Practical exception handling and resolution in concurrent programs

Alexander Romanovsky

Applied Mathematics Department, State Technical University, 195251,
St. Petersburg, Russia
tel: +7 812 555 64 52, email: root@arom.hop.stu.neva.ru, fax: +7 812 552 60 86

The paper discusses how atomic actions based on forward error recovery in the form of concurrent exception handling and resolution can be programmed within standard conventional languages (Ada and Ada95). We express the main characteristics of the general atomic action scheme in terms of these languages and discuss a set of templates (skeletons) and programmers' conventions which would allow to program atomic actions within Ada and Ada95. We offer an approach to implementing a resolution procedure (function) and outline other approaches. The scheme is very flexible in that it gives an opportunity for programmers to use any sort of the resolution procedure. We introduce a general concept of self-checking programming, which allows to have the kind of failure assumption necessary for simplifying the atomic action support, and discuss how it can be applied (to Ada, in particular). It is shown how this approach helps to solve the deserter process problem. We outline the main improvements which can be made in the scheme when Ada95 is used. Naturally, our scheme relies on the peculiarities of Ada and Ada95. We believe that this paper discusses important practical questions because it seems unlikely that an existing practical language will have concurrent exception handling of the level sufficient for supporting atomic actions based on forward error recovery.

Keywords: software fault tolerance, exception handling, concurrent programming, atomic actions, exception resolution, Ada, Ada95, deserter process, multiway rendezvous

1. Atomic actions with forward error recovery

Whereas hardware fault tolerance techniques rely on component redundancy for tolerating physical degradation in hardware, tolerating residual design faults in software is based on design redundancy [1]. Fault-tolerant software detects errors caused by faults and employs error recovery techniques to restore normal computation. Forward error recovery (mostly, exception handling schemes) is based on the use of redundant data and algorithms that repair the system by analysing the detected error and putting the system into a correct state. In contrast, backward error recovery returns the system to a previous error-free state without requiring any knowledge of the errors.

The exception mechanism is a language control structure allowing programmers to describe the replacement of the standard program execution by an exceptional execution when an occurrence of exception (i.e. anything inconsistent with the program specification) is detected (see [2] for a rigorous and thorough discussion). This mechanism is usually considered an essential part of any modern language (Ada95 [3], C++, Eiffel [4], etc.).

For any exception mechanism, *exception contexts* [5], i.e. regions in which the same exceptions are treated in the same way, have to be declared. Very often they are blocks and procedure bodies. Each context has a set of associated exception handlers, one of which is called when a corresponding exception is raised.

There are different exception models. The termination exception model assumes that when an exception is raised, the corresponding handler completes the block execution. It is widely accepted that this model is more suitable for fault tolerant programming [1, 5]: it is adhered to in Ada, Ada95, C++, Eiffel, etc. The resumption model assumes that the handler recovers the program state, and the program continues the execution from the operation following the one which raised the exception. If the handler for the raised exception does not exist in the context or is not able to recover the program, the exception is propagated. The exception propagation goes through a chain of procedure calls or of nested blocks. The appropriate handler is sought in the exception context containing the context in which the handler was not found or was not able to recover the program.

Exceptions are much more difficult to handle and fault tolerance to provide in concurrent systems. The general approach to using exception handling in concurrent systems in [3] extends the well-known atomic action paradigm [1]. Atomic actions offer a general and sound basis for building fault tolerance schemes which allow processes to cooperate during recovery. [5] described the main rules of cooperative recovery: involving all participating processes if any of them detects an error, and calling the features intended for recovery after the same error in all participants.

For forward recovery, the units of cooperative recovery, i.e. atomic actions, are formed as a set of cooperative processes (tasks), each of which participates in the action while executing its corresponding exception context. A set of exceptions is associated with each action. Each process participating in an action has a set of handlers for (all or part of) these exceptions. Participants enter the action by entering the exception contexts with associated handlers. They enter the action asynchronously but have to leave it synchronously to guarantee that no information is smuggled. Besides, this allows to guarantee the action atomicity and recoverability. If an exception has been raised in a process, all action participants have to participate in the recovery, and the handlers for the same exception have to be called in all of them [5]. These handlers cooperate to recover the action. The participants can leave the action in three ways. First of all, this happens if there have been no exceptions raised. Secondly, if an exception had been raised, and the called handlers recovered the action. Thirdly, they can signal a *failure exception* to the containing action if an exception has been raised and it has been found that there are no appropriate handlers or that recovery is not possible.

A mechanism for *exception resolution* is the essential part of concurrent exception handling since several independent exceptions can be raised at the same time, or several errors are detected which could be the symptoms of a different, more serious fault. [5] offered a solution which relies on using a *resolution procedure*: this resolves all concurrent exceptions and works out a generalized exception, the handlers for which will be called in all action participants. In these cases, the concept of the *exception tree* [5] is more appropriate than exception priorities for resolving these exceptions. The exception tree includes all exceptions associated with the action and imposes a partial order on them in such a way that a higher exception has a handler capable of handling any lower level exception.

Of concurrent languages (Arche [6], Ada95, Ada, Modula-3 [7], SR [8], Real-Time Euclid [9], etc.) having exception handling features, only Ada and Ada95 allow handlers to be called in several concurrent tasks when an exception has been raised in one of them. These languages have a restrictive form of concurrent-specific exception propagation: an exception is propagated to both the calling and called tasks if it is raised during the rendezvous. The concurrent object-oriented language Arche allows to resolve the failure exceptions signalled by several implementations of the same class; however, it cannot be used for coordinated recovery of concurrent cooperating objects.

The *deserter process* problem [10] can arise during the execution of an atomic action. This execution can be delayed for an unpredictable time because a participant (the deserter process) fails to enter the action or reach its end. In these situations other participants have to wait for it because they have to synchronize their executions, to cooperate for the recovery and to leave the action together.

