Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems

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

1 Introduction

The Coordinated Atomic (CA) action concept [1] [2] 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 cooperating and competing concurrency and achieving fault tolerance by extending and integrating two complementary concepts - conversations and transactions. CA actions have properties of both conversations and transactions. 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 competitive concurrency.

Each CA action has roles which are activated by some external activities, i.e. participants, (e.g. threads, processes) and which cooperate within the scope of the CA action. 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 prevent information smuggling between CA actions. The execution of a CA action updates the system state (represented by a set of external objects) atomically. In addition, actions can use local objects. They are the only 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 participants 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 to tolerate both hardware and software faults. In particular, backward and forward error recovery (as well as their combination) can be used.

The purpose of the research described in this paper is to demonstrate how CA actions could be used as a system structuring tool for designing dependable distributed systems by applying them to the Production Cell case study [3] and to the Distributed GAMMA model [4], and to explore some of the issues that arise in providing a distributed implementation of CA actions.

2 Why Use CA Actions to Design Dependable Systems?

CA actions, as a design structuring concept, can provide appropriate 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 used for structuring complex concurrent systems and allow the designer to reason about the dynamic structure of a system, thereby controlling complexity and confining damage.

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

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 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 can naturally attach pre- and post-conditions to their specifications. If necessary, these conditions can be checked at execution time. When any of the pre-conditions are violated due to a fault in a previous action or in the enclosing action, the corresponding CA action will not be executed. An appropriate exception must be raised and be handled properly. Similarly, if any of the post-conditions cannot be met because of a fault within the CA action, an exception must be raised and the effects of the action must be undone, leaving the system in the previous validated state.

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 would 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. In principle, CA actions can be thought of as an abstraction of an agreement protocol - some execution threads come together synchronously, perform some actions cooperatively, and agree upon the outcome.

3 Case Studies

Production Cell. The Production Cell model [3] is composed of 6 devices, 13 actuators, and 14 sensors (see Figure 1). 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, then the press processes 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.

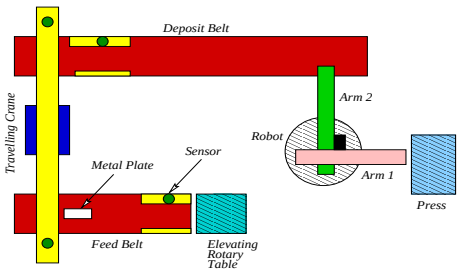

Figure 1: Production Cell

Our design for the Production Cell separates the safety, functionality, and efficiency requirements between a set of actions that occur during the system execution and a set of device/sensor controllers that determine the order in which the 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 relation between the controllers and the actions in our design for the Production Cell.

Typically an action has two roles, one that takes a plate as an input argument, and the other that takes a plate as an output argument. The device correspondig 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 have also sensor roles that check whether or not the devices are in the right position.

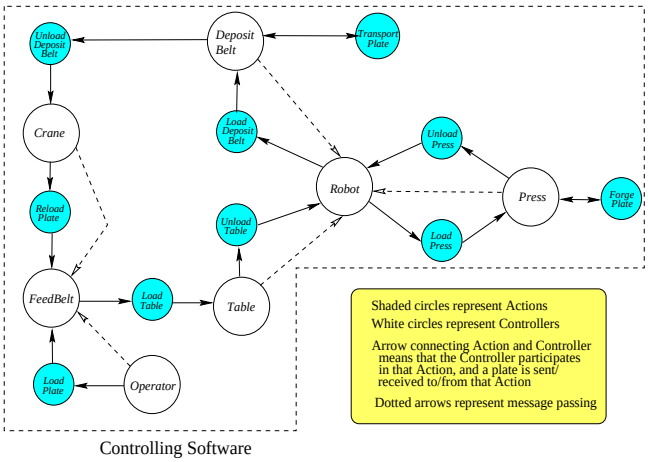

Figure 2: Relations in the Production Cell

For safety reasons, all the actions designed for the Production Cell are synchronous actions, i.e. they will begin only 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 execute a new action when all participants have finished their current role in the action they were executing together.

Distributed GAMMA Computation. GAMMA is a model of parallel computation based on the idea of multiset transformations. It behaves in a similar way to a chemical reaction action upon a collection of individual pieces of data [5]. 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 [4]. Its main novelties are distribution of multisets (each of them is presented as a set of local multisets), and distribution of chemical reactions. 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 CA action scheduler (located on a separate computer) and a set of CA actions. It is designed in two levels. The first level is concerned with information exchange between computers (participants and the CA action scheduler). This is the level on which the execution of the CA actions is scheduled (or the actions are glued together). At the second level of design, individual CA actions perform steps of the GAMMA computation by coordinating the interactions between participants and their access to external objects (multisets). On this level numbers are passed between different local multisets and summed.

