Implementing Synchronous Coordinated Atomic Actions Based on Forward Error Recovery

A.Romanovsky, B.Randell, R.Stroud, J.Xu, A.Zorzo

Department of Computer Science
University of Newcastle upon Tyne, UK

Coordinated atomic action concept is proposed as a means for providing fault tolerance in complex object oriented systems which incorporate both cooperative and competitive concurrency. This paper has two purposes: to discuss a particular implementation of this concept and to address many implementation issues which are common for any experiments with this concept. Our implementation relies on a thoroughly designed set of programming conventions for the standard Ada (Ada95) language and uses forward error recovery which incorporates asynchronous exception handling and concurrent exception resolution. We utilise the peculiarities of Ada as much as possible, which makes our approach practical and useful for many critical applications with high dependability requirements. This scheme offers a basic framework for using coordinated atomic actions and allows us to continue experimenting with them.

Keywords: concurrent systems, atomicity, coordinated error recovery, fault tolerant software structuring, conversations, transactions.

1. Introduction

1.1. Atomic Actions

Providing fault tolerance in complex concurrent systems is a very difficult and challenging task. Atomic transactions [7] are traditionally used to implement fault tolerance in *competitive* concurrent systems. Within this paradigm, a set of operations on shared data can be enclosed in an action in such a way that transactional support guarantees the well known ACID properties - atomicity, consistency, isolation and durability, for all operations carried out within the transaction.

Conversations (and more generally, atomic actions) [14] were proposed as a means of allowing designers to improve the structuring of *cooperative* concurrent systems and to incorporate fault tolerance in a disciplined way by attaching it to actions. Concurrent processes (threads, activities) enter an action and cooperate within its scope in such a way that no information flow is allowed to cross the action border. This obviously restricts system design but makes it possible to regard each action as a recovery region (beyond which the erroneous information cannot be spread) and to attach fault tolerance features (application-dependent or provided by action support) to each individual action [12]. Basically, these features provide error detection and recovery within actions: when an error has been detected, the corresponding recovery starts. Atomic actions can use either backward error recovery or forward error recovery, or a combination of these [3, 12]. In any case, recovery is to be coordinated and all action participants have to be involved in it. Backward error recovery does not depend so much on the application and can be made transparent (or considerable support can be provided by the action run time system) because it uses the rollback of all action 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 action participants [3]. This can be done by imposing a partial order on all action exceptions in such a way that a higher exception has a handler capable of handling any lower exception. Exception handlers are attached to each action (usually to each action participant), and the basic idea of forward error recovery is to call handlers for the same exception in all action participants. This recovery is application-dependent by nature and this is why only basic support and a general structuring mechanism are provided by atomic actions. These actions can be nested; in this case, the execution of the nested action is indivisible and invisible for the containing action, and the nested action results cannot be seen (are not committed) until the containing action is completed.

1.2. Coordinated Atomic Actions

Recently the *coordinated atomic (CA) action* concept has been introduced [19]. It extends the atomic action concept in several ways. First of all, *atomic (transactional) objects* can be concurrently used 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 ACID properties with respect to other sequences. It is assumed that a separate support provides these properties (e.g. a number of schemes is discussed in [7]); in particular, it offers the traditional transactional interface: operations `start`, `abort` and `commit`, which are called (either by the CA action support or by the CA action participants) in the appropriate points of 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. CA actions are intended for providing fault tolerance in concurrent object-oriented systems. This paradigm offers a conceptual framework within which actions have many of the typical properties of classes (allowing actions to be re-usable and extendable).

Our aim in this paper is to clarify this concept further, and to describe the practical Ada-based experiments we have been undertaking. Our purposes are to design a basic framework for investigating all details of CA actions and to offer practitioners clear conventions which would make it possible to use CA actions in Ada [1] applications.