2. Discussion of the existing schemes. Our intentions

2.1. Existing schemes

The paper [11] offers a general scheme for using an extended CSP to implement atomic actions with both forward and backward error recovery. The details of implementing cooperative forward recovery on the basis of the traditional one-process exception handling are discussed.

To guarantee a synchronous exit together with the exception resolution, the authors [11] propose statically connecting all action participants in a virtual chain and synchronizing them by rendezvous through this chain when they reach the end of the action (and, in particular, when a local exception is raised). This allows each process to receive the information about the exception from the 'left neighbour' process, to partially resolve the exceptions and to transmit the resolution result to 'right neighbour' process. At the second step of this chain algorithm the last process in the chain finally resolves the exception and transmits the result to the 'left neighbour' process. This wave goes back to the left along the chain, and each process calls the appropriate handler for the same exception. We find this algorithm rather complex, though it can be simplified for the Ada programs. Besides, it seems that this investigation lacks a discussion of some important details (in particular, failure exception propagation) and has several disadvantages, e.g. it does not solve the deserter process problem. We believe that though this scheme suggests some good ideas, it cannot be directly applied in practice, primarily because CSP is an experimental language. Moreover, the authors had to extend it by time-outs, exceptions and a broadcaster process.

Some preliminary but very important steps were done for Ada in [12]. The authors outlined a technique for using atomic actions with coordinated concurrent exception handling. We believe that this is a very promising approach that makes atomic actions practical. Still, the authors do not discuss the complete atomic action scheme or outline all peculiarities which should be discussed to allow the scheme to be used right now. Besides, we believe that this scheme is rather expensive and can be essentially improved.

We will compare our scheme with both of these in detail in Section 5.6.

2.2. Our intentions

We would like to take the best of these two approaches and develop an atomic action scheme with forward recovery which could be used for Ada and Ada95 (as the only practical concurrent languages). We do not want to introduce a new language construct for atomic action declaration: this would make the approach not feasible for existing languages. We will rely on the general ideas [5, 1] about structuring concurrent software as a set of atomic actions. Each atomic action in our scheme is a dynamic entity consisting of a set of cooperative Ada tasks, each of which uses exception handling. We follow the definition of the atomic action used in [1]: processes exchange information only among action participants. This restricts the behaviours of the action participants: they should leave the action together.

Note that we will use the word *Ada* to refer to both Ada and Ada95 when our discussion concerns both of them (Ada95 has upward compatibility with Ada), and *Ada95* when talking about the peculiarities of this language. Besides, we will use the terms *process* and *task* interchangeably.

Our intention is to use Ada without changing it, to give a design scheme with the corresponding templates and to describe the design steps. This is in the line of some previous research reported recently [12, 13, 14]. We believe that it is time to map the approaches and schemes that are very well researched but little used in practice, onto a practical, widely used existing language. We believe that one of the main disadvantages of the previous research in software fault tolerance is that it is still rather theoretical and is used for exotic systems and languages. Nowadays fault tolerance is becoming more widespread, and more systems should have this property.

In order to deal with concurrent exceptions, we follow the general framework developed in [5]. Our scheme should guarantee either the synchronous exit of all tasks from the actions when all of them have reached the end of exception contexts successfully, or calling the handlers for the same exception (even if several tasks raise exceptions). Our scheme should allow nested actions and exception propagation along nested exception contexts, corresponding to the chain of nested atomic actions. We adhere to the termination model, whereby handlers take over the duties of the processes participating in the action and complete it (either successfully or by signalling a failure exception to the containing action).

Our scheme should resolve concurrently raised exceptions. After such resolution, the handlers for the same exception are called in all participants, and they either cooperatively recover the action and fulfil the function specified by the action specifications or cooperatively signal a failure exception to the containing action. Our belief in the importance of the resolution mechanism for concurrent systems and, in particular, for the systems implemented by using Ada is based on the following reasons:

1) Since there are no perfect error detection tools, the latent period of an error is not negligible, and erroneous information can be easily spread within an atomic action; thus, several errors occurring concurrently in different processes can be the symptoms of a different, more serious fault [5].

2) Very often there is a correlation between errors, so they happen over a very short period of time in different participating processes. For hardware-related errors, several processes can be affected by the same bad conditions or by the same transient error. For software design faults, as the participating processes were designed cooperatively from the same specification, a cooperative misunderstanding can affect the design of all of them.

3) It is impossible in Ada to interrupt or in any other way asynchronously inform the participating processes after one of them has raised an exception; as a result, they can be executed for quite a long period of time within which the errors caused by the same or by another fault could be detected in several of them.

Thus, a hierarchy-based exception resolution approach is needed in order to find the exception 'covering' all the raised exceptions and to determine the correct recovery strategy.

3. Atomic actions based on concurrent exception handling

3.1. Exception declaration

We consider that a common package should be used by all those action participants which are Ada tasks. It includes the declarations of all action exceptions. Each participant has to have a handler for any action exception because the handler of any of them can be raised during the resolution (even of an exception which is never raised in a given participant). For example, three exceptions of action A0 can be declared as follows:

```
A, B, C : exception;  -- action A0 exceptions
```

We strongly recommend here to follow the main ideas of [5]. In particular, we believe that a universal exception whose handler is intended for raising failure exceptions should be declared and used by the participants of all actions:

```
UNIVERSAL_EXCEPTION   : exception;
```

As Ada does not allow to transfer exceptions as parameters, for each action exception a unique integer value will have to be introduced. This will be transferred to the resolution procedure when an exception is raised. Besides, we introduce a special value, NO_EXC, which is used when the exception context is completed without raising an exception. FAILURE value is used to raise a UNIVERSAL_EXCEPTION.

```
NO_EXC: constant INTEGER :=0; -- for all actions
FAILURE: constant INTEGER :=1; -- for all actions
```

