Extending Conventional Languages by Distributed/Concurrent Exception Resolution

A.Romanovsky
Department of Computing Science, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

The state of art in handling and resolving concurrent exceptions is discussed and a brief outline of all research in this area is given. Our intention is to demonstrate that exception resolution is a very useful concept which facilitates joint forward error recovery in concurrent and distributed systems. To do this, several new arguments are considered. We understand resolution as reaching an agreement among cooperating participants of an atomic action. It is provided by the underlying system to make it unified and less error prone, which is important for forward error recovery, complex by nature. We classify atomic action schemes into asynchronous and synchronous ones and discuss exception handling for schemes of both kinds. The paper also deals with introducing atomic action schemes based on exception resolution into existing concurrent and distributed languages, which usually have only local exceptions. We outline the basic approach and demonstrate its applicability by showing how exception resolution can be used in Ada 83, Ada 95 (for both concurrent and distributed systems) and Java. A discussion of ways to make this concept more object oriented and, with the help of reflection, more flexible and useful, concludes the paper.

Keywords: atomic actions, forward error recovery, coordinated exception handling, Ada 83, Ada 95, Java

1. Introduction

Fault-tolerant software detects errors caused by faults and employs error recovery techniques to restore normal computation [13]. 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 handling mechanism is a language/system feature allowing programmers to describe the replacement of the standard program execution by an exceptional one when an occurrence of exception (i.e. anything inconsistent with the program specification) is detected (see [5] for a rigorous and thorough discussion). This mechanism is considered an essential part of any modern language (e.g. Ada 95 [1], C++, Eiffel [14], Java [9]).

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 or procedure bodies. Each context has a set of associated exception handlers; one of these 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 [5, 13]: it is adhered to in all the languages mentioned. The resumption model assumes that the handler recovers the program state, and the program continues execution from the operation following the one that raised the exception. If there is no handler for the raised exception in the context or it is there but unable 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 that in which the handler was not found or was not able to recover the program.

2. Atomic actions. Concurrent exception resolution

Exceptions are much more difficult to handle and fault tolerance to provide in concurrent and distributed systems because, firstly, in the general case several cooperating processes should be involved in handling if an exception has been raised in one of them; secondly, multiple exceptions can be raised concurrently in such systems; and lastly, more sophisticated run-time support should be used [16]. The approach to using exception handling in such systems proposed in [4] extends the well-known atomic action paradigm [13]. Atomic actions are the most comprehensive way of structuring (dynamic) behaviour of the concurrent systems. These actions are units of process cooperation and their execution is indivisible and invisible from the outside. Atomic actions are a general and sound basis for building fault tolerance schemes which allow processes to cooperate during recovery and which associate all phases of providing fault-tolerance with this dynamic system structuring. The paper [4] describes the main rules of cooperative recovery: all processes taking part in an action are to be involved into recovery if any of them detects an error; the features intended for recovery after the same error are called in all processes taking part in an action when the action recovery is required.

An atomic action is formed by a set of cooperative processes (*action participants*) each of which participates in the action while executing its corresponding block of code, i.e. the participant exception context. All participants' blocks form the exception context of the action. A set of exceptions is associated with each action. Each participant has a set of handlers for (all or part of) these exceptions. Participants enter the action by entering their exception contexts. Their entries may be asynchronous but they have to leave the action synchronously to guarantee that no information is smuggled to or from the action (this allows the main action properties to be guaranteed). If an exception has been raised in a process inside an action, all action participants have to participate in the recovery, and the handlers for the same exception have to be called in all of them [4]. These handlers cooperate to recover the action. The participants can leave the action on three occasions. First of all, this happens if there have been no exceptions raised. Secondly, if an exception had been raised, and the called handlers have 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* [4] is the essential part of concurrent exception handling since several independent exceptions can be raised at the same time, or several errors detected which are the symptoms of a different, more serious fault. [4] offered a solution which relies on using a *resolution procedure*: this resolves all concurrent exceptions and works out a generalised exception the handlers for which will be called in all action participants. The concept of the *exception tree* [4] is more appropriate than exception priorities for resolving these exceptions. This tree includes all exceptions associated with the action and imposes a partial order on them in such a way that a higher level exception has a handler capable of handling any lower level exception.