1.3. Previous Research

A set of Ada atomic action schemes was described in the paper [18]: following ideas from [19], 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 presents a set of procedures which 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 well with the CA action concept because packages are units of system design and because Ada classes (tagged types) usually form packages [1]. The paper [15] offers a general framework for using atomic actions with forward error recovery in existing practical languages. This framework relies on a set of programming conventions and regards the exception contexts of atomic actions as a set of the 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 and handlers for the same exception are started when an exception has been raised in any of them. The paper [16] outlines general rules describing how atomic objects should be treated within distributed CA actions and offers a resolution algorithm intended for distributed object oriented systems with CA actions.

1.4. Synchronous and Asynchronous Schemes

We believe that it is important to distinguish between synchronous and asynchronous[1] atomic and CA action schemes [3, 15]. In synchronous schemes [10, 11, 15, 17], 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 it is ready to accept information about the state of other participants. Asynchronous schemes [16, 18] do not wait for this but use some means of interrupting all participants when one of them has found an error.

Recovery and exception resolution are much easier to provide within synchronous schemes than asynchronous ones because each process is ready for recovery and is in a consistent state when its handler is called. Moreover, there is no need to program the abortion of nested actions for these systems because they have to be completed. Obviously, there is a risk of deadlocks arising in these systems, but we believe that careful programming with intensive error detection should make it possible not just to avoid this problem but to simplify the subsequent recovery. Some additional programming rules can make synchronous schemes more efficient and decrease time waste (time-outs; assertions; checking invariants, pre- and post-conditions; see [17] for a detailed discussion). This can allow an early detection of either the error or the

---

[1] We use these terms only to differentiate between two ways of involving participants into action recovery. To put it in a different way, we are talking about synchronous and asynchronous exception handling.

abnormal behaviour of the process which has raised an exception and is waiting for the other processes.

No time is wasted in asynchronous schemes but the features required for the asynchronous transfer of control are not readily available in many languages (such as CSP, Ada83) and systems. Even when they are, they are usually very expensive: for example, many implementations of this feature in Ada will use the two thread model with the abortion and re-creation of one thread [2]. Moreover, they usually have complex semantics; it is more difficult to analyse, to understand and to prove programs which use these features. In addition, restrictions are often imposed on the program segment that can be interrupted asynchronously, in an attempt to make the implementation less expensive. For example, Ada tasks cannot accept messages within this segment. One more difficulty with asynchronous schemes is that the abortion of nested actions is difficult to program. In particular, even if application programmers implement such abortion handlers, a very sophisticated protocol (e.g. the one in [16]) needs to be applied to raise abortion exceptions in all nested actions (recursively and in the proper order); and, obviously, there are no languages which allow this. As for using atomic objects in asynchronous CA action schemes, the abort of the containing action should not be executed before the aborts of all nested actions. Dealing with these objects could be the responsibility of either the above-mentioned abortion protocol or application programmers (in which case abortion handlers have to call operations `abort` for all atomic objects used in the action).

Generally speaking, an appropriate CA action scheme should be chosen depending on the application peculiarities and requirements, on the errors to be detected, on the failure assumptions, etc.

2. Ada

In this Section we are going to briefly introduce some important new Ada features [1] to be used in our scheme.

Protected objects are new concurrency features similar to Hoare's monitors [8]. They are essentially data-oriented because some encapsulated data can be accessed only by calling object entries, functions and procedures. This access is restricted by a well-defined set of that which 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. Protected types can be used, in which case protected objects can be declared as their instances.

Exceptions in Ada are basically the same as in Ada83. But there is a new library package, `Ada.Exceptions`, which is very important for our scheme. Its function `Exception_Identity` 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 or re-raised by procedure `Reraise_Exception` from the same package. This package has a predefined `Null_Id` constant that does not represent any exception. All this makes it possible to collect all exception identities in one location, to resolve them and to raise the resolved exception in all CA action participants.

3. Our Objectives

In this paper we are going to describe an implementation of CA actions in standard Ada, to be presented as a set of programming conventions. This will be a synchronous scheme with forward error recovery based on concurrent exception resolution. What were the reasons for the choices we have made?