Programmers could choose any integer values for the action exceptions except for 0 and 1 which are reserved. For the given action A0 their declarations can be as follows:

```
EXC_A: constant INTEGER :=2;
EXC_B: constant INTEGER :=3;
EXC_C: constant INTEGER :=4;
```

If action A0 has a nested action, A1, then the declaration package for A1 can be as follows:

```
D, E : exception;
EXC_D: constant INTEGER :=2;
EXC_E: constant INTEGER :=3;
```

3.2. Exception context. Handlers. Predefined exceptions

The exception context in each action participant is the conventional Ada exception context, i.e. an Ada block. Each participant has to have handlers for all exceptions declared for this action. These handlers are attached at the end of the context in accordance with Ada rules. The OTHERS clause can be freely used here to handle several exceptions. Each handler includes an application code which provides the cooperative recovery and, in particular, action atomicity and data consistency. As was pointed out above, the handlers are to be programmed cooperatively to recover the action state, i.e. the states of all of its participants, in a coordinated way.

Since predefined exceptions are raised by the run time system, we include a new block with handlers for all of these. The only operation these handlers have to do is to raise an appropriate exception in accordance with our scheme, without recovering the program or handling the exception. It is allowed to handle a predefined exception within our scheme, but it should be declared and assigned a value in the same way as it is done for other exceptions; its handlers should be included at the end of the exception context (see Section 3.1).

The template of the exception context for a participant of action A0:

```
begin -- start of the exception context
   begin -- block for predefined exceptions
     -- ... application code
     exception
```

```
        when NUMERIC_ERROR | CONSTRAINT_ERROR |
          PROGRAM_ERROR | STORAGE_ERROR | TASKING_ERROR =>
             -- ... raising action exception
   end; -- end of the additional block
exception
     when A => -- ... application code
     when B => -- ... application code
     when C => -- ... application code
     when UNIVERSAL_EXCEPTION => -- ... application code
      -- ... raising the exception of the containing action
end; -- end of the exception context
```

3.3. Raising exceptions. Head process

All action participants have to be synchronized while completing the execution of their exception contexts. We offer a special statement which should be used for this. It allows a process to complete the context either by raising an exception or in the normal way. Processes do not use the Ada **raise** statement, and in both cases a service entry in the head process is to be called. This is a very important conceptual feature. It allows each process to inform the head process that it has reached the end of the exception context, and to wait until all of them have reached their context ends. The head process decides what all participants should do and lets them proceed synchronously. In this way our scheme guarantees the properties of atomic actions.

The programmer's conventions are as follows. Each action has a name, known to the designers of all participants. In each action a *head process* is chosen statically whose name is known to all participants. The head process includes a code for concurrent exception handling which is implemented in accordance with our template. Head processes receive information about exceptions raised in the participants or about participants' reaching the end of the exception contexts. For example, P0 can be the head process for action A0 (in which P0, P1 and P2 participate), and P1 is the head process for nested action A1 (with participants P0 and P1), and they have service entries for all action participants:

```
task P2 is
end P2;
task P1 is                     -- head process for A1
  entry RAISE_A1_P2(EXC_P2: in INTEGER);
end P1;

task P0 is                     -- head process for A0
  entry RAISE_A0_P2(EXC_P2: in INTEGER);
  entry RAISE_A0_P1(EXC_P1: in INTEGER);
```

```
end P0;
```

The integer exception values are parameters (Section 3.1). Ada does not allow to accept messages in the procedure (only in the task body); that is why each action has to have its own name of the entries. The assumption is that RAISE_ACTIONNAME_Pi is the entry called by Pi in the head process. The example for participant P2 of action A0 is as follows:

```
task body P2 is
-- ...
begin
  -- ... application code, other exception contexts
  begin -- start of the exception context
    begin
      -- ... application code (***)
      -- no exception, we have to call RAISE_A0_P2 entry:
      P0.RAISE_A0_P2(NO_EXC);
    exception
      when NUMERIC_ERROR | CONSTRAINT_ERROR |
        PROGRAM_ERROR | STORAGE_ERROR | TASKING_ERROR =>
        -- RAISE_A0_P2 is an entry in the head process:
        P0.RAISE_A0_P2(EXC_A);
    end;
  exception
    when A => -- ... handler
    when B => -- ... handler
    when C => -- ... handler
    when UNIVERSAL_EXCEPTION =>
      -- ... application code
  end; -- end of the exception context for action A0
end P2;
```

The head process accepts all entry calls by nested **accept** statements (one for each action participant); it does this in all points in which it completes the exception context (normally or by raising an exception). Head process P0 in action A0 does this as follows (we assume that P0 raises EXC_C):

```
accept RAISE_A0_P2(EXC_P2: in INTEGER) do
    accept RAISE_A0_P1(EXC_P1: in INTEGER) do
        RESOLUTION_A0(EXC_P2, EXC_P1, EXC_C);
    end RAISE_A0_P1;
end RAISE_A0_P2;
```

Programmers should not use handlers or nested blocks with handlers in the accept statements because this stops exception propagation to all tasks.

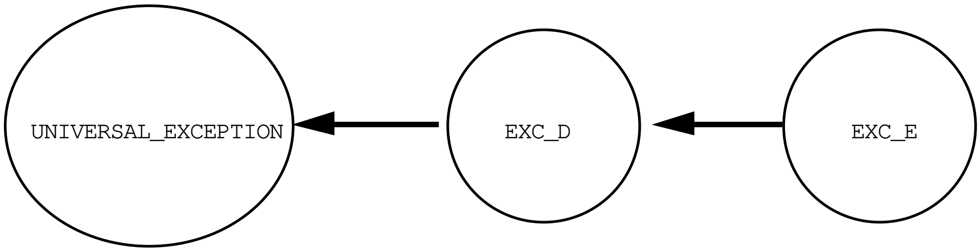
When an action participant executes the exception context, it is allowed to raise an exception of its action only. The signalling of the failure exception, i.e. raising an exception of the containing action, can be done in the handlers of the participants of this nested action. Note that this happens only after exceptions have been raised in all participants of the nested action and have been resolved. Within our scheme, exception re-raising (found in many exception schemes) is understood as raising the exception of the containing action.

