A Distributed Coordinated Atomic Action Scheme

A.Romanovsky and A.Zorzo

Department of Computer Science, University of Newcastle upon Tyne

Coordinated Atomic actions have proved to be a very general concept which can be successfully applied for structuring complex concurrent systems consisting of elements which both cooperate and compete. The canonical Coordinated Atomic action is built of several cooperating participants (roles) and a set of local objects which represent the action state and provide the feature for cooperation. In addition, Coordinated Atomic actions can compete for external objects which have conventional transactional properties. The intention of this paper is to offer a general approach to designing distributed Coordinated Atomic action schemes. Problems of action components partitioning and distribution are discussed. We consider ways of dealing with external and local objects within distributed Coordinated Atomic action schemes; several proposals are discussed in detail. The approach proposed relies on using forward error recovery in the form of distributed and concurrent exception handling and resolution. After discussing the general approach, we demonstrate how it can be applied when the standard distributed model of Ada 95 is used. The presentation of the scheme is sufficiently detailed for it to be used in practice. In particular, a thorough description of the action support and all patterns (skeletons) required for designing application software are given.

1. Coordinated Atomic Actions

1.1. Atomic Transactions and Conversations

Most modern applications are inherently concurrent and distributed, and providing software fault tolerance in such complex systems is a very difficult task. Traditionally it relies on a proper system structuring when fault tolerance features are associated with units of structuring. Atomic transactions (Gray and Reuter 1993) are used to tolerate (hardware) faults in *competitive* concurrent systems (Hoare 1976). Within this paradigm, a set of operations on shared data can be enclosed in a transaction in such a way that transactional support guarantees the well known ACID properties — atomicity, consistency, isolation and durability, for all operations carried out within this transaction.

Conversations (sometimes called atomic actions) (Randell 1975) were proposed as a means of allowing designers to structure *cooperative* concurrent systems (Hoare 1976) and to incorporate software fault tolerance in a disciplined way. Concurrent processes (threads, activities) enter a conversation and cooperate within its scope in such a way that no information flow is allowed to cross the conversation border. This obviously restricts system design but makes it possible to regard each conversation as a recovery region (beyond which erroneous information cannot be spread) and to attach fault tolerance features (application-dependent or provided by conversation support) to each individual conversation (Lee and Anderson 1990). Basically, these features provide error detection and recovery within conversations: when an error has been detected, the corresponding recovery starts. Conversations can use backward error recovery, forward error recovery, or a combination of these (Campbell and Randell 1986; Lee and Anderson 1990). In any case, recovery has to be coordinated, and all conversation participants have to be involved in it. Backward error recovery does not depend on the application much and can be made transparent (or provided, to a considerable degree, by the conversation support) because it uses the rollback of all conversation participants to recover the system. Forward recovery usually relies on an exception mechanism and may incorporate an additional mechanism to *resolve* multiple exceptions raised in several conversation participants (Campbell and Randell 1986). This can be done by imposing a partial order on all conversation exceptions in such a way that a higher exception has a handler capable of handling any lower exception. Exception handlers are attached to each conversation participant, and the basic scheme of forward error recovery is to call handlers for the same exception in all participants. This recovery is application-dependent by nature and this is why only basic support and a general structuring mechanism are provided by conversations. Conversations can be nested; in this case, the execution of the nested conversation is indivisible and invisible for the containing and for the sibling conversations, and the nested conversation results cannot be seen (are not committed) until the containing conversation is completed.

1.2. Coordinated Atomic Actions

The *Coordinated Atomic (CA) action* concept was introduced (Xu et al., 1995) as a unified approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. This paradigm provides a conceptual framework for dealing with both kinds of concurrency (cooperative and competitive) (Hoare 1976) and achieving fault tolerance by extending and integrating two complementary concepts — conversations and transactions. CA actions have properties of both atomic actions and transactions.

Conversations (enhanced with concurrent exception handling) are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

Each CA action has *roles* which are activated by the action participants (some external activities, e.g. threads, processes) and which cooperate within the CA action scope. Logically, the action starts when all roles have been activated (though it is an implementation decision to use either synchronous or asynchronous entry protocol) and finishes when all of them reach the action end. The action can be completed either when no error has been detected or after successful recovery or when the failure exception has been propagated to the containing action.

External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the CA action start and completion has the ACID properties with respect to other sequences. CA action execution looks like atomic transactions for the outside world. One of the ways to implement this is to assume that there is a separate transactional support that provides these properties. A number of such schemes are discussed in (Gray and Reuter 1993). They offer the traditional transactional interface, i.e. operations `start`, `abort` and `commit` transaction, which are called (either by the CA action support or by the CA action participants) at the appropriate points during the CA action execution.

The state of the CA action is represented by a set of *local objects*; the CA action (either the action support or the application code) deals with these objects to guarantee their state restoration (which is vital primarily for backward error recovery). Moreover, local objects are the only means for participants to interact and to coordinate their executions. Two kinds of local objects are treated differently: shared and private local objects. The former are intended for role cooperation, and their consistency is provided on the application level rather than by the CA action support (one of the ways is to design them with monitor semantics (Hoare 1974)). Private local objects are used by individual action participants and represent their internal states.

CA actions can use both backward and forward error recovery as well as their combination. In this respect they inherit all main properties of conversations (Campbell and Randell 1986): the action body is the exception context in which exceptions can be declared, exception handlers are associated with each role, exception resolution is used to resolve several exceptions raised by several roles; the

failure exception is used to inform the containing action when the action fails to recover (in this case the atomicity property requires reversing all changes made during the action execution and doing operation `abort` for external objects). A general object-oriented framework for introducing forward error recovery into CA actions was discussed in (Romanovsky et al., 1996). This paper clearly showed why resolution should be used and why it is vital for distributed systems. In particular, in these systems the overall hardware failure probability is higher than in centralised systems and they are more difficult to program without design faults . Moreover, very often there is a correlation between errors so they happen over a very short period of time in different participants. On the one hand, due to hardware-related operational errors, several nodes can be affected by the same bad conditions or by damage in a channel responsible for traffic between several nodes. On the other hand, because CA action participants were designed cooperatively from a given specification, an error in the specification or a cooperative misunderstanding during the design could affect several or all of them.