CA actions in a standard language. One of the main problems which researchers in fault tolerance face is that there is a big gap between conceptual language proposals and their use in real applications. The reason is that there is little chance that new languages which incorporate these features will actually be developed and used. One of the possible solutions is to employ sets of programming conventions within standard languages [13]. Moreover, this could make it possible to experiment with new conceptual proposals quickly. That is why we believe that this is an important way to make CA action research practical. Our intention is to map the general CA action concept [19] and the CA action exception model [16] onto Ada. In particular, we will describe what it means to have action exceptions, handlers on the action level, etc. In addition, we will rely on the general framework in the paper [15] for introducing atomic actions with exception resolution into standard languages. Our intention is to find a set of easy to follow conventions within the standard Ada and to rely on existing language features as much as possible.

Ada. Although there are several concurrent object oriented languages with exception handling (e.g. C++// [4] or Arche [9]), we have chosen Ada because it is the most popular standard language used in numerous complex applications with high dependability requirements.

Synchronous scheme. Our intention is to design a synchronous scheme since, as was show in our discussion above, these schemes have very important advantages, and there are many application domains in which their use is fully adequate. In particular, this is the case for applications designed in Ada.

Scheme with exception resolution. The papers [3, 15, 16] clearly show why exception resolution cannot be ignored for many applications and why it is better to provide it by action support rather than application software. The obvious reason is that several exceptions can be raised concurrently and the scheme should be able to deal with this situation. Besides, since error detection tools are never perfect, the latent period of an error is not negligible, and erroneous information can be easily spread within a CA action; thus, several errors occurring concurrently in different processes can be the symptoms of a more serious fault. Very often there is a correlation between errors, so they happen over a very short period of time in different action participants. Another reason is that in practice it is impossible to interrupt all participants the moment one of them has raised an exception. The probability of new exceptions being raised in other participants before they are informed about this exception is very high in distributed systems. In these systems, the overall hardware failure probability is very high, too. Another reason is that we believe that it would be dangerous not to use resolution for some applications because in this case some concurrent exceptions can be lost. Moreover, with all action participants having to be synchronised at the CA action end (this is the essence of all atomic action schemes), it seems natural and not too costly to extend this final synchronisation by passing information about the detected errors and by adding the resolution stage. That is why we believe that resolution should be used even for asynchronous schemes in which there is little chance of several exceptions being raised concurrently.

4. Framework for Using CA Actions

4.1. CA Actions as Packages

CA actions should correspond with the main structuring units of the language used. This is why in our scheme all participants of an action are collected in an Ada package in the form of interface procedures to be called by some external tasks (we adopted this idea from [18]). Tasks enter each action in this way. The important advantage of this approach is that the design units of concurrent systems are at the same time the dynamic units of system execution. Basically, with this approach concurrent systems can be designed using Ada packages. We believe that this is the most practical way of using CA actions in Ada because it allows CA actions to have distinct boundaries and because the visibility scope is the same for all action participants. Moreover, it is easy to attach recovery features to each participant when it is a package procedure.

Atomic action `A1` with three participants can have the following specification:

```
package A1 is
    procedure Participant_1(parameters);
    procedure Participant_2(parameters);
    procedure Participant_3(parameters);
    Failure_A1: exception;
end A1;
```

4.2 Exception Declaration

The declarations of all action exceptions are in the body of the action package, in the visibility scope of all participant procedures. Each participant has to have a handler for each action exception because any of their handlers can be called during the resolution (even of an exception which is never raised in a given participant). We strongly recommend here to follow the main ideas of [3]. In particular, we believe that a universal exception should be declared in each action. It is used if the action recovery fails, in which case the corresponding handlers do the clean up (final) operations and raise the failure exception in the containing action. To allow this, action A1 should have failure exception `Failure_A1` declared in the specification part. Besides, each action should have exception `No_Exception` for participants to signal a successful completion of their parts of the action (see Section 4.4). An example of declaring three exceptions `A`, `B` and `C` of action `A1` looks as follows:

```
A, B, C, Universal_Exception, No_Exception :
                              exception; -- action A1 exceptions
```

