On Programming Atomic Actions in Ada 95

A. Romanovsky, S.E. Mitchell and A.J. Wellings
University of Newcastle upon Tyne, UK University of York, UK

Abstract. This paper describes the development of two kinds of atomic action schemes for Ada 95. We start by discussing the basic features required of an atomic action scheme and what choices, e.g. between synchronous and asynchronous actions, are appropriate for Ada 95. We then present two implementations of actions; first using Ada 95 packages to create asynchronous actions and secondly, as sets of tasks for synchronous actions. For each action type, we present code fragments illustrating their development and use. Finally, we discuss some related issues (exception resolution, action nesting, state restoration, software re-use and extension, preventing information smuggling, distributed execution) which have been addressed in our work and show some of the problems encountered (the deserter problem, using different sorts of inter-participant communications and resources).

1 Introduction

1.1 The Basic Requirements of Atomic Actions

Atomic actions (AAs) can be used as a mechanism for structuring complex concurrent systems and for achieving system fault tolerance. The first proposal in this direction was the conversation concept [8], which was later extended to the atomic action concept [7]. Subsequently, important steps were made in [2], where general rules for building atomic actions were outlined. The AA characteristics are summarised in [1]:

- Well-defined boundaries: each action has a start, end and a side boundary.
- Indivisibility: no information exchange between participants and the outside world.
- Nesting: AAs can be nested but they may not overlap. The permissible overlapping and nesting situations are illustrated in Figure 1.
- Concurrency - it should be possible to execute different AAs concurrently.
- Recovery: AAs form the basis of damage confinement within a system and allow recovery procedures to be programmed.

The most important characteristic of actions is the handling of errors. If an error occurs within an action, all participants take part in the action recovery, and if the participants are not able to recover then the action is completed, marked as 'failed', and recovery initiated in the containing action. When several errors (exceptions) are concurrently raised in an action, an *exception resolution scheme* is required which uses a resolution procedure to resolve concurrent exceptions, find the *generalized exception*, and call the appropriate handler in all action participants. The resolution process may use the concept of an exception tree (which is often more appropriate than exception priorities) to resolve exceptions. The tree includes all exceptions associated with the action and imposes a partial ordering on them in such a way that the handler for a higher exception is capable of handling any lower level exception. Abortion handlers for nested actions may be needed for some applications and backward and/or forward error recovery can be used.

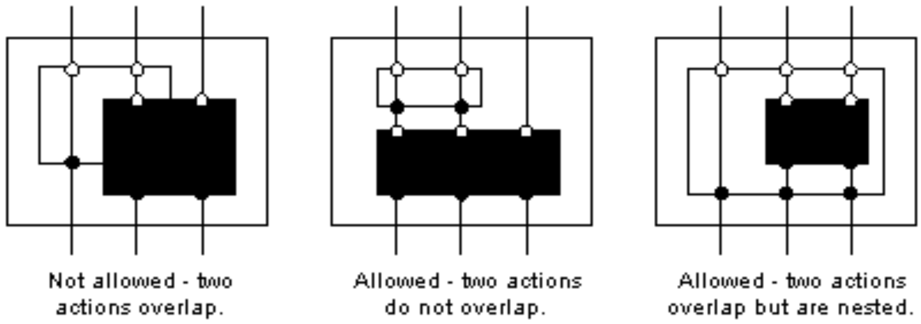

Fig. 1. Illustration of allowed nesting of atomic actions

1.2 Appropriate Atomic Action Schemes for Ada 95

One of the problems faced by researchers in fault tolerance is the big gap between conceptual language proposals and their use in real applications. This gap occurs because there is little chance that new languages which incorporate the features proposed actually being developed and used, and a possible solution is therefore to employ sets of programming conventions within standard languages. Moreover, this makes it possible to experiment with new conceptual proposals quickly.

Ada 95 [5] has no language level support for AAs, but initial work has shown that it is powerful enough to allow the AA paradigm to be implemented [1, 12, 13]. The aims of our current work (within the context of the DeVa Project [3]) are:

- To produce a set of carefully chosen and checked rules/conventions specifying how to design Ada applications using AA concepts, and how to structure applications from atomic actions,
- To maintain the separation of concerns between system layering, on the one hand, and hiding system software from application programmers who use it to design systems, on the other hand. This is achieved by hiding (as many as possible) implementation details from the application programmer,
- To make the approach as object-oriented as possible.