A participant starts when it is loaded into a client computer and establishes a connection with the CA action 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 CA action scheduler when it receives a new number in its local multiset. The CA action scheduler starts a new action whenever there are at least two new numbers available in local multisets. There can be as many actions 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 an action and in this way it is possible for a participant to be involved in several actions at the same time (for example, if there are several numbers available in its local multiset). This allows a better parallelisation of the GAMMA computation.

CA actions are activated dynamically to perform the GAMMA computation. Each action has three roles: two producers (each of which supplies a number from its local multiset) and a

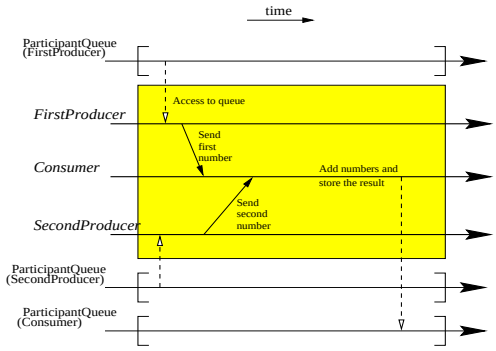

Figure 3: GAMMA CA Action

consumer (which computes the sum of these numbers and stores the result in its local multiset). Each CA action encloses the interactions between the participants in a single step of the GAMMA computation (see Figure 3).

4 Distributed Implementation of CA Actions

We have developed a distributed implementation of CA actions in which actions are represented by objects composed of 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 entry and exit to actions, global exception handling, recovery, consistency and atomicity of external and local objects, and so on[1]. Figure 4 shows how the components of a CA action are distributed in our approach.

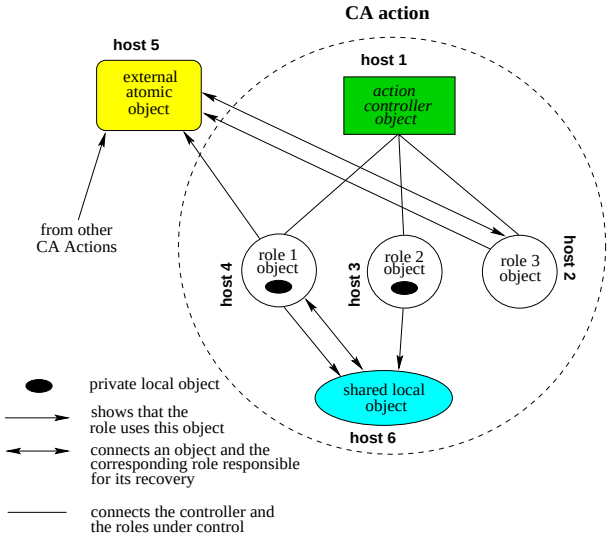

Figure 4: Distribution of CA action components

We have chosen to treat individual roles within an action as the units of distribution because this allows them to be executed in the locations at which information is produced or consumed. Although several approaches (e.g. [6]) view CA actions as packages (modules, objects, etc.), we do not adhere to this approach because these units 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 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.

To understand how local and external objects are distributed more clearly we should discuss how they are manipulated and recovered. Although backward error recovery can be provided in an application-independent fashion, forward error recovery must be performed by the action participants because such recovery is application-specific and cannot be provided by the underlying CA action support mechanism.

Shared local objects 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 participant which has to recover it as part of action recovery if necessary. The object designer should take advantage of any application-specific knowledge. If there is a chance that these objects can be accessed by adjacent CA actions (e.g. parent, sibling or child actions), then some mechanism should be used to guarantee the consistency and atomicity of all modifications carried out within one action. The simplest way could be just to lock the object. Another simplification is using shared local objects which are declared in only one action and are not seen by others. Our conclusion is that the recovery of these 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.

Private local objects are not used concurrently and should be recovered by their owners. If forward recovery is not possible, a failure exception should be raised.

External objects can be supported by a transactional system. It is the responsibility of the CA action support mechanism to ensure that a transaction is started at the same time as the CA action and committed or aborted as appropriate when the CA action completes.

Acknowledgements

This research has been supported by ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa). Avelino Zorzo is also supported by CNPq/Brazil (grant no. 200531/95.6).

References

[1] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. "Coordinated Atomic Actions: from Concept to Implementation". *Submitted to Special Issue of IEEE Transactions on Computers,* 1997.

[2] 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.

[3] C. Lewerentz and T. Lindner. "Formal Development of Reactive Systems: Case Study 'Production Cell' ". Lectures Notes in Computer Science 891, Springer-Verlag, January 1995.

[4] G. Di Marzo and N. Guelfi. "Formal Reverse Engineering of Java Applets Based Client/Server Applications". *Submitted Paper.*

[5] J.-P. Banatre and D. Metayer. "Programming by Multiset Transformation". In *CACM*, vol. 35, no. 1, pp. 98-111. Jan. 1993.

[6] A. Romanovsky, B. Randell, R. Stroud, J. Xu, and A. Zorzo, "Implementation of Blocking Coordinated Atomic Actions Based on Forward Error Recovery". *Journal of System Architecture* (to be published in July/97).