There are two ways of involving participants in a conversation or CA action recovery (Campbell and Randell 1986; Romanovsky 1996). In *blocking* schemes (Kim 1982; Jalote and Campbell 1986; Romanovsky 1996; Romanovsky et al., 1997) each participant has either to reach the end of the action, or encounter an error and inform other participants of an exception; it is only afterwards that this participant is ready to accept the information about the state of other participants. *Pre-emptive* schemes (Romanovsky et al., 1996; Wellings and Burns 1996) do not wait but use some means of interrupting all participants when one of them has found an error instead. We could call this blocking and pre-emptive exception handling.

Recovery and exception resolution are much easier to provide within blocking schemes than pre-emptive ones because in blocking schemes each participant is ready for recovery and is in a consistent state when its handler is called. In particular, in blocking schemes all nested actions are to be completed before the recovery of the containing action starts, which is not the case for the pre-emptive schemes. The abortion of nested actions is difficult to program because it requires keeping dynamic consistent information about all actions in the system. The general approach in (Campbell and Randell 1986) requires programming abortion handlers for all actions and calling them when the action is aborted, but, even if application programmers implement such abortion handlers, a very sophisticated protocol (e.g. the one given in (Romanovsky et al., 1996)) needs to be applied to raise abortion exceptions in all nested actions (this must be done recursively and in the right order). The abort protocol should take into account the possibility of several concurrent abortions being

initiated by several participants (in several actions of different levels of nestedness). If centralised schemes (which assume that there is a controller for each action) are used, then this protocol has to be centralised, and this requires a complex dynamic coordination of action controllers (in blocking schemes the action controller may know nothing about the controllers of nested actions). Moreover, we believe that the nested action abortion contradicts the idea of action atomicity because the nested actions are supposed to be indivisible and invisible for the containing and sibling ones.

Although exception resolution seems to be much more important for blocking schemes than for pre-emptive ones, it seems that losing (ignoring) all but one exceptions is equally dangerous for both schemes. Obviously, there is a risk of deadlocks arising in blocking schemes, but we believe that careful programming should make it possible not just to completely avoid them but to simplify the subsequent recovery. Some additional programming rules can make blocking schemes more efficient and decrease delays (time-outs; assertions; checking invariants, pre- and post-conditions; see (Romanovsky 1996) for a detailed discussion; all this is in the line with defensive programming and designing self-checking software (Yau and Cheung 1975)). This allows early detection of either the error or the abnormal behaviour of the process which has raised an exception and is waiting for the other processes. Although no time is wasted in pre-emptive schemes, the features required for interrupting processes are not readily available in many languages and systems. Even when they are, they are usually very expensive to implement (e.g. asynchronous transfer of control in Ada 95 (Burns and Wellings 1995)). Moreover, they usually have complex semantics; it is more difficult to analyse, to understand and to verify programs which use these features. In addition, restrictions are often imposed on the program segment that can be interrupted asynchronously (e.g. Ada 95 tasks cannot accept messages within this segment). Generally speaking, the choice of the scheme depends on the application peculiarities and requirements, on the errors to be detected, on the failure assumptions, etc. But we believe that using blocking schemes suits the idea of forward error recovery better, and our intention is to design a blocking scheme since these schemes have very important advantages and there are many application domains in which their use is fully adequate.

1.3. Previous Research

A set of Ada 95 atomic action schemes was discussed in (Wellings and Burns 1996). Like the general framework of supporting CA actions (Xu et al., 1995), these schemes

use an action manager to control the execution of action participants. A very important decision was to structure any atomic action as a package: the atomic action is presented as a set of procedures that are designed together and declared within one package. These procedures are to be called by external tasks for the action to be initiated. This approach agrees with the CA action concept well since packages are units of system design and since Ada 95 classes (tagged types) usually form packages (Ada95 1995). This approach uses pre-emptive schemes, with no exception resolution, and with a centralised action manager.

A general framework for using atomic actions with forward error recovery in existing practical languages is offered in (Romanovsky 1996). This framework relies on a set of programming conventions and views the exception context of an atomic action as a set of local exception contexts of all participants. It uses local exception handling (within one task), a form of which can be found in many practical languages. Each action participant has to have a set of handlers for all action exceptions; handlers for the same exception are started when an exception has been raised in any of them. All participants are synchronised at the action exit by the action controller that resolves multiple exceptions (if any have been raised). Although this framework neither was intended for CA actions nor addresses distribution problems, we will rely on it to some extent.

A general distributed decentralised resolution algorithm intended for object-oriented systems designed using CA actions is proposed in (Romanovsky et al., 1996). This algorithm is intended for general distributed systems with a simple message passing feature and can be used for pre-emptive decentralised CA action schemes. In addition, this paper briefly discusses a set of general rules describing how external objects should be treated within distributed CA actions.

Paper (Romanovsky et al., 1997) introduces a blocking centralised CA action scheme intended for single-computer applications and discusses how this scheme can be programmed in Ada 95. Actions are associated with packages and their roles with package interface procedures (this is why this approach can be used for module, ADT, or object-oriented programming). Each action has a special controller that synchronises role execution and resolves concurrent exceptions (forward error recovery is used). Within this scheme, procedure blocks are exception contexts of action participants. This scheme is not intended for distributed systems. In our proposal we are going to discuss the structuring CA actions of distributed components and the splitting of CA actions into different components which should be located in different partitions. In addition, we will show below why this scheme (Romanovsky et al., 1997) does not work for distributed Ada 95 systems.

In this paper our intention is to offer a general approach to designing distributed CA action schemes which are blocking, use concurrent exception resolution and a centralised manager. In addition to our reasons for the choices which have been discussed above, we would like to mention the following. *Forward error recovery* is most general, it is cheaper to execute because it is application-dependent. That is why it suits a considerable part of applications with high dependability requirements better (real time, control, reactive, interactive, etc. systems). Moreover, providing concurrent exception handling and resolution is a very important part of the system recovery support (Campbell and Randell 1986; Romanovsky et al., 1996). Our analysis shows that using a *blocking scheme* is the right choice for distributed systems.

After discussing the general approach, we will demonstrate it by using Ada 95. We believe that it is important to come up with a practical scheme which can be used in an *existing language* (the alternative for which is using the language extensions). Employing programming conventions and a set of skeletons which demonstrate how the scheme should be applied within standard languages is one of the solutions (Randell 1993) to which we adhere. We have chosen Ada 95 because it is the most popular standard language used in complex applications with high dependability requirements and because it has standard features for exception handling, concurrent and distributed programming.