3.4. Resolution procedure

The resolution procedure actually raises the Ada exception which is propagated by the run time system to all action participants through all nested rendezvous.

The convention is that the name of this procedure for action A0 is `RESOLUTION_A0`. It receives information (exception values) about all exceptions raised, resolves them and raises the corresponding exception. To implement the initial idea [5] about the exception tree in a general way, the tree data structure and corresponding operations should be implemented. There are several ways of implementing the resolution procedure practically. For example, we regard using the decision table as a barely adequate simple solution.

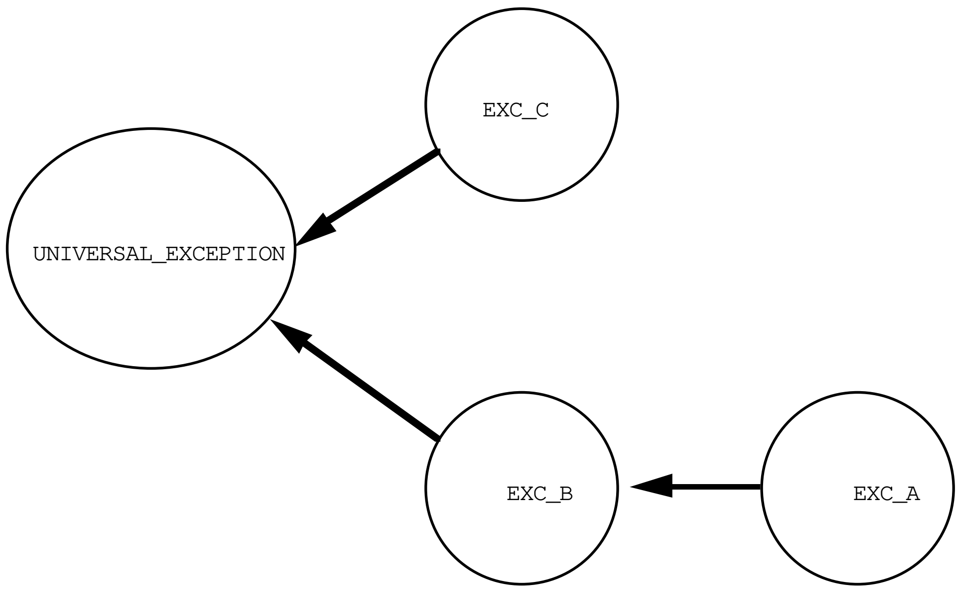
We will demonstrate how two particular cases of the exception tree could be programmed. For example, let us consider the following exception tree for nested action A1:



The corresponding resolution procedure is:

```
procedure RESOLUTION_A1(EXC_P2, EXC_P1: in INTEGER) is
    begin
      if EXC_P2=FAILURE or EXC_P1=FAILURE
                 then raise UNIVERSAL_EXCEPTION;
      elsif EXC_P2=NO_EXC and EXC_P1=NO_EXC then null;
      elsif EXC_P2=EXC_D or EXC_P1=EXC_D then raise D;
      else raise E;
      end if;
end RESOLUTION_A1;
```

For containing action A0 with three participants, the example of the exception tree could be as follows:



with the corresponding resolution procedure:

```
procedure RESOLUTION_A0(EXC_P2, EXC_P1, EXC_P0: in INTEGER) is
 begin
     if EXC_P2=FAILURE or EXC_P1=FAILURE or EXC_P0=FAILURE
     then raise UNIVERSAL_EXCEPTION;
     elsif EXC_P2=NO_EXC and EXC_P1=NO_EXC and EXC_P0=NO_EXC
     then  null;
     elsif (EXC_P2=EXC_A or EXC_P2=NO_EXC) and
        (EXC_P1=EXC_A or EXC_P1=NO_EXC) and
        (EXC_P0=EXC_A or EXC_P0=NO_EXC)
     then raise A;
     elsif (EXC_P2=EXC_A or EXC_P2=EXC_B or EXC_P2=NO_EXC) and
        (EXC_P1=EXC_A or EXC_P1=EXC_B or EXC_P1=NO_EXC) and
        (EXC_P0=EXC_A or EXC_P0=EXC_B or EXC_P0=NO_EXC)
     then raise B;
     elsif (EXC_P2=EXC_C or EXC_P2=NO_EXC) and
        (EXC_P1=EXC_C or EXC_P1=NO_EXC) and
        (EXC_P0=EXC_C or EXC_P0=NO_EXC)
     then raise C;
     else raise UNIVERSAL_EXCEPTION;
     end if;
end RESOLUTION_A0;
```

Within our approach, any sort of the resolution procedure can be designed by programmers. Obviously, other implementations of decision tables can be used here. To conclude the section, we would like to stress that using powerful Ada features together with our practical approach would make the implementation of the resolution procedures a routine job. Procedures of any sort can be easily programmed: they can allow using simple priorities of exceptions, resolution trees or even the most general approach mentioned in [5], within which all action exceptions are presented as a lattice.

3.5. Nested actions. Exception propagation. Failure exceptions

Structuring the system as a set of nested actions is the main way of reducing the complexity of the design [1]. Exception propagation allows to complete the nested action execution and to raise an exception of the containing action. One approach could be to program all these functions in the resolution procedure but we choose a different way which allows participants to execute the 'last will' or 'clean-up' functions and to recover the action state (to guarantee the action atomicity and data consistency). Within our approach, the propagation should be done out of the `UNIVERSAL_EXCEPTION` handlers by calling the entry of the corresponding head process of the containing action. These handlers coordinate their recovery and, if necessary, signal failure exceptions. There could be situations in which they could signal different failure exceptions, or only part of them could signal failure exceptions. The resolution procedure of the containing action resolves these.