None of the distributed and concurrent languages with exception handling features (e.g. Arche [7], Ada 95, Ada 83, Modula-3 [15], SR [6], Real-Time Euclid [12], Java) allows using atomic actions with forward error recovery. We will say that all these languages have a *local* exception handling.

3. Why resolution is important. Resolution as agreement

We believe that the resolution mechanism is very important for concurrent and distributed systems and that it may be worthwhile to consider it as part of a future language or a distributed service (as a natural extension of the traditional exception mechanism). There are many reasons to support this and one of the intentions of this paper is to discuss them. The obvious reason is that several exceptions can be raised concurrently and this situation should be dealt with. We shall offer several arguments to explain why this may happen more often that one expects, why we cannot afford to ignore this situation and why resolution should be included into support rather than left to application programmers. We shall summarise the papers [4, 17, 18] and discuss some new topics.

First of all, 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 more serious fault. Secondly, very often there is a correlation between errors, so they happen over a very short period of time in different participating processes. With hardware-related errors, several processors or/and communication lines can be affected by the same bad conditions (in the case of a line this can affect all traffic going through it). With software design faults, as participating processes were designed cooperatively from the same specification, a cooperative misunderstanding can affect the design of all of them.

Another reason is that in practice it is impossible to interrupt all participants the moment one of them has raised an exception. And the probability of new exceptions being raised in other participants before they are informed about this exception is much higher in a distributed system. The overall hardware failure probability is very high in distributed systems and they are more difficult to program without design errors than centralised ones. Some languages (CSP, Ada 83) simply have no features to interrupt or in any other way asynchronously inform the participating processes when one of them has raised an exception; as a result, they could go on being 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.

Consider a situation when resolution is not used and concurrent exceptions are lost: handlers for only one of them are called in all participants. To cope with this, all handlers for all exceptions should be able to recover in situations either with or without these lost exceptions. But as the information about lost exceptions can be vitally important, all these handlers have to execute general error detection ('lost exception' detection), which can be very expensive. The original idea in [4] is that handlers should cooperate during recovery, but without resolution they will have to cooperate to detect errors before they can start recovery. After this, since each of them can find different errors, they are to cooperate to reach an agreement about the 'covering' error for which recovery actions should be started afterwards. It is clear that the approach in [4] tries to make recovery faster and more universal and programmers' jobs simpler because error detection which is executed by all participants before exceptions are raised is taken into account and because finding the covering exception is provided by the fault tolerance feature. It is obvious that, unlike backward error recovery, forward error recovery cannot be made transparent; it is inherently complex and application dependent, and traditionally only a basic exception scheme is used. Exception resolution makes this recovery much simpler. It should be interpreted as reaching an agreement among action participants, to be provided by the underlying support in a unified and automatic way. In Figure 1 we show two systems; in the first of these, application handlers have to repeat error detection and resolve the errors detected, but there is no need to repeat the error detection for the second one. Moreover, the underlying support executes resolution.