In Section 2 we shall discuss the general CA action architecture, which is suitable for distributed systems, and analyse the problems of how the action components should be partitioned and recovered. In Section 3 we shall demonstrate how this approach can be applied using features of distributed programming in Ada 95. Section 4 shows how our approach can employ a small part of the design of Production Cell control system.

1.4. Ada 95

In this Section we shall briefly introduce important the new features in Ada 95 (Ada95 1995) which are used in the implantation of our scheme.

Protected objects are a new concurrency feature similar to Hoare's monitors (Hoare 1974). They are essentially data-oriented because the encapsulated data can be accessed only by calling object entries, functions and procedures. This access is restricted by a well-defined set of rules that allow the consistency of the encapsulated data to be guaranteed: only one entry or procedure can be executed at any time,

entries have barrier conditions (boolean guards) that can either allow the entry or disallow it.

Exceptions in Ada 95 are basically the same as in Ada 83. But there is a new library package, `Ada.Exceptions`, which is very important for our implementation. It provides function `Exception_Identity` which returns the identifier of the exception raised (each distinct exception is represented by a distinct value of type `Exception_Id`). This identifier can be assigned, passed as a parameter, compared, etc. An exception can be raised using its identity by procedure `Raise_Exception` from the same package.

Ada 95 Distributed Annex (Ada95 1995) introduces the concept of *partitions* as the units of distribution. Partitions can be either passive or active. Active partitions, which may have their own threads of control, are configured on processing elements. Passive ones, which have no thread of control, are associated with storage elements. Passive partitions are intended to provide data and subprograms which are shared among active partitions. Library units (e.g. packages) can be categorised to maintain type consistency across distributed programs. In particular, there are the following categories: `Pure` (to supply the same types to multiple active or passive partitions), `Remote_Call_Interface` (to define the interface between active partitions and to manage global data shared by several active partitions). Other categories are `Preelaborate`, `Remote_Types` and `Shared_Passive`. One of the important features of distributed programming in Ada 95, which we are going to use, is that exceptions are propagated through remote procedure calls.

The main peculiarities of distributed system programming within Ada 95 which make the implementation of the CA atomic scheme presented in (Romanovsky et al., 1997) unsuitable are as follows: protected objects cannot be called through partition borders and the exception identifier cannot be passed through the partition borders. Moreover, an appropriate categorisation of the distributed components has to be chosen.

2. CA Actions in Distributed Systems

Our purpose is to understand clearly what are the units of distribution in our approach, and to try to offer as much distribution as possible because this can allow a better use of all advantages of distributed programming. It is clear although, that each particular application should rely on the analysis of the system peculiarities and trade-offs, and use the most suitable and reasonable way of component distribution.

Within our approach action roles are distributed. They are units of distributed action design and are to be executed distributedly in the locations in which information is

produced or consumed. Although several approaches (e.g. (Romanovsky et al., 1997)) 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 handlers to roles suits role distribution very well, in spite of the fact that these handlers are controlled (called) by the action controller (exception resolution and coordinated action exit is not a local decision). The recovery which a role provides is application-specific and should be executed in its context. Moreover, because of this, roles deal with external and local object recovery. Another decision we are taking is that the action controller should be located in a separate *partition* (we use this Ada 95 term and the term 'location' interchangeably).

To understand how local and external objects are distributed more clearly we should discuss how they are manipulated and recovered. Their recovery is an immanent part of action recovery. It is clear that all these objects should be recovered by action participants because this recovery is application-specific and cannot be provided by the underlying support or by the controller.

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 the containing, sibling, or nested CA actions, then some mechanism should be programmed to guarantee the object consistency and atomicity of all modifications carried 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 die to this that it can be made fast and simple; if this is not possible, then these objects should be treated as external ones.

The participant context consists of its private local objects. This is why these objects should be recovered by their owners. Designing them is facilitated by the fact that they are not used concurrently, so, there is no need in concurrency control over them. They should be recovered by role handlers. If recovery is not possible, these objects should be returned to the initial state to guarantee the all-or-nothing semantics of the failure exception.

There are two basic ways of supporting external objects in CA actions. In the first one, all service requests are executed by *transactional support* which, in particular, returns the transaction identity when the transaction is started (it should start at the

same time as the CA action). After this, all external object calls are accompanied by the transaction identity. This can be provided either by the action supported or on the participant level, in which case one of the action participants starts the transaction and passes its identity to other participants if they are going to execute operations on these external objects. In either case it is the responsibility of the application code (handlers) to either abort or commit the corresponding transaction. The second way uses *atomic objects*, each of which has its interface extended by operations `start`, `abort` and `commit`. These objects are involved in the CA action when their `start` operations are called. Our proposal is to attach each of these objects to an action role that should not only involve the object in the action but recover it after an exception has been raised. This recovery is part of role recovery and should be executed by handlers designed together with the role.

Table 1 summarises different choices for the external and local (both shared and private) object distribution.

Location	External Objects	Local Objects
with roles	impossible	private local objects only
with controller	impossible	impossible
one separate partition for all objects	possible but not typical, although can make the support simpler	shared local objects only, possible but not typical
separate partition for each object	more realistic and general for external atomic objects	shared local objects only
other	possible only when transparent object distribution is provided by transactional support	impossible

Table 1. Possibility of locations for external and local objects in distributed CA actions

Figure 1 shows how CA action components are distributed in our approach.

The problems related to hardware fault tolerance are very important and there is no doubt that these faults can affect the approach proposed. In dealing with them, we rely on the existing approaches (e.g. (Powell 1991; Tanenbaum 1995)) which use different sorts of software replication together with some redundant hardware (nodes and links), and on special protocols which guarantee a reliable message delivery. These directions are well-developed, and appropriate approaches can certainly be found and applied in the system in which CA actions are used. Moreover, we believe that the replication support and reliable message delivery should be transparently provided by

the underlying system levels, on top of which CA action schemes function. What we need for our scheme is a hidden partition replication with reliable remote procedure calls, and in the rest of the paper we will assume these. This corresponds well with all ideas on system layering and structuring which provide a clear separation of different concerns during the design of complex systems.