In Section 3.3 we considered a template of process P2 participating in action A0. The following code can represent a nested exception context in process P2, e.g. the participation of P2 in action A1. This code should be inserted in the position marked by (***) in the template:

```
begin  -- the exception context for nested action A1
  begin
   -- ... application code
   P1.RAISE_A1_P2(NO_EXC); -- at the end of the context
   exception
     when NUMERIC_ERROR | CONSTRAINT_ERROR |
       PROGRAM_ERROR | STORAGE_ERROR | TASKING_ERROR =>
       P1.RAISE_A1_P2(EXC_D);
  end;
 exception
   when D => -- ... handler
   when E => -- ... handler
   when UNIVERSAL_EXCEPTION =>
   -- application code, cooperative recovery,
   -- providing atomicity and consistency
     P0.RAISE_A0_P2(EXC_A); -- signalling exception to A0
end; -- end of the exception context for A1
```

To provide the proper action nesting and exception propagation, we rely on a set of programmers' conventions. Processes should be implemented in accordance with the templates and rules mentioned. A subset of tasks from the containing action takes 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 the nestedness: in this respect handlers are parts (preludes) of the containing exception context. Though this is not the way how conventional propagation works (say, in Ada, C++, 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 for practice. It makes programmers include 'clean up' operations into each action and, in this way, guarantee local data consistency (e.g. by providing all-or-nothing semantics of the action with the exception raised).

3.6. Self-checking programming

Self-checking programming is intended to allow each process to be executed under the following failure assumption: it is guaranteed that within some specification-dependent time interval each process enters the atomic actions it is supposed to take part in and either raises an exception or reaches the end of the exception context. Implementing the atomic action features becomes much simpler with this assumption, under which each participant always enters the action and starts a cooperative recovery (or synchronizes its exit with other participants) after any faults that might happen in this or in other processes. Within this assumption, each nested action is completed either by signalling a failure exception or successfully. If this assumption is guaranteed , the atomic action support and/or all participants are free from the responsibility of checking the execution of each process and of solving the deserter process problem. This assumption is weaker than the well-known fail-stop assumption (guaranteeing which is a very hard task). We believe that, on the one hand, self-checking programming is not difficult to use and the corresponding assumption to guarantee, and, on the other hand, it allows facilitating the atomic action support considerably.

The idea is to make it the responsibility of each participant designer to implement the process properly. The main conventions by observing which programmers can guarantee this failure assumption are as follows. The catch-all handlers should be used within the exception context to guarantee that any predefined or un-handled exception is caught (note that this exception can be propagated from the nested procedure/entry calls). To avoid deadlocks, only selective accepts with delays and timed entry calls should be used. The recursion, while and repeat loops should be used in a restrictive form with some 'assert' operations to check its execution and to guarantee the restricted time of this execution. No input/output operations or resource requests without restricted waiting time are allowed.

With some languages, a watchdog, external for the process, can be used to make this programming simpler. There is no satisfactory solution of this sort either in CSP [11] or in Ada [12], because there is no way of interrupting a process asynchronously. Say, in Ada it can be done only by task aborting which is a rather drastic solution and cannot be regarded as a means of cooperative recovery.

Though we often use Ada terminology, we believe that the concept of self-checking programming is of great importance for using atomic action schemes in general. It allows to find a reasonable balance point between programming support and programmers' conventions in the situations in which it is very expensive and unpractical to lay all responsibility on the atomic action support. In Section 5.5 we will demonstrate how the deserter process problem can be avoided by using this programming.

4. Atomic action design

In this section we will briefly enumerate the main programmers' conventions for atomic action design. They allow to guarantee the atomic action properties and a proper functioning of our scheme.

The first is to follow closely our templates for building handlers and exception contexts, declaring and raising exceptions. The absence of information smuggling will be guaranteed by not allowing any data to be shared with non-participants, or any messages to be sent or received to or from non-participants, or any global variables to be used. All Ada predefined exceptions should be handled and should have handlers within each action context. All action exceptions should have handlers in each action participant. These handlers are to provide the action recovery, but even if they are not able to do this and decide to signal a failure exception, they should guarantee the action atomicity and data consistency. To make this easier, we recommend that no **in-out** formal parameters be modified in the exception context if it is part of the procedure body. Care should be taken of the consistency of all data which could have been modified before an exception is raised, in particular, of the data of other packages used during the action execution; the easiest way is to prohibit any procedures with a side-effect (see an extended discussion in [15]).

5. Comparisons and discussion

5.1. Our implementation

Though Ada has several important features which enabled us to implement concurrent exception handling and resolution and, speaking more generally, a sort of atomic actions with coordinated forward error recovery, some limitations of this language did not always permit reaching simple solutions. Ada allows using exceptions only in the operations of raising and in handlers (plus in `rename` declarations). Exceptions in Ada cannot be passed as parameters, compared, etc., which complicates the implementation of concurrent exceptions. This made us introduce exception values.

Another important restriction is that there is no feature in Ada to interrupt a process asynchronously and to raise an exception in it.