4.3 Exception Context. Handlers

Within our scheme, the exception context in each CA action participant is the procedure body, which is the conventional Ada exception context. Exception handlers are attached at the end of the context in accordance with Ada rules. Clause `others` can be freely used here to handle several exceptions. Each handler includes application code that provides cooperative action recovery and, in particular, action atomicity and data consistency. As has been pointed out before, the handlers of different action participants for the same exception are to be programmed cooperatively to recover the action state, i.e. the states of all of its participants, atomic and local objects, in a coordinated way.

To unify the treatment of all exceptions (predefined, propagated from nested procedure calls or nested CA actions and raised within the block) we introduce a new nested service block which catches all exceptions with clause `others`. Another important detail is that within our CA action scheme we have to synchronise all tasks, even those without raised exceptions. To do this, each task that is going to complete its execution successfully has to raise a special exception `No_Exception`. It should be declared in each action together with application-dependent exceptions (see Section 4.2). The only operation these handlers have to perform is to call entries of the synchronising and resolving object (SR object) and pass the exception identifier.

The template of the exception context for a participant of action `A1` looks as follows:

```
procedure Participant_1(parameters) is
-- ... declarations
begin -- start of the exception context
   begin -- block for predefined exceptions
     -- ... application code
     exception
         when No_Exception => SR.Finish; -- Id for No_Exception is default
         when E: others  => SR.Finish(Exception_Identity(E));
   end; -- end of the additional block
exception
     when A => -- ... application code
     when B => -- ... application code
     when C => -- ... application code
     when Universal_Exception => -- ... application code
         raise Failure_A1;
end Participant_1; -- end of the exception context
```

Application programmers have to deal with both local and atomic objects. Handlers decide cooperatively to call either `commit` or `abort` for atomic objects involved in the CA action. We recommend to call operation `commit` when the action is successfully completed (in particular, after a successful recovery executed by handlers) and call `abort` for all of these objects before the action failure exception is raised (from handler `Universal_Exception`). Operation `start` is to be called by an action participant when it enters the action (e.g. as the first operation of procedure `Participant_1`).

4.4. SR Object. Raising Exceptions

The execution of each action is managed by a service object. We design a protected Ada parameterised type `SR_object`, so that an instance of it is created for each CA action in the action package body. This object controls the execution of all action participants, and when all of them have either reached the action end or raised exceptions, this object resolves all exceptions and raises the same resolved exception in all participants. The SR object plays the role of the manager in the general CA action concept [19].

All action participants have to be synchronised while completing the execution of their exception contexts. Each of them has to complete the execution of the context either by raising an exception or in the normal way; in the latter case it has to raise the standard exception `No_Exception` . This is a very important feature. It allows each participant to inform the SR object that it has reached the end of the exception context and to wait until all of them have reached their context ends. The SR object decides what all participants should do and lets them proceed synchronously. In this way our scheme guarantees the properties of CA actions.

The parameterised protected type `SR_object` can be implemented as follows. It has entry `Finish`, which is to be called from the handlers of action participants. The identities of the raised exceptions are collected in list `Results`. Procedure `Resolution` takes this list and, using the resolution tree, finds the resolving exception, which is assigned to variable `Resolved`. Note that `Resolved` is equal to `Null_Id` when all participants call entry `Finish` from handlers `No_Exception`, in which case no exception is raised and the action completes successfully. `SR_object` can be used many times just as the same action can be executed many times.

```
protected type SR_object(Number : Positive) is
     entry Finish(E : in Exception_Id := Null_Id);
private
     entry Real_Raise(E : in Exception_Id := Null_Id);
     Finished : Integer :=0;
     Results : ... ; -- list of all exceptions raised
     Resolved : Exception_Id :=Null_Id;
     Let_Go: Boolean := False;
end SR_object;

protected body SR_object is
procedure Resolution is
-- ...
begin
     -- it takes Results and, using the resolution tree, finds
     -- the resolving exception, which is assigned
     -- to Resolved; if everything is ok Resolved is Null_Id
end Resolution;
entry Finish(E : in Exception_Id := Null_Id) when True is
begin
     Finished:=Finished+1;
     -- add E to Results
     if Finished=Number then
            Resolution;
            Let_Go:=True;
     end if;
     requeue Real_Raise;
end Finish;
```

```
    entry Real_Raise(E : in Exception_Id := Null_Id) when Let_Go is
    begin
         if Real_Raise'Count=0 then
                Let_Go:=False; Finished :=0;
         end if;
                Raise_Exception (Resolved);
    end Real_Raise;
  end SR_object;
```