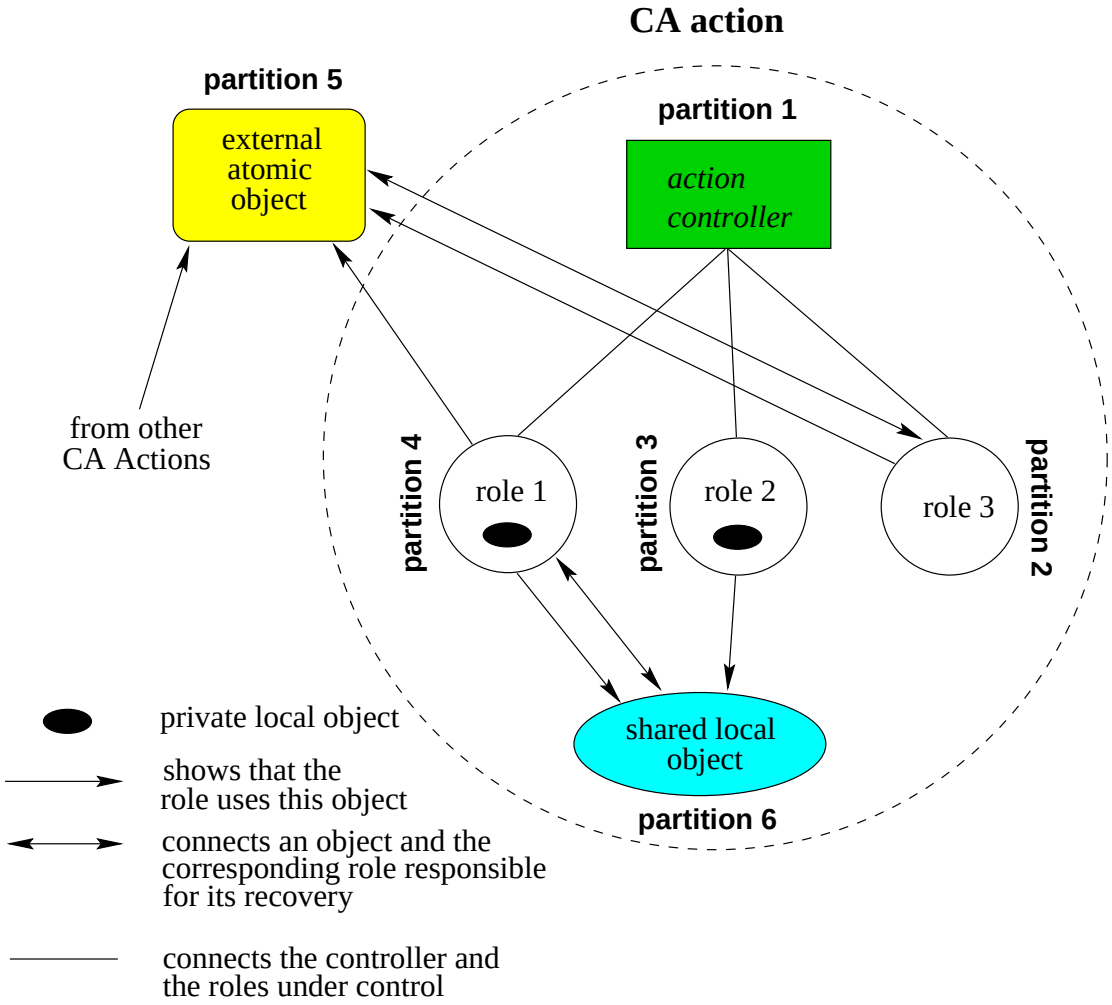

Figure 1. Distribution of CA action components

For example, paper (Wellings and Burns 1996) offers several ways of using passive and active replication in Ada 95 distributed systems. The units of replication are partitions, this is why any of the approaches proposed can be used for our Ada 95 implementation. There are two particular problems involved in our scheme which can make the entire system more vulnerable to hardware faults: using a centralised controller and choosing the blocking approach. The solution for the first one is controller replication (passive or active) with node replication. The second problem can be solved by using watch-dogs, time-outs and participant replication. Although some extension of our approach would be not difficult to design, we believe that all these features should be hidden from the application code (including CA action) designers.

In the following section we will demonstrate how our approach can be used in distributed systems which are designed using standard Ada 95 features and, in particular, its standard distribution model (Ada95 1995).

3. Distributed Ada 95 Scheme

3.1. General Outline

The CA action concept is very general, and we believe that it can be successfully mapped to different concurrent and distributed computational models. We will show now how our approach to designing distributed CA actions (Section 2) can be mapped onto Ada 95. As we have said above distributed CA actions cannot be mapped onto classes or packages because their methods cannot be distributed. Our approach relies on role distribution, so action participants are any Ada 95 blocks located in different partitions, a participant enters the action when it starts the execution of a block. Blocks can be procedures, tasks or object methods — we do not want to bound ourselves to any particular way of role representation which depends on the application and can be distributed or local.

We have chosen the centralised approach for implementing CA actions, so we introduce a package with the action controller which is located in a separate partition. To guarantee the consistency of the controller, we implement it as a protected object which has to be hidden inside the package (because its entries cannot be called distributedly; thus, concurrency control cannot be implemented on the partition level in Ada 95). This protected object synchronises all action participants and raises the resolved exception which is propagated to all of them. The controller package has the only procedure in its interface which is called using Ada 95 remote call by all action participants when they either reach the action end, or find an error. There are several practical reasons why the action controller is not an object of a class in our scheme: i) calling the methods of such objects is a complicated task in distributed Ada 95 systems, and we would like to keep our presentation clear and simple (see (Burns and Wellings 1995) for a thorough discussion of this problem); ii) it is essentially application-dependent (because the resolution tree and the type of the action exception are application-dependent); iii) there is no need for it to be extended or for its methods to be overridden.

Ada 95 has only local (sequential) exception handling, we will apply here the general approach to introducing atomic actions based on concurrent exception handling into languages with local exception handling (Romanovsky 1996) (see our discussion in Section 1.3). This approach requires the synchronisation of all participants at the

action exit with the following exception resolution, and this will be the responsibility of the action controller in our scheme.