To increase the robustness of our scheme, timed entry calls and selective accepts with delays could be used when tasks rendezvous through RAISE entries. When the time-out expires, the handler for the UNIVERSAL_EXCEPTION should be called in the process; as a result, these handlers will be called in all participants which have reached the entry calls, because if one of them went this way, the others have to follow it. For example, the selective accept for head process P0 in action A0 can be as follows:

```
select
   accept RAISE_A0_P2(EXC_P2: in INTEGER) do
      select
       accept RAISE_A0_P1(EXC_P1: in INTEGER) do
           RESOLUTION_A0(EXC_P2, EXC_P1, EXC_B);
       end RAISE_A0_P1;
      or
       delay A0_TIME_OUT;
       raise UNIVERSAL_EXCEPTION; -- in the containing action
      end select;
   end RAISE_A0_P2;
 or
   delay A0_TIME_OUT;
   raise UNIVERSAL_EXCEPTION; -- in the containing action
end select;
```

For any process (P2 and P1 in our example) which is not the head one it could be as follows:

```
select
  P0.RAISE_A0_P2(EXC_A);
or
 delay A0_TIME_OUT;
 raise UNIVERSAL_EXCEPTION; -- in the containing action
end select;
```

5.2. What features could facilitate implementation

It is not the purpose of our paper to discuss disadvantages or extensions of Ada. But to clarify the main ideas, we would like to outline the language features which could make the implementation of concurrent exception handling and resolution simpler (the same holds for CSP implementation [11]). Exceptions should be of the type allowing to have values: to pass them as parameters, to compare and assign them. Language features supporting self-checking programming would facilitate the implementation of not just our scheme but of fault tolerant software on the whole. Having time exceptions that are raised on breaching time constraints could allow extending our approach to real time applications. Raising asynchronous exceptions in some other task could essentially facilitate the implementation of our approach.

5.3. Ada95

Exceptions in Ada95 [3] are basically the same as in Ada. But there is a library package, `Ada.Exceptions`, which can simplify our scheme. Its function `Exception_Name` returns the name (type String) of the exception raised. Our modified scheme will require using two nested blocks to declare the exception context (as the basic scheme described) with the only handler `OTHERS` in the nested block. Exceptions are to be raised by Ada **`raise`** statement. In the handler `OTHERS` the name of the exception raised is transferred to the head process as a parameter of the entry call. The resolution procedure manipulates all exceptions raised, resolves them and raises the exception which will be handled by all action participants. The resolution procedure deals with exception names (of type `String`), so it has to have the names of all action exceptions as constants(this eliminates using exception values). The exception resolved is to be raised in the resolution function by procedure `Reraise_Occurence` from package `Ada.Exceptions`. The handlers for all exceptions are to be declared in the second block of the exception context. An important detail is that an additional exception, `NO_EXCEPTION`, should be declared and raised when a participant completes the exception block successfully. This modified scheme treats predefined exceptions and programmer's exceptions in the same way.

Ada95 allows using an interesting approach to implement the resolution. The resolution tree could be declared as the tree of the derived types rooted at `UNIVERSAL_EXCEPTION`. Each of them should be a tagged type with two components of types: `exception` and `String`. The second component is the name of the exception and should be initialized when declared. The derived type tree exists at run time (for dispatching), which allows dynamic exception resolution. This approach allows to declare the resolution tree in a way which is very natural for Ada95 programmers and, we believe, will be found practical after further investigation.

Ada95 has new features which could allow us to facilitate self-checking programming (see the problems discussed in Section 3.6). The time limited calculation can be implemented using the asynchronous transfer of control: any block can be included into statement **select**, within which a time constraint is imposed by statement **delay**. With our scheme, each exception context should be implemented in this way with a time-out imposed by the programmer. In particular, for real time systems it could be the worst case execution time. A special programmer's exception `TIME_CONSTRAINT` can be used here with a convention for each participant to have the corresponding handler. Together with the suggestions in Section 5.1, this would allow using our concurrent exception scheme in real time systems for building atomic actions with predictable time behaviour.

Other features useful for self-checking programming can be found in Annex L (Safety and Security) of [3]. First, they are intended for supporting the predictable program execution: initializing all objects to canonical values, documenting the effects of all bounded errors (Ada95 Standard does not require these errors to be detected). And, second, they allow the use of some language features to be restricted (for example: recursion, subprogram reentrance by several tasks, unchecked type conversion, etc.).

5.4. Multiway rendezvous

Recently several proposals have been discussed which offer ways of multiparty process synchronization and information exchange [16, 17]. In particular, the multiway rendezvous (MWR) [4] is considered to be a natural extension of the two-participant rendezvous (found in Ada, CSP, SR, etc.). A MWR is a situation in which a block of statements is executed when, and only when, a specified number of appropriate processes reach a corresponding stage of execution [16]. The author discussed a general construct - compact - that encapsulates the MWR and allows declaring a set of entry blocks with **in** and **out** parameters in each of them, MWR statement blocks (one for each entry), etc. Each entry block lists entries that must be called for the corresponding rendezvous statement block to be performed. These entries are to be called by different processes. The MWR can occur only when all entries of an entry block have been called. The author discussed the corresponding extensions of several languages: Ada, CSP, etc.

We want to discuss two interesting topics: how the MWR can be implemented as a set of nested statements **accept** with a head process within the standard Ada by using our approach; and how the MWR can facilitate the implementation of concurrent exceptions.

Our implementation of the set of nested statements **accept** in the head process actually has the basic properties of the MWR: it starts when all participants are ready; the body of the innermost nested accept can manipulate all formal parameters of all statements **accept**; **out** parameters can be returned to any of the calling tasks, etc.

A caller - Pi - should use the following Ada statement:

```
   Ph.MWR_NAME_Pi(in and out formal parameters);
```

A callee - Ph - has a set of nested **accept** stements, one for each Pn, ..., P1:

```
accept MWR_NAME_Pn(in and out formal parameters) do
   -- ...accepts
      accept MWR_NAME_P1(in and out formal parameters) do
        -- ... rendezvous statement block
      end MWR_NAME_P1;
      -- ... end's
end MWR_NAME_Pn;
```

Although this scheme does not allow selective waiting or timed waiting, we believe that it can be used in practice. Moreover, it can be made efficient and capable of compensating for these disadvantages with the help of additional application dependent information (thus, we included timed waiting into our concurrent exception scheme in Section 5.1).