The programming conventions are as follows. There should be a SR object in each action whose name is known to all participants. An instance of this object for action A1 is created using the protected type above as follows:

```
SR_A1 : SR_object(3);
```

When an action participant executes the exception context, it is only allowed to raise an exception of this action. The signalling of the failure exception can be done in handlers. Note that this can only happen after exceptions have been raised and resolved in the action. Within our scheme, exception re-raising (found in many exception schemes) is understood as raising the failure exception of the containing action, which is a uniform signal of the nested action failure [3, 6]. If all participants raise exception `No_Exception`, then no resolving exception is raised and the action is successfully completed. The universal exception can be raised either by a participant when it decides that the action should fail or by the SR object (when it is found to be the resolving exception).

4.5. Resolution Procedure

The resolution procedure calculates the resolved exception and raises it in such a way that it is propagated by the run time system to all action participants through entry calls. This happens because all participants are suspended on entry `Real_Raise` until the resolved exception is calculated.

To implement the initial idea [3] about the exception tree in a general way, the tree data structure and corresponding operations should be implemented. Many approaches can be used but Ada does not let `Exception_Id be` known without raising the exception (we need these identities to build the resolution tree before the action is executed). Thus, building the resolution tree should be done at the initialisation stage; for our example with action `A1`, it can look as follows:

```
-- find Id for each action exception:
begin  raise A;
    exception when Ex_Oc : others  => A_Id:= Exception_Identity(Ex_Oc);
end;
begin  raise B;
    exception when Ex_Oc : others  => B_Id:= Exception_Identity(Ex_Oc);
end;
begin  raise C;
    exception when Ex_Oc : others  => C_Id:= Exception_Identity(Ex_Oc);
end;
begin  raise Universal_Exception;
    exception when Ex_Oc :
         others  => Universal_Exception_Id:= Exception_Identity(Ex_Oc);
end;

-- adding new branch into the resolution tree:
```

```
New_Branch(Universal_Exception_Id, A_Id);
New_Branch(Universal_Exception_Id, B_Id);
New_Branch(A_Id, C_Id);
```

In this way we can build any tree for any action. The actual programming of procedures for adding branches and looking for the covering exception is a routine exercise. When the resolution procedure finds the resolving exception, it assigns its identity to variable `Resolved` and nullifies list `Results` in which the raised exceptions have been collected.

There are several other ways of implementing the resolution procedure. For example, we regard using the decision table as a barely adequate simple solution. To conclude the section, we would like to stress that using powerful Ada features together with our approach makes the implementation of resolution procedures a routine job. Procedures of any sort can be easily programmed: they can make it possible to use simple priorities of exceptions, resolution trees or even the most general approach mentioned in the paper [3], within which all action exceptions are partially ordered as a lattice.

4.6. Nested CA Actions. Exception Propagation. Failure Exceptions

Structuring a system as a set of nested actions is the main way of reducing the complexity of design [12]. Exception propagation allows us to complete the nested action execution and to raise an exception of the containing action. Within our approach, the propagation should be done by universal exception handlers by raising the failure exception in the containing action. These handlers coordinate their recovery and, if necessary, signal failure exceptions to the containing action. This allows participants to execute the 'last will' or 'clean-up' functions and to recover the nested action state (to guarantee the CA action atomicity and the consistency of external and local objects). Failure exceptions for all nested CA actions should be included into the resolution tree of the containing action and handlers should be programmed.