Within our scheme, shared local objects are located in separate partitions with a remote procedure call interface and should be accessed with consistency guarantees (the best way of doing this is to implement them as protected objects). But this cannot be explicitly programmed in Ada 95 because protected objects cannot be called remotely in Ada 95. Our solution is to introduce a set of interface procedures, one for each protected object entry or subprogram. These interface procedures are called remotely and, in their turn, call through private protected object entries and subprograms. Private local objects are hidden inside CA action roles. We assume that external objects are either supported by transactional support or implemented in such a way that their application interfaces are extended by operations `start`, `abort`, `commit` (see Section 2).

3.2. Action Participants. Exceptions. Handlers.

As we explained in Section 1.4, exception identities cannot be passed between partitions in Ada 95 (the identities of exceptions with the same name but from different partitions are different). That is why we introduce an enumeration type (one type for each distributed atomic action): this makes it possible to collect all exception values in one partition, to resolve them and to raise the resolved exception in all CA action participants. Each package (including the service one) should be compiled with a package (of category `Pure`) containing types and data common for all action participants (the action exceptions and the enumeration type). For example:

```
package Action_A0 is
       pragma Pure;
       type Action_A0_Exceptions_T is (Exc_A, Exc_B, Exc_C, No_Exc, Fail_A0);
       A, B, C, No_Exception, Failure_A0, Universal_Exception: exception;
       Participant_Number_A0 : constant := 2;
end Action_A0 ;
```

The meanings of the universal and failure exceptions (Campbell and Randell 1986) will be explained in detail later. Note that the same exception raised in different partitions by different action participants will have the same identity because it is declared in the same `Pure` package.

We believe that the best approach which allows any action exception (predefined, raised in the nested procedure, raised in the action exception context, raised by the nested action) to be dealt with in a unified way is to catch all exceptions by clause `others` and to call a special function which returns the value of the corresponding enumeration type. Afterwards, this value is passed to the corresponding controller

package via a remote procedure call. For example, in any participant of action A0 this should be done in the following way:

```
exception -- service handler
        when E : others => A0_resolve(Find_A0(Exception_Identity(E)));
end;      -- pseudoblock
```

Function Find_A0 has a simple structure; it is a library unit (of the normal category) which is linked with each partition in which a participant resides. It is the same for all action participants and is to be programmed by the action designer (designers of participants need know nothing about it). For example, this function for action A0 can be as follows:

```
function Find_A0(EI : in Exception_Id) return Action_A0_Exceptions_T is
begin
       if EI=A'Identity then return Exc_A;
       elsif EI=No_Exception'Identity then return No_Exc;
       elsif EI=Tasking_Error'Identity then return Exc_A; -- predefined exception
       elsif EI=Failure_A0'Identity then return Fail_A0;
       else return Fail_A0; end if;  -- did not find
end Find_A0;
```

Handling the predefined Ada 95 exceptions (Ada95 1995) (e.g. Tasking_Error) is identical to handling the programmer's exceptions: function Find_A0 should manipulate them, their handlers should be included in each participant, the identities of these exceptions in the resolution tree and the corresponding values in type Action_A0_Exceptions_T definition. If they are not treated in this way, they will cause the action to be completed with the failure exception raised.

The action body is represented in each participant as an Ada 95 block. As we have explained, it can be any block located in a separate partition:

```
begin  -- start of action A0 context
          begin
                 -- ... action application code
                 raise A;
          exception -- service handler
             when E : others => A0_resolve(Find_A0(Exception_Identity(E)));
          end;
exception  -- all A0 handlers:
  when No_Exception => ...; -- action A0 success
  when A => ...; handler for A
  when B => ...; handler for B
          raise Failure_A0; -- for example
  when C => ... ; handler for C
          raise Universal_Exception; -- for example
  when Universal_Exception =>   ...; -- clean up
          raise Failure_A0;
end; -- end of action A0 context
```

The resolved exception is raised by the action controller in such a way that it is propagated to all participants, and the corresponding handlers are called in all of them. All handlers for all action exceptions are to be designed and included into the exception context (the action body) of each participant. These handlers can raise

exception `Failure_A0` (to be propagated to the containing action). The last handler should be the handler for exception `Universal_Exception`; this is why all handlers but the last can raise exception `Universal_Exception`. This exception cannot be raised in the main context but its handler must be programmed in all action participants. It executes the last will and/or clean up functions (basically, it assumes that the action recovery is not possible and that the action state has been corrupted and needs restoration (Campbell and Randell 1986)) and raises `Failure_A0` (to be signalled to the containing action). Handler `others` can be used, but it should have all functionalities of the `Universal_Exception` handler.

3.3. Raising Exceptions. Controller Partition. Resolution Procedure

CA action exceptions should be raised by the conventional Ada operation **`raise`**. These exceptions, predefined exceptions and exceptions raised inside nested procedures are dealt with in a unified way in our scheme.

In our scheme action participants reside in different partitions; besides, we introduce additional service partition `Controller_A0_P` (of category `Remote_Call_Interface`) to locate the action controller. This package has the only service procedure `A0_resolve` which should be called by all action participants from their service handlers:

```
package Controller_A0_P is
       pragma Remote_Call_Interface;
       procedure A0_resolve (E: in Action_A0_Exceptions_T := No_Exc);
end Controller_A0_P;
```

Each participant remotely calls it and passes the value of the exceptions it is going to raise. There is a private protected object `Controller_A0` in this package. The resolution procedure is called in this object when the last participant calls procedure `A0_resolve`. This procedure finds the covering resolved exception, which is raised afterwards and propagated via all procedure `A0_resolve` calls to action participants.

Thus, each action `A0` should have package `Controller_A0_P` in a separate partition with private protected object `Controller_A0`. With some additional complication a protected parameterised type can be designed which is suitable for programming controllers for any actions. But, as we have explained before, this is not a simple task because this object essentially depends on type `Action_A0_Exceptions_T` (parameterising of the resolution tree and the participant number is not difficult) and for the sake of simplicity we will not discuss this further.

```
package body Controller_A0_P is

protected Controller_A0 is
entry Finish(E: in Action_A0_Exceptions_T := No_Exc);
private
       entry Wait_All;
       procedure Resolution;
       Finished : Integer :=0;
       Results : ... ; -- all exceptions raised, type Action_A0_Exceptions_T
       Resolved : Exception_Id;
       Let_Go : Boolean := False;
end Controller_A0;

procedure A0_resolve (E: in Action_A0_Exceptions_T := No_Exc) is
begin
       Controller_A0.Finish(E);
end A0_resolve;

protected body Controller_A0 is
    procedure Resolution ... -- assigns Id of resolved exception to Resolved

    entry Finish(E: in Action_A0_Exceptions_T) when True is
    begin
       Finished:=Finished+1;
       --   ...              -- add E to Results
       if Finished = Particpant_Number_A0 then
              Resolution; Let_Go:=True;
       end if;
       requeue Wait_All;
    end Finish;
    entry Wait_All when Let_Go is
    begin
       if Wait_All'Count=0 then
              Let_Go := False; Finished :=0;
       end if;
       Raise_Exception(Resolved); -- in each participant
    end Wait_All;
end Controller_A0;
begin
       null;
end  Controller_A0_P;
```

Exception resolution procedure `Resolution` uses list `Results` and the resolution tree which imposes a partial order on the action exceptions: `A`, `B`, `C`, `Failure_A0`, `Universal_Exception` and on some or all of the Ada 95 predefined exceptions. It assigns the identity of the resolved exception to variable `Resolved`.