The intention here is to make the application programmer's job easier (and so less error prone), which has led to the following goals for the research described in this paper:

- To maximise the re-use of actions and to hide implementation structure by embedding as much as possible of control structure within a support package with a well-defined interface,
- To make "client side" action packages (i.e. action users) as simple as possible,
- To allow the extension of an action via inheritance.

Several AA schemes have been proposed for Ada 83 (e.g. [4, 12]). Our analysis shows that the new features introduced in Ada 95 ease the implementation and use of AAs. The use of object-oriented programming makes it possible to design reusable and extendable AA schemes; the improved concurrency features, namely protected objects and asynchronous transfer of control (ATC), simplify action control (in Ada 83 it was impossible to design an asynchronous scheme); and finally, the use of the Distributed Annex [5] allows distributed AA schemes to implement.

1.3 Using Atomic Actions for System Structuring

Various atomic action schemes have been designed (see [9] for a comprehensive survey). They are intended for different applications and languages, differ in how participants enter actions and exchange information, allow backward and/or forward error recovery, etc.

There are two potential approaches to designing systems consisting of actions:

- Actions have internal tasks (e.g. concurrency is hidden from the outside world in [12], where tasks are forked when an action starts and jointed when it finishes) and the system is executed by processing actions in some order.

- The system consists of a set of tasks which participate in actions. This approach can be realised in several ways, for example, in [10] the synchronisation agents required to implement AAs are encapsulated in a package and a procedural interface is provided to end/recover actions. The scheme described in Section 2 extends this approach by encapsulating the user code, too.

We believe that the second approach with actions programmed/represented as tagged abstract data types (as in [1, 13]) provides for better encapsulation, in much the same way as a monitor provides better encapsulation for mutual exclusion than a semaphore, and should therefore be preferred. Such an approach also permits the development of libraries of atomic actions, the extension of actions through inheritance, re-use, etc.

1.4 Synchronous and Asynchronous Schemes in Ada 95

As well as there being two different mechanisms for structuring a system with actions, one can classify atomic actions into two distinct kinds depending on how action participants are involved in recovery:

- *Synchronous* - Each participant in an action has to either rendezvous at the action end or to find an error and inform other participants of the exception; it is only after each participant has completed that it enters a phase where it is ready to accept information about the state of other participants.

- *Asynchronous* - In contrast to synchronous schemes, asynchronous approaches do not use a “wait” but instead use some language feature to interrupt all participants when one of them has found an error. Consequently there is no ‘wasted time’ while participants that have found an error wait for others to complete their computation before the action can itself be completed.

Error recovery and exception resolution are much easier to provide in synchronous systems because each participant is in a consistent state, ready for recovery, when handlers are called. As a consequence, there is no need to program the termination of nested actions because they either will have completed successfully or have had any errors dealt with by the nested action’s handler. Obviously, there is a risk that deadlock can arise in these systems, but we believe that cautious programming with an intensive error detection makes it possible to avoid this problem and simplify subsequent recovery. Some additional programming rules can increase the efficiency of synchronous schemes and decrease the amount of time wasted (e.g. time-outs; assertions;

checking invariants, pre- and post-conditions; etc.). This can allow an early detection of either the error or the abnormal behaviour of the participant which has raised an exception and is waiting for the other participants.

Asynchronous schemes avoid many of the above problems but have been little used because the required language feature (e.g. ATC) is not readily available in many languages and systems. Even when available, the costs can be very high; for example, many implementations of ATC in Ada 95 use the two thread model with the termination and re-creation of a thread [1] resulting in high overheads. Moreover, they usually have complex semantics, are more difficult to analyse and understand, and it is harder to prove programs which use these features. To try and make the implementation less expensive, restrictions are often imposed on the facilities that can be used in a program segment that can be interrupted asynchronously. Finally, programming nested action abortion is a difficult problem because, first, this action is supposed to be indivisible and invisible for the containing action and, secondly, it can have its own faults and nested actions.

Which approach is appropriate depends largely on the application, on the errors which are to be detected, on the failure assumptions, etc. Thus any generalised scheme should allow programmers to choose the most suitable approach.