On the other hand, concurrent exception handling seems to show off the benefits of using the MWR because several processes have to rendezvous to allow one of them to resolve exceptions and to raise synchronously an appropriate exception in all of them (or to let them leave the action if there were no exceptions found). Thus, using the Ada extension proposed in [4], the operation of the head process can be programmed as follows:

```
accept RAISE_A0_Pn(EXC_Pn: in INTEGER) and ...
          and RAISE_A0_P1(EXC_P1: in INTEGER) do
                   RESOLUTION_A0(EXC_Pn, ..., EXC_P1);
end;
```

Note that we do not use **out** parameters here. We believe that Ada MWR [4] should be extended by rendezvous exception propagation conventional for Ada: an exception raised in a callee is to be propagated to all callers.

5.5. Deserter process problem

The deserter process problem has been given a lot of researchers' attention in the last years (e.g. [10, 11, 18]), but no satisfactory and simple solution has been found. The matter is that a process could be involved in a previous action which is not controlled by the action it is supposed to enter afterwards; or it could be in a nested action which is invisible and indivisible for the containing action that cannot be completed because of the desertion. We believe that self-checking programming is the right practical way of coping with deserters. It allows to involve all processes into the action execution and cooperative recovery because the execution of each of them is self-checked, and unpredictable delays cannot happen.

Time-outing by participants or by the action support with the purpose of detecting the deserter process is the wrong way. It cannot help because, generally speaking, recovery is impossible without a participant. Note that even producing the action results is not an acceptable solution because the deserter may try to produce the result later when it tries to participate in the action which has been completed. Moreover, even if this process has any information about the action completion, it has to execute its part of the computation anyway. Thus, the only solution is to signal a failure exception to the containing actions one by one until the action in which the deserter participates is found. The second problem is to coordinate somehow the time-outs of different actions, because detecting deserters by time-outs makes sense only if the time-out used in the previous action, in which the deserter can be kept, is shorter than those in the following actions, in which it is supposed to participate. This problem will get more complex if we take into account the fact that the deserter for the given action can exist because of another deserter process in another action. This effect could be called the *cascade desertion* and it seems to have no practical solution. A similar problem arises when the deserter is delayed in a nested action. A rather complicated solution could be to use coordinated time-outs which could guarantee that all nested actions have been completed before the time-out of the containing one is breached. Another solution could be to abort the nested actions but this contradicts the underlying idea of the nested actions because they are invisible and indivisible for the containing action, and, besides, a very complex protocol together with some programmers' conventions should be used here. The last problem is that even providing a recovery in which only some of the participants (without the deserter) [7] are involved does not actually recover the system because the fault which caused the desertion is not tolerated.

From the practical point of view, there is no satisfactory approach to time-outing (either in CSP [11], or in Ada [12]), which would allow interrupting the process execution asynchronously and calling an appropriate handler in it (task watchdogging and aborting is clearly an unacceptable solution for long-living processes [12]).

Thus, we insist on self-checking programming, which, apart from solving the deserter process problem, very often allows processes to detect an error before the exceptional situation could happen.

Though our scheme could be extended by using timed entries and select with waiting in all tasks (Section 5.1), we do not think that this is the right way. In accordance with self-checking programming, processes are to reach action exits and to participate in all actions they are supposed to.

5.6. Comparison

5.6.1. Jalote and Campbell's scheme

Paper [11] describes an atomic action scheme which uses an extended CSP. Unfortunately, CSP has no exception mechanism proper, and the paper does not give its complete picture: what is the exception context in each process, how exceptions are to be declared. Besides, the algorithm of exception propagation is not discussed and it is not clear how it works, how an exception of the containing action could be raised from within the nested one, etc. It is not clear what the authors mean by saying that an FT action is completed by signalling exception *e*. It seems to us that this paper did not discuss clearly how failure exceptions should be treated and propagated within CSP.

Within our approach, only one process resolves only once because anyway all processes have to wait for it: this could have been applied to [11]. It seems unnecessary for each process to resolve partially exceptions each time it receives information about a new exception. Our scheme does not use the chain algorithm, so the resolution works only once in the head process, and there is no additional parameter re-passing (it could be rather expensive for distributed systems with parameter marshalling/unmarshalling).

Unlike [11], we have no synchronous process entries into contexts because it would give only slight advantages but decrease the system parallelism. It cannot help to avoid the deserter process or to increase the fault tolerance of the scheme: a process can stop in the action and fail to reach the exit anyway. The scheme [11] cannot solve the problem of deserters: a process could enter an action and desert from there without raising an exception or reaching the end of the exception context. Besides, the scheme [11] uses synchronous entries to check the set of participants, but we do not need this sort of checking in our scheme because entry names are different for different actions (Ada allows this), since CSP uses only the process name to describe the destination but our scheme uses the process name and the action name for this purpose. The approach proposed in [11, p.66] to inform participants asynchronously about an exception raised by one of them is not complete, because if process has no message input or output (e.g. an endless loop, a rather long calculation, etc.) it cannot be informed by the broadcaster process.

We believe that though the idea of extending CSP [11] was suitable for demonstrating new approaches, it is not acceptable for practical programming. We do not rely on new constructs in the original language: processes participate in the atomic action by executing Ada blocks which are exception contexts. We introduce atomic actions with forward error recovery only by using Ada concepts and give clear rules how to program these actions using standard Ada.

5.6.2. Burns and Wellings's scheme

The authors [12] discuss an Ada atomic action scheme which uses a service task (action controller) having a set of nested **`accept`** statements, one for each participant (the head process does this in our scheme). Each of them informs controller about the code of the exception to be raised. Having received all codes, the controller raises the appropriate exception which propagates to all participants (similarly to our approach).