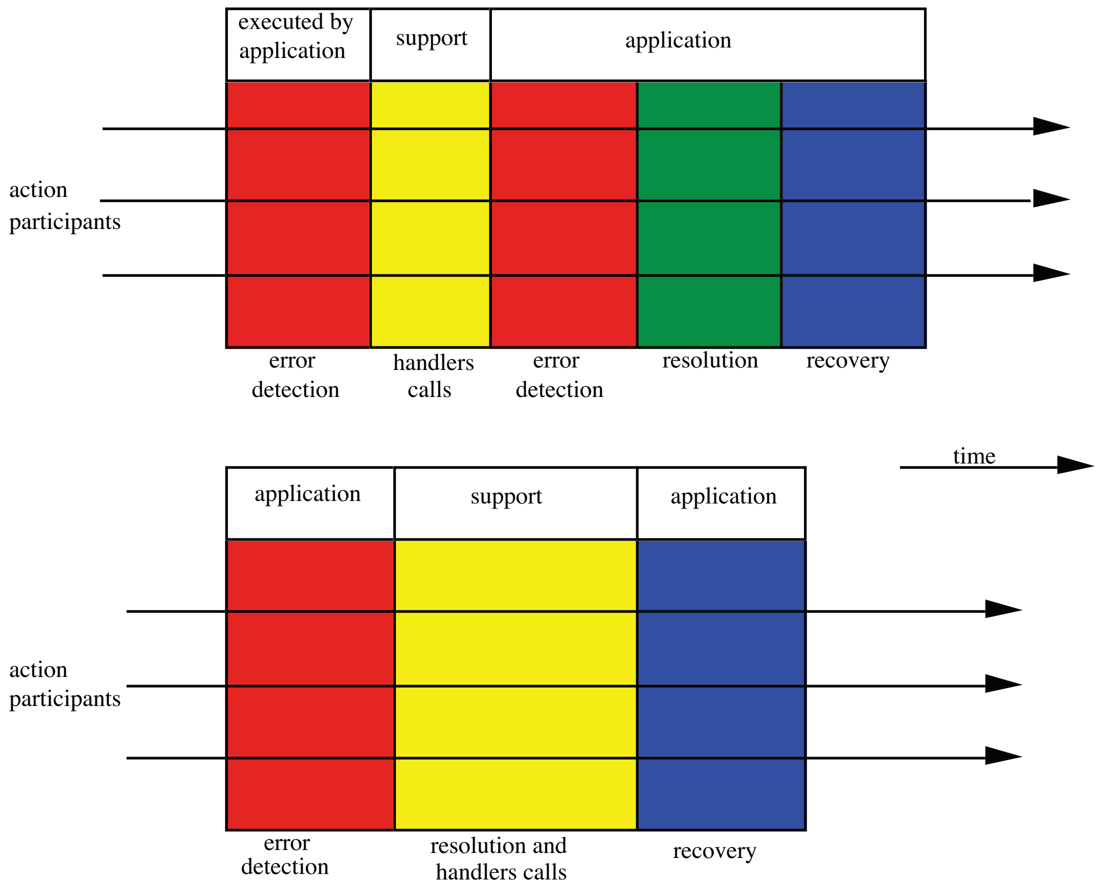

Figure 1. Action recovery for cases when resolution is executed by handlers (the first system) and by the underlying support (the second one)

Now we would like to consider these arguments in detail. If exceptions can be lost, then, we believe, each participant has to find out (from within the handler) whether it had an exception which was lost and only afterward it can try to recover. Consider a situation when two participants raise `Fire_Alarm` and `Gas_Leakage` exceptions concurrently. It is obvious that we must not lose either of them and that knowing both of them changes the recovery drastically. The same holds for situations when handlers for a less important exception are called in all participants and the information about a very important error is lost. This shows that all exceptions should be found when handlers are executed, and it is the responsibility of each handler.

Losing exceptions seems to be an idea that does not go together with exception handling; the point is to pass as much information as possible between the normal and abnormal program states (it is for this purpose that parameterised exceptions have been introduced). Taking this idea to its extreme, we should use no exception names at all (but rather raise `The` exception), because handlers would have to detect the error(s) anyway.

Moreover, because all action participants have to be synchronised at the 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 an additional resolution stage.

In the next section we will discuss all existing resolution schemes, each of which offers a particular implementation of the general approach in [4]. They are intended for different application areas, languages, and atomic action schemes.

4. Existing resolution schemes

The paper [8] offers a scheme for using an extended CSP to implement atomic actions with both forward and backward error recovery. To guarantee a synchronous exit together with the exception resolution, the authors propose statically connecting all action participants in a virtual chain and synchronising them by rendezvous through this chain when they reach the end of the action (and, in particular, when exceptions are raised). This allows each process to receive the information about the exception from the 'left neighbour' process, to partially resolve the exception and to transmit the resolution result to the '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 believe that though this scheme suggests a very interesting approach, it cannot be directly applied in practice, primarily because CSP is an experimental language. Moreover, the authors had to extend it by time-outs and exceptions. Another drawback is that the scheme requires all participants to enter the action synchronously.