3.4. External and Local Objects

The general rule we are applying is that all these objects should be recovered by action participants as part of action recovery; this cannot be done by the underlying support or by the controller because this is forward error recovery, which is application-specific.

Shared local objects should be recovered by the handler of one of the participants in an application-specific way. We assume that each shared object is logically attached to an action participant. As we have explained the consistency of these objects is provided by their designers. The simplest way to do this is by using locks. For

example, a unique CA action name (identity) can be introduced. The corresponding role locks the object when the action starts. Each operation carries the action name which can be checked and only the operations which are issued within the same action are allowed. For example, `op1` carries the action identifier in the following way:

```
op1(my_id : in CAA_name_T; application parameters);
```

A simple extension of this scheme allows passing locks to nested actions and returning them back to the parent action. For example, an object interface can be extended by the following operation:

```
lock(Parent_id : in CAA_name_T := no_id; My_id : out CAA_name_T);
```

We assume that each local shared object is/can be located in a separate partition. One of the simplest ways is to implement it as a package of category `Remote_Call_Interface`, in which case all of its methods can be called distributedly.

Unfortunately, it is difficult to guarantee the object consistency (e.g. to allow only one method updating the object data to be active) only by using locks. The more general way, which relies on the Ada 95 features of data-oriented programming, is to use protected objects. The problem is that in Ada 95 one cannot distributedly call methods of protected objects. Our proposal is as follows: each partition in which a shared local object is located has all object methods in its interface so that they can be called concurrently and distributedly. There is a private protected object which keeps all object data. Each of the interface procedures has a corresponding protected object entry, function or procedure which it calls through. This allows a very sophisticated distributed control and protection to be implemented on the object method level.

Another problem is to support both the passing of these objects into nested actions and a consistent use of them by concurrent sibling actions. This should also be done in an application-dependent way; for example, one of these nested actions can just lock it to use exclusively. The general rule for solving all these problems is that providing the shared local object consistency is essentially application-dependent and that is why it can be made fast and simple. Otherwise these objects should be viewed and treated as external.

Private local objects: should be recovered by the handlers of their owners. In particular, they are returned to the initial state by handler `Universal_Exception` to guarantee the all-or-nothing semantics of the failure exception. To implement this semantics, any existing approach to state restoration can be used (Lee and Anderson

1990), but we believe that for most systems simple application-specific ways can be applied (re-initialisation, cleaning data up, discarding all changes, etc.).

As we explained in Section 2, there are two basic ways of providing transactional interface for *external objects*. Within the first one, all service requests are executed by transactional support, which, in particular, returns the transaction identity when the transaction is started. It is the responsibility of the application code (handlers) to either abort or commit the corresponding transaction. If there is no exception raised, then one of `No_Exception` handlers commits it. If one or more exceptions have been raised and the handlers have succeeded in the action recovery, then one of them commits this transaction. Otherwise, as we have explained, exception `Failure` is raised to be propagated to the containing action. But before this the corresponding transaction is to be aborted; in particular, handler `Universal_Exception` always aborts the transaction.

The second way uses atomic objects, each of which has its interface extended by operations `start`, `abort` and `commit`. The following example, based on the implementation from (Strigini and Romanovsky 1993), demonstrates an interface of external objects:

```
package SpreadSheet is
       pragma Remote_Call_Interface;
       procedure Start_Spread_Sheet (id: out Transaction_Id;
                     splock: in Lock; ...; fatherid: in Transaction_Id:=null);
       procedure Abort_Spread_Sheet (id: in out Transaction_Id; ...;);
       procedure Commit_Spread_Sheet (id: in out Transaction_Id; ...;);

       -- application object methods:
       procedure Set (id: in Transaction_Id; x,y: in Row_Colomn; ...);
       procedure Get (id: in Transaction_Id; x,y: in Row_Colomn; ...);
       ...;
end SpreadSheet;
```

In this case any external object is to be attached to an action participant and recovered by the corresponding participant (as part of the action recovery). Handler `Universal_Exception` needs to call operations `abort` for external objects associated with the participant. Operation `Commit` should not be used in the application code (because even if a participant successfully reaches the action end, an exception can be raised by other participants), so only the handlers which have recovered the action should commit external objects. This is obviously the responsibility of the handlers for exception `No_Exception`. Other handlers (basically all those which raise the failure exception) should abort objects.

3.5. Nested Actions. Exception Propagation. Failure Exception

Our main rules concerning nested actions and using failure exceptions are simple: a failure exception (e.g. `Failure_A0`) is declared for each action and should be propagated to the containing action by the action handlers if they are not able to recover the action (handlers `Universal_Exception` always do this). In this case it is handled by the handlers of the containing action; that is why failure exceptions for all nested actions should be viewed as the exceptions of the containing action.

Our scheme is blocking, which allows us to use the fact that all nested actions are to be completed before the resolved exception is found in the containing one. This means that there is no need in aborting these actions, which can be done only through cooperation among the controllers of containing, nested and sibling actions. This simplifies our scheme tremendously.

3.6. Introducing Synchronous Entry

Although we believe that asynchronous action entry is more suitable for most distributed applications, we have implemented an extended controller which synchronises all participants at the action entry. There is a need for this feature when the action designer wants to make sure that the action starts only when all participants are ready. This can facilitate action design, make the application code simpler and serve as an additional feature in concurrent programming for guaranteeing the mutual exclusion of action execution. In addition, synchronous entry makes it possible to guarantee that the action starts only if all pre-conditions for all participants are satisfied.