2 Atomic Actions as Packages

[1, 13] described the development of a set of asynchronous AA schemes, and ongoing work at both the Universities of Newcastle and York has continued to improve on this work. Within these schemes, actions are Ada packages and participants enter them by calling interface procedures. Actions can use both forward and backward error recovery. This section presents a brief introduction to the current state of the implementation.

The execution of the AA is controlled by a protected object `Action_Controller` that uses ATC to interrupt the execution of participants in the event of an error. A tagged type `Action_T` includes the controller as a component. Derived packages can be written which will implement atomic actions for particular systems. In these packages all action participants are represented as procedures called by external tasks. Actions can also be parameterised on the number of participants. The Ada code for the protected object is encapsulated within an `Atomic_Support` package:

```
with Ada.Exceptions; use Ada.Exceptions;
package Atomic_Action_Support is
  type Action_T(At_Least : Positive) is tagged
                              limited  private;
  type Vote_T is (Commit, Aborted);
  generic
    with procedure Work;
    with function Error_Handler(E: Exception_Id)
                                           return Vote_T;
    procedure Action_Component(A: access Action_T'Class);

  Atomic_Action_Failure : exception;
private
  protected type Action_Controller(At_Least : Positive) is
    entry Wait_Abort(E: out Exception_Id);
    entry Done(Vote: Vote_T; Result: out Vote_T);
```

```
    procedure Signal_Abort(E: Exception_Id);
  private
    entry Entered(E: out Exception_Id);
    entry Wait(Vote: Vote_T; Result: out Vote_T);
    entry Done_Cleanup(Vote: Vote_T; Result: out Vote_T);
    -- ...
 end Action_Controller;

 type Action_T(At_Least : Positive) is tagged limited
   record
     C : Action_Controller(At_Least);
   end record;
end Atomic_Action_Support;
```

The body of this package which controls the execution of all action participants wrapped into a standard template in procedure `Action_Component` is as follows:

```
1  package body Atomic_Action_Support is
2    protected body Action_Controller is separate;
3
4    procedure Action_Component(W: Work; EH: Error_Handler;
5                                A: access Action_T'Class) is
6      X : Exception_Id;
7      Decision : Vote_t;
8    begin
9      select
10       A.C.Wait_Abort(X);
11       Raise_Exception(X);
12     then abort
13       begin
14         W;
15         A.C.Done(Commit, Decision);
16       exception
17         when E: others =>
18           A.C.Signal_Abort(Exception_Identity(E));
19       end;
20     end select;
21   exception
22     when E: others =>
23       Decision := EH(Exception_Identity(E));
24       A.C.Done_Cleanup(Decision, Decision);
25       if Decision = Aborted then
26         raise Atomic_Action_Failure;
27       end if;
28   end Action_Component;
29 end Atomic_Action_Support;
```

In line 9, the `select` statement signifies the start of the ATC block. At this point, the run-time system detects that it must wait on the event specified in line 10 (an entry in the `Action_Controller`) but in the meantime begins to execute from line 14. If the event is triggered (by an exception occurring in another component) before the `then abort` branch completes then execution is halted and line 11 executed. If an exception is raised by procedure `W` (i.e. the procedure encapsulating the work for the action participant) then the exception handler in line 16 is invoked which uses an entry on the `Action_Controller` to signal all components that an error has occurred which will cause ATC to propagate to each component. The exception handler in lines 21-27 is used after the completion of the component to decide whether or not to commit or abort the action.

When an action is to be used within a program it is declared within a package which uses the support described above. An example specification of an atomic action `Action` with two participants is given below:

```
1  with Atomic_Action_Support; use Atomic_Action_Support;
2  package Action is
3    type My_Action_T is tagged limited private;
4    procedure Participant_1(A: access My_Action_T);
5    procedure Participant_2(A: access My_Action_T);
6    Action_Failure: exception;
7  private
8    type My_Action_T is tagged record
9      C: Action_T(2);
10   end;
11 end Action;
```