Some preliminary but very important steps were done for Ada 83 in [3]. The authors discuss an Ada 83 atomic action scheme which uses a service task (the action controller) having a set of nested statements **`accept`**, one for each participant. Each of them informs the controller of the code of the exception to be raised. Having received all codes, the controller raises the appropriate exception which propagates to all participants. We believe that this scheme is a promising approach that makes atomic actions practical. Unfortunately, this scheme is rather expensive (because of the repeated process synchronisation) and can be essentially improved (see [18] for a detailed analysis). Note that Ada 95 is upwardly compatible for virtually all Ada 83 applications and this scheme can be directly used in Ada 95.

Concurrent object-oriented language Arche [7] allows resolving failure exceptions signalled by several different implementations of the same class (view, in Arche terms), so, this approach is not intended for coordinated recovery of concurrent cooperating processes or objects. The resolution procedure inputs all exceptions and returns one "concerted" exception to be handled in the context of the calling object. These procedures are application dependent because Arche uses parameterised exceptions and, as is rightly pointed out in [7], the resolution tree is not applicable for these exceptions (one cannot know the parameters required for the resolved exception).

The paper [17] discusses two very important issues: introducing concurrent exception handling into object oriented systems and a distributed resolution algorithm. Though this research is intended for the coordinated atomic (CA) action scheme [20] (a scheme that integrates conversations and transactions and allows both forward and backward recovery), it is quite general and can be used for any object oriented system, and so can the algorithm for resolving exceptions in any distributed system in which its assumptions are valid. This approach is essentially object oriented and exceptions are thought of on the class/object levels. The distributed resolution algorithm is less complex than the one briefly discussed in [4].

Basically, resolution implementations can be either decentralised or centralised depending on the way the synchronisation of participants with subsequent resolution is implemented. The two schemes in [8] and [3] are centralised and the location of the synchronising process is known statically: it is the head process for the first scheme and the controller for the second. The distributed scheme in [4] is essentially decentralised because all processes inform each other about their states and all of them resolve exceptions. Another distributed scheme in [17] is 'less' decentralised because the resolving process is found dynamically as the one having the biggest number of all processes in which an exception has been raised. The question involved is where to keep the resolution tree during the run-time. Depending on the implementation and on the resolution algorithm used, the tree can be kept either in all participants [4, 8, 17], or in one location (in the action manager [3]). Obviously, the resolution tree is kept where the resolution happens (or can happen): in schemes [4] and [8] all processes resolve exceptions (in [8] partially), but in schemes [17] and [3] only one site does so. The choice of the resolution implementation depends on application requirements and peculiarities, failure assumptions, the underlying support, etc.

5. Concurrent exceptions in asynchronous and synchronous actions

With respect to involving participating processes into recovery there are two kinds of atomic action schemes: synchronous and asynchronous ones. In synchronous schemes, each participant has to either come to the action end or to find an error and to raise an action exception; it is only afterwards that it is ready to accept information about the states of other participants and, if necessary, to be involved into action recovery (this means that its execution cannot be pre-empted if another participant raises an exception). Asynchronous schemes do not wait for this but use some language feature to interrupt all participants when one of them has found an error. Generally speaking, resolution is required for either kind of schemes. In the meantime the asynchronous Ada 95 atomic action scheme [19] relies on ignoring all exceptions but one. So, there is no resolution: although several exceptions can be raised concurrently, only one of them will survive and the handlers for it will be called in all participants. Analysing this scheme shows that it is impossible not to lose some of the exceptions which are raised concurrently when Ada 95 asynchronous transfer of control is used as the only means to implement an asynchronous scheme.

The choice between the two kinds of schemes will be made depending on the application, on the error which has been detected, on the failure assumption, etc. But the general scheme should include features for programming actions of either kind.

Recovery and resolution are much easier to provide in synchronous systems than in asynchronous ones because each participating process is involved in the action recovery only when it is ready for it 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 all nested actions have to be completed before the recovery starts. Obviously, there is a risk that deadlocks can stop these systems, but we believe that cautious programming with an intensive error detection would make it possible not just to avoid this problem but make the subsequent recovery simpler. There is no wasting of time in asynchronous schemes but the features required for process interruption are not readily available in many languages and systems. Even when they are, they are usually very expensive: for example, many implementations of the asynchronous transfer of control in Ada 95 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 them. Very often some restrictions are imposed on the program segment that can be interrupted asynchronously, in an attempt to make the implementation less expensive. For example, Ada 95 tasks cannot accept messages within a segment like this. One more difficulty with asynchronous schemes is that the abortion of nested actions is difficult to program. Some additional programming rules can improve synchronous schemes and decrease time waste (time-outs, assertions, checking invariants, pre- and post-conditions, etc.). Examples of the systematic approaches which allow programmers to achieve this are defensive programming and self-checking programming [21]. They can allow an early detection of either the error or the abnormal behaviour of the action participant which has raised an exception and is waiting for the other participants.