Packages declaring nested CA actions should be in the visibility scope of the package of the containing one: they can be either attached by the clause `with` or declared in the declaration part of this package. To provide proper action nesting and exception propagating, we rely on a set of programming conventions. Action participants should be implemented in accordance with the templates and rules discussed above. A subset of tasks from the containing action can take part in the nested ones. It is allowed for a task to be only in one nested action at a time. Our scheme requires an explicit exception propagation from the nested action to the containing one for each level of nestedness: in this respect handlers are parts (preludes) of the containing exception context. Though this is not the way conventional propagation works (for example, in Ada, C++ and Eiffel an exception is automatically propagated through any levels of the nestedness until the handler is found), we believe that this restriction is extremely sensible and useful in practice and, in particular, for CA actions. It makes programmers include 'clean up' operations in each action. Moreover, dealing with atomic and local objects and guaranteeing their properties should be programmed within these handlers. (One of the examples could be guaranteeing local object consistency by providing all-or-nothing semantics).

4.7. Simple Example

We believe that there are many complex applications which can be structured better because a new design level with abstractions describing complex coordinated concurrency can be introduced with the help of CA actions. Moreover, using this concept allows programmers to provide fault tolerance in a more disciplined way. Some of these applications are groupware, complex control systems or modelling human activities: games, plays. We will outline briefly a CA action `Railway_Station` which coordinates the execution of a complex activity concerned with the cooperation between several trains calling at a particular station. There are four participants, which are trains calling at the station at different platforms. When a train approaches the station, it enters the action by calling the corresponding procedure (depending on the platform it is going to make the stop at):

```
package Railway_Station is
    procedure Train_Platform_1(parameters);
    procedure Train_Platform_2(parameters);
    procedure Train_Platform_3(parameters);
    procedure Train_Platform_4(parameters);
    Failure_Railway_Station : exception;
end Railway_Station;
```

We assume that these trains execute some coordination activity within this action (when they stop at the station): they exchange mails and luggage; some carriages of one of them can go with other trains; passengers can change here. This application requires introducing a new private task serving as the station dispatcher. Trains use a central data base (designed as a set of transactional objects) to update their timetables (in case of delays), to know the timetables of approaching trains, to keep the information about the mail and luggage they carry. Local objects are intertrain messages, trolleys with luggage, mail bags, carriages, passengers, etc. Action exceptions can be declared as follows:

```
Train_Is_Not_Here, Train_Out_Of_Order, Carriage_Out_Of_Order,
    No_Space_In_Train, Out_of_Time, Approaching_InterCity,
    Universal_Exception, No_Exception : exception;
```

Some examples of recovery actions are: re-loading the trains (passengers, luggage), leaving some luggage at the station, calling for train or carriage repair or replacement.

5. Discussion

5.1. Scheme Extensions

The proposed scheme can be easily extended to allow participants to be checked when entering an action. We will briefly outline one possible solution. In practice, situations can arise when different sets of tasks participate in the same action; e.g. if two sets of tasks (`T1`, `T2`, `T3`) and (`T4`, `T5`, `T6`) try to participate in action `A1` introduced above, it could happen that tasks T1, T5 and T3 will be allowed to take part in this action unless something is done to prevent this. Moreover, our scheme allows several tasks to call the same action procedure (e.g. Participant_1). The extended scheme should allow only participants of the same set to be within an action at the same time, so it guarantees the mutual exclusion of executions of different sets of participants in the action. All participant procedures should have their first parameter represent the CA action name or identifier, which should be the same for all participants from the same set (they have to know it in advance). The SR object should have an additional entry Enter(Action_Id : Id_T), which is called in each package procedure as the first statement. This identifier is checked in the entry body and only participants with the same Id are allowed to proceed. All other calls are requeued to private entry Enter_Wait(Action_Id : Id_T), which has a barrier condition Let_Enter. When all participants of the current task set leave the action (entry Real_Raise of the SR object), this condition is assigned to True, so the re-queued calls from the participants of other sets can be accepted (e.g. by re-queuing them to entry Enter).