Lines 3 and 8-10 create a new private type, `My_Action_T`, derived from an existing type supplied from package `Atomic_Action_Support` and constrained to contain two components. Since this type is private, its internals cannot be accessed from outside the package `Action`; also, because the unconstrained base type is declared as abstract in package `Atomic_Action_Support`, one must first derive a new type before an instance can be declared. The package also declares a new exception (line 6) which is used to propagate failure by the support package - this relives the client of having to “`with`” the support package. The package body is declared as follows:

```
with Ada.Exceptions; use Ada.Exceptions;
package body Action is

procedure Participant_1(A: access My_Action_T) is
  procedure My_Work is
  begin
    -- perform work ...;
  end My_Work;

  function My_Error_Handler(E: Exception_Id) return
                                                  Vote_T is
  begin
    -- handle error ...
    return Commit;
  end My_Error_Handler;

  procedure A1 is new Action_Component(My_Work,
                                       My_Error_Handler);
  begin
    A1(A.C'Access);
  exception
    when Atomic_Action_Failure => raise Action_Failure;
end Participant_1;

procedure Participant_2(A: access My_Action_T) is
  -- as before for Participant_1
end Action;
```

The new My_Action_T type can be subsequently used by clients as follows:

```
with Action; use Action;

A1 : My_Action_T;

task Client_1;
task Client_2;

task body Client_1 is
  -- ...
  Participant_1(A1'Access);
exception
  when Action_Failure =>  -- ...
end;

task body Client_2 is
  -- ...
  Participant_2(A1'Access);
  -- ...
end;
```

At first sight it would seem simpler to declare action A1 in global scope and to incorporate the code for components directly in the tasks. However, the use of a package promotes the reuse of the components as well as prevents possible *information smuggling* [6] - the escape of the internal details of the action prior to the commit/abort decision.

Further details of this scheme and more examples of its use can be found in [13].

3 Atomic Actions as Sets of Tasks

Two synchronous schemes which use forward error recovery with concurrent exception resolution have been designed [10, 11]. In the first, one of the participating tasks (the head process) synchronises the rest of them at the action exit. When all tasks have reached it, this process either resolves the exceptions raised and propagates the resolved exception via nested rendezvous, or lets the tasks leave the action. The second scheme [10] uses a “resolving” protected object to synchronise the participant exits, to resolve all exceptions and to raise the resolved exception in all participants. The parameterised protected type SR_Object can be implemented as follows. It has two entries Finish (called from each action participant) and Wait_All (which is private). The identities of the raised exception(s) are collected in a list kept by SR_Object. The procedure Resolution uses this list and a resolution tree to find the resolved exception which is assigned to variable Resolved. Note that if Resolved is equal to Null_Id (all participants have raised exception No_Exception), no exception is raised and the action completes successfully. An important detail is that an additional exception, No_Exception, should be declared and raised when a participant completes the action successfully. This scheme treats predefined exceptions and the programmer's exceptions in the same way. An instance of type SR_Object is created for each action. The specification of this type is as follows:

```
protected type SR_Object(Participants_Number: Positive) is
  entry Finish(E: in Exception_Id := Null_Id);
private
```

```
   entry Wait_All(E: in Exception_Id := Null_Id);
   procedure Resolution;
   Finished : Integer :=0;
   Results : Results_T; -- list of all exceptions raised
   Resolved : Exception_Id := Null_Id;
   Let_Go: Boolean := False;
end SR_Object;
```