The scheme in [8] is basically a synchronous scheme, although there is a proposal, based on a CSP extension, which allows an asynchronous scheme to be implemented (although this proposal is not complete, because if a process has no message input or output it cannot be informed by the broadcaster process). The paper [3] describes a synchronous scheme as Ada 83 has no features to interrupt another process (other than just abort it).

In the following section we offer a general approach to introducing exception resolution in centralised synchronous atomic actions programmed in existing languages and demonstrate its applicability using Ada 83, Ada 95 and Java.

6. Introducing exception resolution into existing languages

6.1. General approach

We would like to describe now how exception resolution can be used in systems programmed in existing languages. This description should serve as a basis to be developed into a set of templates and conventions for application programmers to follow (we realise that this may be error prone, but there are some features which can help programmers to avoid mistakes: post- and pre-processors, libraries, syntax oriented editors, macro-processing). We do not want to introduce a new language construct for atomic action declaration: this would make the approach not feasible for existing languages. This is in line with some research reported recently (see, for example, schemes [3, 16, 19] which are intended for standard Ada 83/Ada 95). We believe now is the right time to map the fault tolerance approaches and schemes [13] that are very well researched but are not used in practice very often, onto practical, widely used existing languages. It seems to be one of the main flaws of the previous research in software fault tolerance that it is still rather theoretical and is applied to exotic systems and languages.

We would like to rely on the existing language features as much as possible and we will use their local exception mechanism as the base. We will rely on the general ideas [4, 13] about structuring concurrent software as a set of atomic actions. Let us describe how an atomic action scheme should be programmed. Each atomic action in this scheme is a dynamic entity consisting of a set of cooperative concurrent processes using their local exception handling. We follow the definition of the atomic action used in [13]: processes exchange information only among action participants. This restricts the behaviours of action participants: they must not interact with the outer processes and should leave the action synchronously. The scheme should guarantee either the synchronous exit of all processes from actions when all of them have reached the end of exception contexts successfully, or the call of handlers for the same resolved exception (even if several processes raise their exceptions). The scheme is to 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). The scheme should resolve concurrently raised exceptions by using the resolution tree (which is application dependent and provided by programmers). 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.

We need a special synchronisation protocol in which all participants are to be involved. It should work as follows:

- all action participants start it by sending information about their states (successful completion or raising an exception);
- a special *synchronising and resolving process* (SR-process), which plays the role of the centralised action controller, collects all this information (it waits until all participants arrive);
- if two or more processes have raised exceptions, the SR-process resolves them by using a statically declared resolution tree;
- the SR-process either calls the handlers for the resolving exception in all participants or allows processes to leave the action;
- no process is allowed to leave the action without executing the synchronisation protocol.

The set of local exception contexts of all participants forms the atomic action. We assume that the SR-process keeps the resolution tree of all exceptions associated with a given action and that all participants have local exception handlers for all exceptions from the tree. When we approach the implementation of this protocol in a concurrent language with sequential exception handling we have to tackle the questions which have language-dependent answers: passing the local exception to the resolving process, process synchronisation, raising the resolving exception. To demonstrate the applicability of the approach proposed we will outline several particular implementations.

One detail should be mentioned. We assume that there are ways which allow the atomic action support to detect a process which has died and failed to inform the SR-process. We did not want to extend our schemes by additional code which would provide this service. Moreover, we believe that for many applications this service is quite separate and can be provided by the underlying support: for example, the crash of a node containing an action participant can be detected by the inter-node heart-beating. There are many techniques which essentially depend on applications and on features available in the language (e.g. timed entries and asynchronous transfer of control can be used in Ada 95). One solution of this problem is to allow SR-process to know the list of action participants. Another good approach is using special design for creating self-checking processes [21].