A more elaborate extension of the scheme above can allow the SR object to deal with atomic objects (call their start, commit and abort operations). To do this, protected type SR_object should be parameterised with the list of atomic objects. Another possible extension is guaranteeing the properties of local objects. An important feature of the SR object could be cooperation with the SR objects of the nested CA actions, which would be able to improve the support provided (e.g. by checking that only a subset of the containing action participants enters the nested action).

5.2. How Can Participants Cooperate?

The concept of a local object is very important for CA actions. In particular, within this implementation there is no way for action participants to interact directly by rendezvous (Ada entries cannot be declared in procedures). That is why the only way for action participants to exchange messages or synchronise their execution is by sharing local objects, declared in the action package. Moreover, our previous research [5] shows that it makes sense to have not just *shared local objects* but *private local objects* as well. Their implementation is much easier because each of them belongs to one of the participants. In this scheme we assumed that, with objects of either sort, application programmers are responsible for their state restoration, consistency and for passing them correctly to nested CA actions. Although this corresponds well with the concept of forward error recovery [12], we believe that additional support could make the programmer's job much easier and less error prone. First of all, local object consistency can be guaranteed if they are Ada protected objects. The next step could be implementing support to pass them to nested CA actions. The latter can be done in a disciplined way if these objects are extended with new service methods: create, save, discard, restore, distract. These methods are obviously application-dependent but their calling can be done by either the application code (from exception handlers) or the CA manager. The purpose is to leave these objects in a correct known state when the action is finished (e.g. all-or-nothing semantics can be implemented).

5.3. General Action Handlers

Traditionally exception handlers are attached to each action participant, and it is assumed that they cooperatively provide action recovery. But we believe that it would be important to introduce a new level of handlers declared on the action level (in the action package, for our scheme) [16]. They have to provide some general action recovery. In particular, their responsibilities could be dealing with local and atomic objects, coordinating the joint recovery of action participants, executing a simplified functionality (to provide degradation if the main algorithm fails), etc. Generally speaking, all 'local' handlers attached to participants should be completed before these action handlers start. The important thing is that packages are units in which action handlers can be declared. Although Ada does not allow handlers to be attached to packages, this can be implemented within the standard language. For example, it is possible to have a set of handler procedures declared in the action body, or they can be declared as conventional Ada handlers in the executable part of the action body. The action manager can be extended to call the appropriate handler/procedure for the resolved exception on the action level.

5.4. Comparison with Atomic Action Scheme in [18]

Any CA action scheme includes support for conventional atomic actions, and our approach can be compared in this respect with an Ada atomic action scheme intended for forward error recovery (Section 5.3 in [18]). As we have mentioned before, we have borrowed several ideas from this approach but there are many important differences. The asynchronous scheme in [18] ignores all but one of a set of concurrently raised exceptions; that is why there is no need in exception resolution here (which is, from our point of view, not quite correct: see Section 3). We believe that this scheme does not address the problem of aborting nested actions properly (see our discussion in Section 1.4). Our scheme is essentially synchronous, and we do not need features of this sort. Another difference between the two approaches is that in our scheme there is an individual handler for each action exception; whereas the scheme [18] uses one handler for all exceptions. Although it can find out the exception identity and execute some appropriate recovery, we believe that our approach provides a better structuring.

5.5. Object Orientation Issues

As we have mentioned above, there is another reason for choosing Ada packages as language units corresponding to CA actions. Object orientation in Ada is entirely based on tagged types which unify all procedures of a package (that use a parameter of this type) in one class [1]. This makes it possible to offer a more object-oriented CA action scheme. We can use here the ideas from [18] where a standard package for action support is introduced. This package declares an abstract action tagged type which includes the action manager. To create a concrete type for a particular action, this initial type has to be extended by including participant procedures.