This extension is simple. We introduce a new parameterless procedure `A0_entry` in package `Controller_A0_P`. It has to be called remotely by each action participant at the beginning of the exception context execution (e.g. when it enters action `A0`). The procedure calls through additional entry `CA_enter` of action controller `Controller_A0`; the entry code of this protected object is similar to that of entry `Finish`. These calls are requeued to private entry `Wait_All_Enter` which lets all participants continue only when all of them have called `A0_enter` (there is no need for calling the resolution procedure and raising exceptions). It is clear that the calls of procedures `A0_enter` and `A0_resolve` will never be mixed: all participants have to enter the action before any of them reaches the action end or raises an exception.

4. Example: Production Cell

Recently the Production Cell system has been designed at Newcastle University using CA actions. This is a well-known case study (Lewrentz and Linder 1995) which has been used by many research teams to demonstrate the applicability of their approaches. The case study is mainly used for formal specification and verification but we have found it very useful for demonstrating the CA action concept. Our design will be described in a separate paper but just to give a flavour of our approach we will show some parts of it using the distributed CA action scheme proposed.

The task of the case study is to develop a program for controlling a metal-processing industrial production cell. The system comprises a number of devices: a feed belt, rotary table, two-arm robot, press, crane, deposit belt, together with 14 sensors and 13 actuators to control these devices. We have design the system as a set of CA actions, each of which controls the processing of one plate in one device or the cooperation between two devices while executing one step of processing a plate. The first CA action is `FB_RT_Action` action which has five roles: a plate, feed belt, rotary table, extreme sensor and belt actuator (Figure 2 shows the action with asynchronous entry). When the action is completed, another plate can enter the next instance of this action together with other participants (if and when they are ready). The moved plate is free to enter the following action `RT_RA1_Action` together with the rotary table, robot arm 1 and the corresponding sensors and actuators.

Our Ada 95 CA action scheme (Section 3) allows action `FB_RT_Action` to be programmed in the following way. Its participants are located in different partitions and represented by the corresponding procedures:

```
procedure Plate;
procedure Feed_Belt;
procedure Feed_Belt_Extreme_Sensor;
procedure Feed_Belt_Actuator;
procedure Rotary_Table;
```

Shared local objects are used for inter-role communication (e.g. they can be of types `mailbox`, `queue`, `rendezvous`). Private local objects represent the states of roles. Forward error recovery is more suitable for this application because it allows fast application-specific recovery without losing the plate and without moving the mechanical devices back. The code of each participant is implemented using the skeleton proposed in Section 3. Action controller `Controller_FB_RT_Action` is located in a separate partition:

```
package Controller_FB_RT_Action_P is
```

```
        pragma Remote_Call_Interface;
        procedure FB_RT_Action_resolve (E: in FB_RT_Action_Exceptions_T := No_Exc);
end Controller_FB_RT_Action_P ;
```

and is programmed in the way explained above.

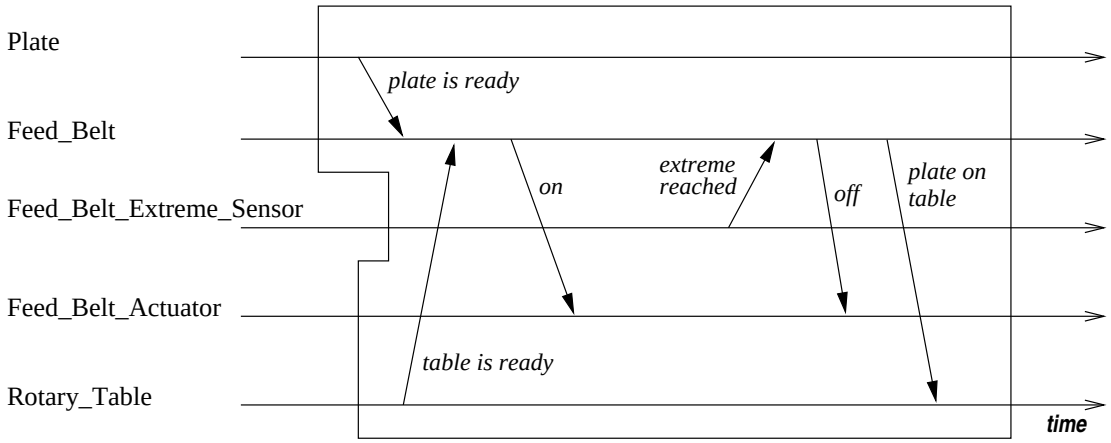

Figure 2. Production Cell design: CA action `FB_RT_Action`

If one or more exceptions have been raised by these participants, the action controller resolves them and raises the resolved exception in all participants; after this the corresponding handlers recover the action.

As we mentioned in Section 3.6, having synchronous entry can facilitate the action design. This can be demonstrated by this example because, with synchronous entry, role `Feed_Belt` can start only when the plate is on the feed belt, so the first message from role `Plate` can be omitted (as well as the first message from role `Rotary_Table`).

5. Discussion

There are some similarities between our scheme and one of the Ada 95 atomic action schemes in (Wellings and Burns 1996). This is a distributed scheme which action participants are located in different partitions, and the data, that they share are put in a separate partition. Several coordinators (controllers) are introduced into the scheme: a local controller for each participant, a distributed action controller (located in a separate partition), a shared data controller. The authors (Wellings and Burns 1996) give a complete set of templates for programming distributed atomic actions with forward error recovery. This is a pre-emptive centralised atomic action scheme which uses forward error recovery. Unfortunately, the need for exception resolution is just mentioned, and the scheme ignores all but one concurrent exceptions (this may be suitable for some applications but ,generally speaking, is not acceptable). We believe that exception resolution should be used; but when introduced, it affects the entire

scheme design. Nested actions and their abortion are not discussed in this scheme but seem to be an essential problem which we do not have in ours, because, unlike to this one, our scheme is blocking. We believe that different handlers should be used for different application exceptions (which is not the case for the scheme in (Wellings and Burns 1996)), it does not seem to make sense to have one unified handler `Others`. The scheme requires a local controller for each participant mainly because it is pre-emptive. Another drawback is vague treatment of shared resources as the only means of communication (we believe that the concepts of external and local objects, which are parts of the CA action concept (Xu et al., 1995), should be used here in the general case).