6.2. Ada 83

We consider the use of atomic actions in Ada 83 as the first example of employing our general approach. The scheme in [18] which we are going to describe is synchronous. One of the action participants is the head process, which is chosen statically by the action designer and which executes nested statements **`accept`** when it finishes the action. Each of those is called by one of the other participants on finishing the action (with or without exception). An enumeration type containing the names of all action exceptions is to be declared for each action (e.g. `Action_A0_Exceptions_T`) and a value of this type is passed to the head process by each participant. When the head process has received information from all participants, it calls the resolution procedure to resolve exceptions and raises the covering exception. An interesting detail is that this exception is propagated by the Ada 83 run-time to all action participants through the nested statements **`accept`**. For example, the head process for an action with three participants looks as follows:

```
task P0 is
       entry RAISE_A0_P2(Exc_P2: in Action_A0_Exceptions_T );
       entry RAISE_A0_P1(Exc_P1: in Action_A0_Exceptions_T );
end P0;
```

When it completes the action (in either way), it executes:

```
accept RAISE_A0_P2(Exc_P2: in Action_A0_Exceptions_T ) do
       accept RAISE_A0_P1(Exc_P1: in Action_A0_Exceptions_T ) do
              RESOLUTION_A0(Exc_P2, Exc_P1, Exc_C);
       end RAISE_A0_P1;
end RAISE_A0_P2;
```

The exception context for each participant (except for the head process) is as follows:

```
begin -- start of the exception context
   begin
     ... -- application code, participation in the action
      exception
        when Numeric_Error | Constraint_Error | Program_Error |
                     Storage_Error | Tasking_Error =>
             ... --  raising an action exception
   end; -- end of the additional block
exception
     when A => ... -- application recovery code
     when B => ... -- application recovery code
     when C => ... -- application recovery code
     when Universal_Exception => ... -- application recovery code
      ... -- raising the failure exception in the containing action
end; -- end of the exception context
```

An additional Ada 83 block has been introduced to catch predefined exceptions. This scheme can be classified as a centralised one, with one resolving process keeping the resolution tree.

6.3. Ada 95

Exceptions in Ada 95 [1] are basically the same as in Ada 83 and the approach above works for Ada 95. But the new package `Ada.Exceptions` allows it to be simplified. Its function `Exception_Identity` returns the distinct identity (`Exception_Id`) of the exception raised. The modified scheme requires using two nested blocks to declare the exception context (identically with the basic scheme described) with the only handler `OTHERS` in the nested block. Exceptions are to be raised by Ada statement **`raise`**. In handler `OTHERS` `Exception_Id` of the raised exception is transferred to the head process as a parameter of the entry call. The resolution procedure manipulates the identities of the exceptions raised, resolves them and raises the exception which will be handled by all action participants. The resolution procedure deals with the exception identities, which are kept in the resolution tree (this eliminates using exception values). The resolved exception is raised by procedure `Raise_Exception` from the same package. 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_Exc`, will 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.

Further simplifications can be made using Ada 95 protected types [1]. 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.

6.4. Distributed systems in Ada 95

In this section we will give a sketch of programming atomic actions with exception resolution in distributed Ada 95 programs (see Distributed Systems Annex [1]). The schemes above will not work in distributed systems because protected objects and task entries cannot be called remotely and `Exception_Id` cannot be passed between Ada 95 partitions. We will modify the scheme proposed in Section 6.3 and make use of the fact that exceptions are propagated through remote procedure calls in Ada 95. We assume that action participants (say, `P1`, `P2`, `P3`) reside on different partitions and we introduce a service partition, `SR_partition`, of category `Remote_Call_Interface` [1]. This partition (which is an Ada package) has a service procedure for each action participant:

```
package SR_partition is
       pragma Remote_Call_Interface;
       procedure P1_resolve (Exc_P1: in Action_A0_Exceptions_T := No_Exc);
       procedure P2_resolve (Exc_P2: in Action_A0_Exceptions_T := No_Exc);
       procedure P3_resolve (Exc_P3: in Action_A0_Exceptions_T := No_Exc);
end SR_partition;
```

