods. We have decided to implement our own simple transactional system for purposes of this investigation, although we could have used an existing one like the Arjuna system [13]. External objects can be recovered from a failure in the CA action in two ways: forward error recovery or backward error recovery. Backward error recovery is provided in an application-independent fashion, while forward error recovery must be performed by the action roles because such recovery is application-specific and cannot be provided by the underlying CA action support mechanism. External objects are passed to CA actions via input parameters when activating a role (see Figure 9 and 11).

4.5 SharedLocalObjects

The fourth class in our framework is the `SharedLocalObject` class. Shared local objects are the objects used by the roles in order to exchange information with each other. They should be recovered by participant handlers in an application-specific way. Our proposal is to assume that each shared object is attached (logically) to an action role which has to recover it as part of action recovery if necessary. The object designer should take advantage of any application-specific knowledge. Recovery of shared local objects is essentially application-specific, and it is only due to this that it can be made fast and simple; if simple recovery is not possible, then these objects should be treated as external ones.

Shared local objects are remote objects in our framework, so there is a chance that these objects can be accessed by adjacent CA actions (e.g. parent, sibling or child actions). However, even though shared local objects can be seen by other actions, only the action that created them (or roles belonging to that action) should be able to access them. To ensure these semantics, we require the current action identifier to be passed as an extra argument to each method invoked on a shared local object from within a CA action (see Figure 9). It is possible to perform an internal check because shared local objects are bound to particular instances of CA actions at object creation time (see Figure 8).

The roles in a CA action may also use private local objects. These objects are not used concurrently, so it is the owner's responsibility to take care of them. If any kind of recovery is necessary they have to be recovered by their owner. If their owner cannot recover them, then an exception should be raised.

4.6 Scheduling CA Actions

The classes described above are used by a programmer to define CA actions. However, CA actions also need to be activated by some scheduling mechanism. We have implemented the scheduling mechanism differently for the two applications presented in this paper. In the FZI Production Cell case study, we used a set of controller objects that exchange messages with each other in order to decide when to enter an action jointly, e.g. when the table controller is ready to execute the *UnloadTable* action it sends a message to the robot controller informing it that the table is ready. The robot controller can then choose to execute that action. In the GAMMA example, we provide a scheduler object that has information about the local multisets held by each participant. This scheduler object chooses which participants will execute an action.

```
...
// Executing a role in a CA action, e.g.  consumer
ExternalObject obj[] = {localMultiset};
gammaAction.inAction(consumer,obj);
...
```

Figure 11: Executing a role in the `GammaAction`

Every thread intending to participate by executing a role in a CA action must call a special method of the `CAActionManager` object corresponding to that action, called `inAction()`, informing the manager which role it intends to execute in the action, and what are the input/output objects the role will need. The manager checks to see if that participant is allowed to play that role in this action, and if so, the participant is synchronised with all other participants. (This is direct synchronisation; we do not support virtual entry and exit synchronisation of participants as described in [3].) When activating the role object, the `CAActionManager` object sends the action identifier, so the role knows it has been called by the right action. Figure 11 shows how to activate the consumer role in a particular instance of a `GammaAction`. Note that when activating a role in an action, the role that is to be activated, and the external objects that are to be used in the action must be sent as parameters of the `inAction()` method.

5 Conclusion

In this paper we have described how we have used Coordinated Atomic actions to implement two different applications: the FZI Production Cell (a safety-critical system), and the Distributed GAMMA model (a fault-tolerant parallel system). Both applications were designed using the CA action concept and an object-oriented approach. We showed that applying CA actions to these two different areas helped to achieve a better structuring by controlling complexity.

Our implementation of both applications in Java showed that the same framework could be used without modification to implement both forward error recovery (Distributed GAMMA system) and backward error recovery (FZI Production Cell). Faults were enclosed by the CA actions hence achieving damage confinement and fault tolerance in both systems.

It is also important to add that both systems were designed in two levels, one for the action activity and another for the scheduling mechanism. In both applications the level of action activity was implemented using the same Java classes. However, for the scheduling mechanism, we used different approaches in order to test the first level, and to see if it would be possible to use different scheduling mechanisms for the same kind of actions. In this way, we were able to demonstrate that our framework for implementing CA actions was generally applicable and could be reused with different scheduling mechanisms.

In conclusion, we believe that the CA actions concept is a promising way of addressing dependability requirements in object-oriented systems. Our framework for implementing CA actions supports rapid prototyping of such systems and provides a clean separation between application-independent and application-dependent error recovery.

Acknowledgements

This research has been supported by the ESPRIT Long Term Research Project 20072 on “Design for Validation” (DeVa). Avelino Zorzo is also supported by CNPq/Brazil (grant no. 200531/95.6). We also thank G. DiMarzo and N. Guelfi (EPFL/Lausanne) for introducing us to the idea of the GAMMA paradigm.

References

[1] C. Lewerentz and T. Lindner. “Formal Development of Reactive Systems: Case Study ’Production Cell’ ”. In *Lectures Notes in Computer Science 891*, Springer-Verlag, January 1995.

[2] G. Di Marzo and N. Guelfi. “Formal Development of Java Based Web Parallel Applications”. To appear in *Proceedings of HICSS’98,* Hawaii, USA, 1998.

[3] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. “Coordinated Atomic Actions: from Concept to Implementation”. Department of Computing Science, TR595, University of Newcastle upon Tyne, UK.

[4] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". In *Proc. of the 25th Int. Symp. on Fault-Tolerant Computing*, IEEE CS Press, USA, 1995, pp. 450-457.

[5] J.N. Grey, "Notes on Database Operating Systems". In *Lecture Notes in Computer Science*, vol.60, Springer-Verlag, pp.393-481, 1978.

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

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

[8] A. Romanovsky, J. Xu and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems". In *Proceedings of 16th IEEE International Conference on Distributed Computing Systems*, pp.545-552, Hong Kong, May 1996.

[9] J.-P. Banatre and D. Metayer. "Programming by Multiset Transformation". In *CACM*, 35(1), pp. 98-111, Jan. 1993.

[10] K. Arnold and J. Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 1996.

[11] A. Romanovsky, B. Randell, R. Stroud, J. Xu, and A. Zorzo, "Implementation of Blocking Coordinated Atomic Actions Based on Forward Error Recovery". In *Journal of Systems Architecture,* 43, pp.687-699, 1997.

[12] A. J. Wellings and A. Burns, "Implementing Atomic Actions in Ada95". In *IEEE Transactions on Software Engineering*, 23(2), Feb. 1997, pp. 107-123.

[13] S. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System". In *IEEE Software*, 8(1), pp. 66-73, 1991.