Although our Ada 95 implementation is in some sense restricted and we have not been able to make it fully object-oriented, we believe that in the future the approach can be successfully applied in other object-oriented languages or systems. In particular, the controller can be designed as a class (in our implementation it is rather application-dependent, so, we did not do this). Our scheme allows objects to be action roles by executing one method as part of the action, but some additional support is required to guarantee participant consistency by imposing restrictions on the execution of methods of these objects. The immanent restriction of using CA actions in distributed systems is that it is not allowed to distribute methods of an object or a class, so CA actions cannot be static units of system structuring (e.g. classes).

We believe that, within the general object-oriented approach, action participants should be designed as objects. An important feature which can make this simpler is treating exception handlers as special private methods (e.g. `Handler_A`, `Handler_B`). This will allow all action participants to be designed by inheriting from the same class `Action_A0_Roll`, which is common for all of them. Together with designing the action controller as a class, this can make our approach fully object-oriented. The problems to be addressed are: overriding and inheriting handlers and the resolution tree, extending the resolution tree, re-using the tree and/or handlers (e.g. what happens if we add a new exception, or override the old one; how we can re-order exceptions with minimum handler re-design; whether we should re-design and override all handlers on the tree path between the newly inserted one and the root; etc.).

6. Conclusions

The purpose of this research is to discuss how distributed CA action schemes can be introduced and to outline the main problems and solutions for using CA actions in distributed systems. The main components of CA actions are roles, external and local

objects, the controller; that is why we discuss different approaches to CA action component distribution. We concentrate on forward error recovery, which is application-dependent by its nature, and propose providing the recovery of external and local objects by roles, and, in particular, to attach each object to a role. We give the rules of object recovery to be followed which guarantee action atomicity and data (object) consistency.

The approach we have chosen relies on using blocking and centralised CA action schemes based on exception resolution. The action controller synchronises action roles and resolves the exceptions raised. We have given the reasons for our choices and, in particular, shown why we believe using blocking actions with exception resolution is the best approach for many applications.

Our general approach has been demonstrated using Ada 95 features: protected objects, exceptions, distributed programming. An Ada 95 CA action scheme is presented as a complete set of patterns and programmer's conventions to be followed when designing systems with high dependability requirements. We use Ada 95 because this is a standard language with a standard distribution model, and this allows us not only to demonstrate our approach but to present a scheme which is ready for use in designing such systems.

Acknowledgements. Thanks go to our colleagues: B.Randell, J.Xu, R.Stroud, I.Welch (Newcastle University), A.Burns, A.Wellings and S.Mitchell (York University). This research has been supported by the ESPRIT Long Term Research Project 20072 on “Design for Validation” (DeVa). A.Zorzo is also supported by CNPq (Brazil) under grant 200531/056.

References

Ada95 (1995). Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E), Intermetrics, Inc.

Burns, A. and Wellings, A. (1995). *Concurrency in Ada*, Cambridge University Press.

Campbell, R.H. and Randell, B. (1986). Error recovery in asynchronous systems. *IEEE Trans. on Soft. Eng.* SE-12(8): pp. 811-826.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. San Mateo, California, USA, Kaufman Publishers.

Hoare, C.A.R. (1974). Monitors - an operating system structuring concept. *Communication of ACM* 17(10): pp. 549-557.

Hoare, C.A.R. (1976). Parallel Programming: an Axiomatic Approach. Languages Hierarchies and Interfaces, Lecture Notes in Computer Science, LNCS-46. Eds. G. Goos and J. Hartmaur, Springer-Verlag: pp. 11-39.

Jalote, P. and Campbell, R.H. (1986). Atomic Actions for Fault-Tolerance Using CSP. *IEEE Trans. Softw. Engng.* SE-12(1): pp. 59-68.

Kim, K.H. (1982). Approaches to mechanization of the conversation scheme based on monitors. *IEEE Trans. Softw. Engng.* SE-8(3): pp. 189-197.

Lee, P.A. and Anderson, T. (1990). *Fault Tolerance: Principles and Practice*. Wien - New York, Springer-Verlag.

Lewrentz, C. and Linder, T. (1995). *Formal Development of Reactive Systems: Case Study Production Cell. Lecture Notes in Computer Science, LNCS-891.*, Springer-Verlag.

Powell, D., Ed. (1991). *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Research Reports ESPRIT. Berlin, Springer-Verlag.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Trans. Softw. Engng.* SE-1(2): pp. 220-232.

Randell, B. (1993). *Approaches to Software Fault Tolerance*. In *25th Annual LAAS Conference, Toulouse, France, LAAS* pp. 33-42.

Romanovsky, A. (1996). Atomic actions based on distributed/concurrent exception resolution, TR 560, Computing Department, University of Newcastle upon Tyne.

Romanovsky, A. (1996). Practical exception handling and resolution in concurrent programs, TR 545, Computing Department, University of Newcastle upon Tyne (accepted for Computer Languages J.).

Romanovsky, A., Randell, B., Stroud, R., Xu, J. and Zorzo, A. (1997). Implementation of Blocking Coordinated Atomic Actions Based on Forward Error Recovery. *Journal of System Architecture* (to be published in July).

Romanovsky, A., Xu, J. and Randell, B. (1996). *Exception handling and resolution in distributed object-oriented systems*. In *16th Int. Conference on Distributed Computing Systems, Hong Kong, IEEE CS Press* pp. 545-553.

Strigini, L. and Romanovsky, A. (1993). Implementing atomic transactions in Ada, (internal Technical Report), IEI CNR, Pisa.

Tanenbaum, A.S. (1995). *Distributed Operating Systems*. New Jersey, Prentice-Hall.

Wellings, A.J. and Burns, A. (1996). Implementing Atomic Actions in Ada95, TR YCS-263, Department of Computer Science, University of York.

Wellings, A.J. and Burns, A. (1996). Programming Replicated Systems in Ada 95. *Computer J.* 39(5): pp. 361-373.

Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R. and Wu, Z. (1995). *Fault tolerance in concurrent object-oriented software through coordinated error recovery*. In *25th Int. Symp. on Fault-Tolerant Computing, Pasadena, USA, IEEE CS Press* pp. 499-508.

Yau, S.S. and Cheung, R.C. (1975). *Design of Self-Checking Software*. In *Int. Conference on Reliable Software, LA, California, USA* pp. 450-457.