Each package (including the service one) should be compiled with the application dependent package of category `Pure` [1] containing types and data common for all action participants (action exceptions and the enumeration type):

```
package Action_A0 is
       pragma Pure;
       type Action_A0_Exceptions_T is (Exc_A, Exc_B, Exc_C, No_Exc, Failure);
       A, B, C : exception;
       Universal_Exception : exception;
end Action_A0;
```

When each action participant completes its execution or finds an error it calls the corresponding service procedure (remotely and synchronously) and passes the value of the exception it is going to raise. There is a private service protected object `SR_object` in package `SR_partition`. The resolution procedure is called from entry `Finish` when all participants have called procedures: `P1_resolve`, `P2_resolve`, `P3_resolve`, and, afterwards, the resolved exceptions is propagated to each of these procedures and to all action participants.

6.5. Java

The idea is again to rely on the local exception handling of Java [9]: the exception contexts (blocks `try`) of all action participants (threads) form the exception context of the atomic action. Each participant completes the execution of its exception context by throwing either an action exception it is going to raise, or a predefined exception `noException` (to inform about the successful completion). There is one service handler which catches all exceptions and calls service `SRmethod`, passing the exception as the parameter. `SRmethod` is a Java synchronised method and that guarantees that it is executed in an exclusive mode by only one thread (but if **`wait`** is executed, another thread can start execution). All but one threads are waiting on **`wait`** until the last one arrives, resolves the exception and notifies all of them (to let them proceed). Each of the threads throws the resolved exception:

```
       private exceptionActionA0 re;
private synchronized void SRmethod(exceptionActionA0 e)
                     throws excA, excB, excC, universalException {
       number--;
       if (!re.getClass().getName().equals ("noException")) collect(e);
       if number == 0 {
                     re = resolve();
                     number = before;
                     notifyAll();
       }
       else {
              try {
                    wait();
              } catch (InterruptedException ex) { ...
              }
       }
       if (!re.getClass().getName().equals ("noException")) throw re;
       }
}
```

Each block `try` in each action participant has to have blocks `catch` to catch all exceptions which are raised during the execution. These blocks should be included in the code, from the leaves of the resolution tree to the handler for the universal exception (the root), because the handler is searched for starting from the first block `catch` and the ancestor handler is called when a descendant exception is raised.

This scheme works for distributed Java systems which use remote method invocation [10].

7. Future research. Object orientation. Conclusions

Although several interesting ideas about ordering exceptions in other ways than trees (any partial order can be used, e.g. lattices) were mentioned in [4], we agree with the authors that the tree structure is the most suitable one. Since the paper was published, new evidence has been obtained to show that it matches object oriented system design. In particular, it works for languages in which exceptions are classes: nearly all C++ and Java tutorials discuss the hierarchical structuring of exception classes. It remains to be seen how resolution procedures can be used for parameterised exceptions: although the paper [7] claims that the only way is for the former to be application-dependent, we believe that this problem can be solved by a proper choice of default values and by linguistic features which could allow dynamic binding based on procedure profile analysis and conformance rule checks. The problem of designing the corresponding handlers in appropriate ways is getting more complex because these handlers are ordered, and any handler for any exception must be able to do all recovery which is done by the handlers for its descendant exceptions. On the other hand, these handlers can be very complex because they are to provide cooperative action recovery. To make this approach practical and routine, clear and strict engineering rules should be described which would explain how resolution should be used and how handlers should be designed. Although it is clear that software fault tolerance should be designed together with application software, the resolution tree can be built only when the designer is clear about the kinds of errors (and faults) that are to be tolerated, and the handlers can be implemented only after the tree has been built.

Using atomic actions with exception resolution should be made more object-oriented in the future. The right way is to regard actions as instances of special classes [17, 20]. The problems to be addressed are: overriding and inheriting handlers and the resolution tree, extending the resolution tree, reusing the tree and/or handlers (e.g. what happens if we add a new exception, or override the old one; how we can re-order exceptions with minimum handler re-design; whether we should re-design and override all handlers on the tree path between the newly inserted one and the root; etc.).

We believe that the approach and implementations discussed can be used directly for introducing the CA action concept [20] into existing languages. In particular, in any centralised synchronous implementation the functions of the action manager can be extended for it to be the SR-process and to resolve concurrent exceptions.