These similarities notwithstanding, our approach is very different. We have found that the approach [12] is not complete and a lot of important questions were left unanswered. We do not use any service task because it causes some problems: it is not clear when and how it should be started, whether service tasks should always exist for all actions which can dynamically appear and disappear, how controllers could be implemented for a task participating successively in several actions (with different participants). We give a complete set of rules and templates for signalling failure exceptions, building nested actions and guaranteeing the absence of information smuggling and action atomicity. Following [5], we have introduced a handler for the universal exception to structure and to unify signalling a failure exception and building the exception tree. Actually, the scheme [12] does not work when several actions are executed concurrently, because conventions for naming controllers and entries are not discussed properly. Within our general scheme, the exception contexts of different participants form atomic actions rather than task bodies, as is the case for [12].

Periodical synchronization [12] is rather expensive and restrictive: it could hardly increase the scheme robustness or facilitate error detection. It does not allow either detecting the erroneous process, or initiating its recovery. Clearly, there is no need in the additional synchronization if the participants are executed correctly, so its use contradicts one of the main requirements fault tolerance schemes should meet: not to decrease the performance where there is no error detected. Besides, we believe that in the situation when an action can be naturally split into several consecutive 'frames', it would be reasonable and cheap to make each of them an atomic action and to design the system as a chain of actions. This could give real benefits.

The resolution within our scheme extends the approach from [12] in a very natural way because this scheme has resolution features only in embryo. We have demonstrated how the resolution procedure can be used and have given several examples of programming this procedure by using different resolution trees. Note that the problem of the predefined Ada exceptions was not addressed in [12].

Our approach is much safer than the scheme in [12]: first, it relies on self-checking programming and, second, it can be easily extended to allow robust programming and the continuation of the service when some processes fail and do not reach the end of exception contexts (Section 5.1).

The authors [12] rightly outlined the main problems which arise during implementing coordinated forward recovery in Ada. We hope that self-checking programming gives a sound approach to solving them. We believe that, unlike the approach [12], we discuss a complete and practical scheme which corresponds to the general atomic action paradigm [5].

6. Summary and conclusions

We have discussed the approach which allows programming atomic actions with coordinated concurrent exception handling by using standard Ada and Ada95 languages. Our scheme relies naturally on the peculiarities of these languages: atomic actions are formed of the exception contexts of the cooperating tasks, a synchronous exit is guaranteed by the synchronous execution of the nested rendezvous's, raising the same exception in all participants is provided by the exception raising semantics of the rendezvous.

We would like to conclude with two points which we believe to be important. The first is that our approach can be used for implementating industrial applications in the nearest future. Secondly, the idea of extending existing languages by a set of templates and programmers' conventions (maybe, with an appropriate post- or pre-processor) can essentially extend the applicability of all main fault tolerance schemes in the situation where the gap between the achievements in theoretical and experimental research and an increasing demand for fault tolerance of all applied systems (which have to be programmed in standard languages) is getting wider.

7. Acknowledgments

The cooperation with Brian Randell and Jie Xu, my colleagues from Newcastle University, profoundly influenced this research. The supports from the Royal Society (project 638072) and DeVa Basic Research ESPRIT Action are gratefully acknowledged.

8. References

[1] LEE, P.A., and ANDERSON, T.: 'Fault Tolerance: Principles and Practice' (Springer-Verlag, Wien - New York, 1990)

[2] CRISTIAN, F.: 'Exception Handling and Tolerance of Software Faults', in: LYU, M., ed., Software Fault Tolerance (J. Wiley, 1994) pp. 81-107

[3] 'Ada Reference Manual. Language and Standard Libraries'. Version 5.95. ISO/IEC 8652:1995(E). Intermitrics, Inc., 1994

[4] MEYER, B.: 'Eiffel. The language' (Prentice Hall, N.Y., 1992)

[5] CAMPBELL, R.H., and RANDELL, B.: 'Error Recovery in Asynchronous Systems', *IEEE Trans.* **SE-12** (8) (1986) pp. 811-826

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

[7] NELSON, G. (Ed.). 'System Programming with Modula-3' (Prentice Hall, N. Y., 1991)

[8] HUANG, D.T., and OLSSON, R.A.: 'An exception handling mechanism for SR', *Comput. Lang.* **15**, (3) (1990) pp. 163-176

[9] KLINGERMAN, E., and STOYENKO, A.D.: 'Real-Time Euclid: A Language for Reliable Real-Time Systems', *IEEE Trans.* **SE-12** (9) (1986) pp. 941-949

[10] KIM, K.H.: 'Approaches to Mechanization of the Conversation Scheme Based on Monitors'. *IEEE Trans.* **SE-8** (1) (1982) pp. 189-197

[11] JALOTE, P., and CAMPBELL, R.H.: 'Atomic Actions for Fault-Tolerance Using CSP', *IEEE Trans.* **SE-12** (1) (1986) pp. 59-68

[12] BURNS, A., and WELLINGS A.J.: 'Real-time Systems and their Programming Languages' (Addison-Westley, 1989)

[13] CLEMATIS, A., and GIANUZZI, V.: 'Structuring conversation in operation/procedure oriented programming languages', *Comput. Lang.* **18** (3) (1993) pp. 153-168

[14] ROMANOVSKY, A.: 'Application specific conversation schemes for Ada programs', *Microprocess. and Microprogram.* J., **41** (1995) (accepted)

[15] ROMANOVSKY, A., and STRIGINI, L.: 'Backward error recovery via conversations in Ada', *Softw. Engng. J.* **10** (8) (1995) pp. 219-232

[16] CHARLECWORTH, A.: 'The Multiway Rendezvous', *ACM Trans. on Prog. Lang. and Systems* **9** (2) (1987) pp. 350-366

[17] EVANGELIST, M., FRANCEZ, N., and KATZ, S.: 'Multiparty Interactions for Interprocess Communication and Synchronization,' *IEEE Trans.* **SE-15,** (11) (1989) pp. 1417-1426

[18] DI GIANDOMENICO, F., and STRIGINI, L.: 'Implementations and extensions of the conversation concept', Proc. 5-th International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems - Tests, Diagnosis, Fault Treatment, Nßrnberg, Germany (1991)