The `Resolution` procedure is called from the body of `Finish` when all participants have completed execution, and the resolved exception is then propagated to all of them. The body of this object is much simpler than the body of the action controller intended for the asynchronous scheme:

```
protected body SR_Object is
  procedure Resolution is ...;

  entry Finish(E: in Exception_Id := Null_Id) when True is
  begin
    Finished:=Finished+1;
    -- add E to Results
    if Finished = Participants_Number then
      Resolution; Let_Go:=True;
    end if;
    requeue Wait_All;
  end Finish;

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

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. Within this scheme, re-raising exceptions (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.

4 Comparison

Both kinds of schemes have a number of common features, the most significant one is that all of them are centralised in the sense that there is a single action controller [1, 13], a resolving and synchronising object [10] or a resolving head process [11]. This limits their scope for distribution and consequent ability to cope with network failures. However, one can distribute actions so that each node within an action is assigned a nested action which restricts the load placed on central services. Another realistic approach is to replicate the partition containing the action controller.

A synchronous scheme, as presented here, has the disadvantage that it is not extendable since the actions are not Ada 95 packages with internal tagged type. However, in a multi-processing environment, asynchronous schemes also suffer because of the lack of exception resolution leaving them susceptible to an incorrect handling of concurrently raised exceptions.

The analysis of asynchronous schemes shows that it is impossible not to lose some of the exceptions which are raised concurrently when Ada 95's ATC is used as the only means to implement an asynchronous scheme.

The two kinds of schemes are complementary since they are suited to different application areas. For example, an asynchronous scheme such as in [13] would be better for real time systems whereas distributed systems would require a scheme with exception resolution to handle exceptions raised concurrently.

5 Related Issues

5.1 Information Smuggling

Information smuggling was first defined in [6] for the conversation scheme [8] and occurs when a task participating in a conversation either obtains information from or leaks information to a task not participating in the conversation. Generally speaking, information smuggling is difficult to prevent without explicit language support. One approach we hope to explore for Ada 95 is a set of rules which could be defined to enforce the required restriction on the components operating as AAs. For example, if action components are encapsulated in a package then only pure packages may be "`with`'ed" in the package. Also, only access parameters pointing to constants should be allowed in the parameters of these package procedures.

5.2 Use of an Asynchronous Scheme: Type Extension

A very important characteristic of the schemes in [13] is extendability, for which Ada 95 offers a comprehensive set of mechanisms. We have explored ways of extending the `Action_T` type. Generally speaking, a derived type `My_Action_T` can be created from the parent type by:

- Combining any new participant procedures with the parent package or overriding the parent's participant procedures,
- Adding new components (which may be action states, say, local objects) for new and overridden participants to work with,
- Making the number of participants concrete.

A chain of derivations can be used to extend, step by step, the original `Action_T` type. It is important to note that:

- Participant procedures can be kept 'abstract' (virtual) in this inheritance chain as long as is required (but obviously the type remains abstract and so cannot be used to create objects); the same is true for the `Action_T` type itself.
- The number of participants is a parameter of all types derived from `Action_T`, so, it is just of type `Positive` (no restriction) in the basic `Action_T` type. It can be either made concrete in the derived action type with a known number of participants or left unchanged.

Ada 95 imposes minor restrictions on this:

- If the parameter (number of participants) is made concrete it cannot be changed in the derived types;

- One cannot rename the parent participant procedure without overriding it with the same name and renaming it (in the same type definition);

- One has to know how many participants are in the action to make it non-abstract (this can be known only if all elements of the derivation chain are known, which is basically an engineering issue).

5.3 Nested Actions

Only properly nested atomic actions are allowed. Nested actions are programmed by introducing an object of the action to be nested in the body of the nesting action. Consider, for example, a new action which wishes to use package `Action` as part of its implementation. Suppose this action has three participants; i.e.

```
with Atomic_Action_Support; use Atomic_Action_Support;
package New_Action is
   type My_Action_T is tagged  limited private;
  procedure My_Action_1(A: access My_Action_T);
  procedure My_Action_2(A: access My_Action_T);
  procedure My_Action_3(A: access My_Action_T);
  Action_Failure : exception;
private
  type My_Action_T is tagged limited
  record
    aliased Action_T(3);
  end record;
end New_Action;
```

Now suppose that `My_Action_1` and `My_Action_2` use the `Action` package as part of its implementation. The body of this package would be:

```
with Ada.Exceptions; use Ada.Exceptions;
with Action; use Action;
package body New_Action is
  Nested_Action: aliased Action.My_Action_T;
  procedure My_Action_1(A: access My_Action_T) is
  procedure My_Work is
    begin
      -- ...
      Action.My_Action_1(Nested_Action'access);
      -- ...
    exception
      when Atomic_Action_Failure =>
        -- nested action has failed
        raise;
  end My_Work;

  function My_Error_Handler(E: Exception_Id) return
                                            Vote_T is ...;

  procedure My_Action_1 is new Action_Component(My_Work,
                                       My_Error_Handler);
  begin
    A1(A.C'access);
```

```
   end My_Action_1;
   -- similarly for My_Action_2
   -- My_Action_3 does not use the Nested_Action
end New_Action;
```

With the synchronous model, no further consideration is necessary as no attempt is made by `My_Action_3` to abort its action until all participants are ready. However, in the asynchronous model, extra facilities are needed to ensure that the nested action is not aborted without error recovery being undertaken [13].

To facilitate recovery in the inner action, it is necessary to use another Ada 95 facility called Controlled types. Objects of a controlled type can have (amongst other things) finalisation routines defined. Hence, each `Action_Component` procedure has the following extra components:

```
type Abort_Recovery(N: access Action_T) is new
        Finalization.Limited_Controlled with null record;
procedure Finalize(Ar: in out Abort_Recovery);
```

Here the finalisation action is to signal to the action controller that the action is to be aborted. A variable of `Abort_Recovery` is now introduced into the `Action_Component`:

```
procedure Action_Component(A: access Action_T'Class) is
  Ar : Abort_Recovery(A);
   -- ...
begin
   -- ...
```

When the action is aborted, the `Ar` controlled variable goes out of scope. However, before this can happen, the finalisation procedure is called. Note that some small modifications to the controller type are required as the `Finalize` routine is called *every* time the variable goes out of scope irrespective of whether the action was aborted or not!

6 Discussion and Future Work

In [4] the authors discussed why Ada 83 was not sufficient for programming AAs. Whilst we agree, to a limited degree, with this view (thus, neither Ada 83 nor Ada 95 support the prevention of information smuggling), we believe that one should not wait for the standard language to include AAs. Moreover, although Ada 95 does not solve all the problems mentioned, it does allow us to program actions in a simple and re-usable way whilst hiding the implementation details in the support units.

A potential problem with both synchronous and asynchronous schemes is their susceptibility to failure if a participant stops and thus never arrives at the end of the action. One possible approach to avoiding this is to implement a timeout (using either ATC or timed entry calls) based on the worst case execution time of each participant and thus other (waiting) participants or an action controller can detect the situation when a failure has occurred.

Further research will be required to address the problems mentioned in the paper as well as a number of outstanding issues. State restoration features for different kinds of inter-participant communication and of local data will need be designed to support backward error recovery and provide the atomicity semantics. More investigation and experiment is necessary to provide distributed atomic actions, though a general outline of the schemes was given in [10, 13]. Also, research is required to help programmers to solve (or to avoid) the deserter process problem [6].

We believe that a synchronous scheme could be programmed on the basis of the schemes described (with `SR_Object` playing the role of the action controller): actions will be presented as tagged types with methods playing the roles of action participants, which would allow for extendability, reuse, encapsulation and exception resolution.

This paper has presented two practical and usable kinds of AA schemes written in Ada 95. With these schemes, it is possible to create systems of actions with a minimal application programmer code. The use of Ada 95 features has improved confidence in the non-interference firewalls between actions and also promoted extendability of actions through type extension. The work is still in development but nevertheless forms a solid and usable implementation of actions in Ada 95.

Acknowledgements. This research is supported by DeVa, ESPRIT Basic Project No. 20072. We would especially like to thank the members of the DeVa project who have commented on initial versions of this paper - A. Burns at York University, J.-C. Fabre, T. Pérennou, D. Powell and Y. Crouzet at LAAS and R. Stroud at Newcastle University.

References

[1] A. Burns and A. Wellings, Ada95: An Effective Concurrent Programming Language, in: Proc. Ada Euro 96, Switzerland, 1996, 58-77.

[2] R.H. Campbell and B. Randell, Error Recovery in Asynchronous Systems, IEEE Transactions on Software Engineering SE-12, 8 (1986) 811-826.

[3] DeVa Project Programme and Description, Design for Validation, Proposal Document (http://www.newcastle.research.ec.org/deva/index.html, 1995).

[4] S.T. Gregory and J.C. Knight, On the provision of Backward Error Recovery in production programming languages, in: Proc. FTCS-19, USA, 1989, 506-511.

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

[6] K.H. Kim, Approaches to Mechanization of the Conversation Scheme Based on Monitors, IEEE Transactions on Software Engineering SE-8, 3 (1982) 189-197.

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

[8] B. Randell, System Structure for Software Fault Tolerance, IEEE Transactions on Software Engineering SE-1, 2 (1975) 220-232.

[9] B. Randell and J. Xu, The Evolution of the Recovery Block Concept, in M.R. Lyu, eds., *Software Fault Tolerance* (J. Wiley, 1994) 1-22.

[10] A. Romanovsky, Atomic actions based on distributed/concurrent exception resolution, TR 560, Comp. Dept., University of Newcastle upon Tyne, 1996.

[11] A. Romanovsky, Practical exception handling and resolution in concurrent programs, TR 545, Comp. Dept., University of Newcastle upon Tyne, 1996.

[12] A. Romanovsky and L. Strigini, Backward error recovery via conversations in Ada, Software Engineering Journal 10, 8 (1995) 219-232.

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