Many object oriented systems allow their own behaviour to be reflected upon and changed by changing its representation. The main technique relies on using meta-object protocols [11], where each object is associated with a meta-object representing different aspects of the object behaviour. This opportunity to program specialised cooperating meta-objects can be of great use for exception resolution because if we can reflect upon raising exceptions and processes entering and leaving exception contexts, then we can move virtually the entire resolution, together with the synchronisation involved in the action completion, to a meta-level, which is a reasonable thing to do because the resolution protocol is not functional software. This can allow system programmers to design meta-object protocols relying on and choosing from resolution trees, lattices or other partial exception ordering; the decentralised or centralised implementation of resolution; different agreement protocols (depending on fault assumptions and the current system state). Meta-object protocols can change the resolution tree location or replicate it, for it to be used by several processes/sites, or make the resolution procedure itself fault tolerant. Another protocol can deal with parameterised exceptions because, generally speaking, the entire object state is accessible from the meta-level and any required actual parameters can be picked from there. Besides, if there is an additional feature allowing asynchronous program interrupts, they can be used from within the meta-level, and synchronous schemes can be transformed into asynchronous ones (and vice versa). All this can be done transparently for functional application software and changed (adjusted) in the run-time.

Nowadays exception handling is obviously a very important part of a vast majority of practical languages. Many distributed and concurrent systems are programmed using these languages. A demand for concurrent/distributed exception handling, as a vital step in introducing forward error recovery into these systems, is recognised by many researchers because it would make this recovery simpler, uniform and less error prone. The main purpose of this paper is a thorough discussion of this subject. We believe that this feature should be regarded as part of the underlying support for future languages and systems.

Acknowledgements. This research is supported by DeVa Basic ESPRIT Project and my thanks go to all my colleagues in this project. I am grateful to A.Burns, A.Wellings and S.Mitchell (York University) for the fruitful discussions, criticism and sharing with me many ideas which helped me tremendously, and to A.Zorzo (Newcastle University) for helping me with Java scheme design.

References

[1] Ada 95, Information technology - Programming languages - Ada. Language and Standard libraries. ISO/IEC 8652:1995(E), Intermetrics, Inc., 1995.

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

[3] A. Burns and A.J. Wellings, *Real-time Systems and their Programming Languages* (Addison-Westley, New York, 1989).

[4] R.H. Campbell and B. Randell, Error recovery in asynchronous systems, *IEEE Trans. Softw. Engng.* SE-12, 8 (1986) 811-826.

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

[6] D.T. Huang and R.A. Olsoon, An exception handling mechanism for SR, *Computer Languages.* 15, 3 (1990) 163-176.

[7] 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.

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

[9] Java, The Java Language Specification. Version 1.0 Beta, Sun Microsystems, Inc., 1995.

[10] Java, Java Remote Method Invocation Specification. Revision 1.42, Sun Microsystems, Inc., 1997.

[11] G. Kiczales, J. des Rivières and D.G. Bobrow, *The art of the metaobject protocol* (The MIT Press, Cambridge, Massachusetts, 1991).

[12] E. Klingerman and A.D. Stoyenko, Real-Time Euclid: A Language for Reliable Real-Time Systems, *IEEE Trans. Softw. Engng.* SE-12, 9 (1986) 941-949.

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

[14] B. Meyer, *Eiffel: The Language* (Englewood Cliffs, N.J., Prentice Hall, 1992).

[15] G. Nelson, *System Programming with Modula-3* (Prentice Hall, 1991).

[16] A. Romanovsky and L. Strigini, Backward error recovery via conversations in Ada, *Softw.Engng. Journal.* 10, 8 (1995) 219-232.

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

[18] A. Romanovsky, Practical exception handling and resolution in concurrent programs, *Computer Languages* 23, 1 (1997) 43-58.

[19] A.J. Wellings and A. Burns, Implementing Atomic Actions in Ada95, *IEEE Trans. Softw. Engng.* SE-23, 2 (1997) 107-123.

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

[21] S.S. Yau and R.C. Cheung, Design of Self-Checking Software, in: Proc. *Int. Conference on Reliable Software.* California, USA (1975) 450-457.