More issues are involved here: dealing with local and atomic objects (in particular, we would like to treat them as the 'properties' or 'parameters' of the CA action and to provide a set of standard classes to make their implementation simpler), including exception resolution on the object-oriented level (with the overloading of resolution procedures and trees), the overloading of the action manager, preventing the information smuggling, the overloading of action exception handlers (if they are implemented as procedures in the action package, see Section 5.3), etc.

6. Conclusions and Future Research

Although our scheme is a simplification of the CA action concept, we believe that this research still has merit. First of all, we have discussed a complete CA action scheme which can be used in practice. In fact we regard it as a platform and framework which makes it possible to investigate further practical issues of CA actions using Ada. All the main responsibilities of the CA action support were discussed in [19], and in particular, it was proposed to use an action manager to implement this support. The SR object in our scheme is a particular implementation of these ideas. Its functionalities will be enriched in future research in many ways. It will allow the list of local objects to be declared and their state restoration provided when an action signals failure. Another important responsibility of the action manager is dealing with atomic objects. In our scheme these two are to be programmed by application programmers in the application code. This is obviously error prone. On the other hand, a faster and easier code can be designed by programmers who know the peculiarities of the applications; moreover, forward error recovery assumes that all recovery is application-specific [12]. Among other things, we wanted to discuss some basic practical framework for using CA actions to be able to start experimenting with them.

Acknowledgements. This research has been supported by DeVa ESPRIT project. We would like to thank A. Burns, S. Mitchell and A. Wellings (University of York), our colleagues in this project.

References.

[1] Ada. Information technology - Programming languages - Ada. Language and Standard libraries. ISO/IEC 8652:1995(E), Technical Report Intermetrics, Inc., 1995.

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

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

[4] D. Caromel, F. Belloncle and Y. Roudier, The C++// Language, in G. Wilson and P. Lu, eds., *Parallel Programming Using C++* (MIT Press, 1996)

[5] A. Coccoli, Implementing Local Objects in Ada95, MSc Thesis, University of Pisa, Italy, 1996 (in preparation).

[6] F. Cristian, Exception Handling and Tolerance of Software Faults, in M. Liu, ed., *Software Fault Tolerance* (J. Wiley, 1994) 81-107.

[7] J. Gray and A. Reuter, *Transaction Processing: concepts and techniques* (Morgan Kaufman Publishers, San Mateo, California, 1993).

[8] C. Hoare, Monitors - an operating system structuring concept, *Communication of ACM*. 17, 10 (1974) 549-557.

[9] V. Issarny, An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software, *J. of Object-Oriented Programming*. 6, 6 (1993) 29-40.

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

[11] K.H. Kim, Approaches to mechanization of the conversation scheme based on monitors, *IEEE Trans. on Soft. Eng*. SE-8, 3 (1982) 189-197.

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

[13] B. Randell, Approaches to Software Fault Tolerance, *Proc. 25th Annual LAAS Conference*. Toulouse, France (1993) 33-42.

[14] B. Randell, System structure for software fault tolerance, *IEEE Trans. on Soft. Eng*. SE-1, 2 (1975) 220-232.

[15] A. Romanovsky, Atomic Actions Based on Distributed/Concurrent Exception Resolution, Technical Report, Computing Dept. University of Newcastle upon Tyne, 1996 (in publication).

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

[17] A.B. Romanovsky, Practical exception handling and resolution in concurrent programs, Technical Report 545, Computing Dept. University of Newcastle upon Tyne, 1996.

[18] A.J. Wellings and A. Burns, Implementing Atomic Actions in Ada95, Technical Report YCS-263, Department of Computer Science, University of York, 1996.

[19] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud and Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, *Proc. Twenty Fifth Int. Symp. on Fault-Tolerant Computing*. Pasadena, USA (1995) 499-508.