Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation

Jie Xu, Alexander Romanovsky and Brian Randell

Department of Computing Science
University of Newcastle upon Tyne, Newcastle upon Tyne, UK
{jie.xu, alexander.romanovsky, brian.randell}@newcastle.ac.uk

In Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS '98) , Amsterdam, The Netherlands, 26-29 May 1998,! ,!pp. 12-21
IEEE Computer Society Press,!1998

© 1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation

Jie Xu, Alexander Romanovsky and Brian Randell

Department of Computing Science
University of Newcastle upon Tyne, Newcastle upon Tyne, UK
{jie.xu, alexander.romanovsky, brian.randell}@newcastle.ac.uk

Abstract

Exception handling in concurrent and distributed programs is a difficult task though it is often necessary. In many cases traditional exception mechanisms for sequential programs are no longer appropriate. One major difficulty is that the process of handling an exception may need to involve multiple concurrent components that are cooperating in pursuit of some global goal. Another complication is that several exceptions may be raised concurrently in different nodes of a distributed environment. Existing proposals and actual concurrent languages either ignore these difficulties or only cope with a limited form of them. This paper attempts a general solution, developed especially for distributed object systems, starting from a conceptual model, together with algorithms for coordinating concurrent components and resolving multiple exceptions, through to an actual system implementation. An industrial production cell is chosen as a case study to demonstrate the usefulness of the proposed model and algorithms. A system that supports coordinated atomic actions and exception resolution is implemented in distributed Ada 95 and examined through several performance-related experiments.

Key Words — Concurrent programs, coordinated exception handling, distributed object systems, exception resolution, nested atomic actions.

1: Introduction

Concurrent and distributed computing systems often give rise to complex asynchronous and interacting activities. The provision of exception handling and error recovery becomes very difficult in such circumstances [12]. One way to control the entire complexity, and hence facilitate error recovery, is to somehow restrict interaction and communication. Atomic actions are one way employed in both research and practice to achieve this goal. Most of the existing schemes for exception handling in concurrent systems use the concept of an atomic action as a unit of error confinement, though there is no clear consensus on how to handle exceptions when asynchronous activities occur [4][5][8][14].

Many new architectural developments in the area of distributed computing systems are, to some extent, object-based or object-oriented (OO). The OO technique, with its modularity, flexibility and reusability features, can be usefully exploited for handling complexity and dependability issues of a distributed system. The concept of *Coordinated Atomic Actions* (or CA actions) [16], as a generalized form of the basic atomic action structure, has been developed especially for distributed object systems to provide a mechanism for the strict enclosure of interaction and recovery activities. CA actions are a natural structuring unit for performing complex exception handling in distributed object systems.

Exception mechanisms used in sequential programs cannot be applied to complex concurrent software without appropriate change and adjustment. A distributed system may contain many components, and several components may be involved in a cooperative computation. Once an exception occurs in one of these components, not only the user of the computation, but also the other components involved need to be informed of the exception so as to enable a coordinated recovery activity. Moreover, different components may raise different exceptions and the exceptions may be raised simultaneously. This can further complicate the process of exception handling. (A more detailed discussion of the necessity of coping with concurrent exceptions can be found in our previous research [13].) The work in [4] argues that for handling exceptions in distributed systems, a hierarchy-based approach is essential in order to find a higher-order exception that can "cover" all the exceptions concurrently raised**.** This further requires a distributed scheme for determining the proper recovery strategy and for involving all the related components in the recovery activity.

In this paper we first establish a conceptual exception model for distributed object systems, using CA actions as a structuring unit. Two efficient distributed algorithms (and their supporting mechanisms) are then developed for coordinating concurrent exception handling. The correctness of the first (major) algorithm is proved and its communication complexity is shown to be lower than existing proposals such as the algorithm in [4] and our original algorithm presented in [13]. An industrial case study is chosen to demonstrate the practical usefulness of the proposed model and algorithms. Realistic system implementation is provided in distributed Ada 95 and several performance-related experiments are carried out to assess this implementation.

2: Exception Handling and CA Actions

We consider a *distributed object system* consisting of nodes connected by a communication network. The objects that run on network nodes communicate with each other by message passing. *Exception handling* is viewed here as a general mechanism for coping with exceptional system conditions or *errors* caused by *hardware faults* or *software faults*. Hardware faults include transient faults in, or crashes of, nodes or the communication network, while software faults are mainly due to incorrect specification, poor program design and implementation. From either type of faults, erroneous information may spread through communication channels and thus affect multiple nodes.

In principle, fault-tolerant software detects errors by various detection mechanisms, such as executable assertions and memory-protection checks, and employs *error recovery* techniques to restore normal computation. Forward error recovery is based on the use of redundant data that repairs the system by analyzing the detected error and putting the system into a correct state. In contrast, backward error recovery returns the system to a previous (presumed to be) error-free state without requiring detailed knowledge of the errors.

2.1: Exception Handling in Concurrent and Distributed Systems

An exception (handling) mechanism is a programming control structure that allows programmers to describe the replacement of the normal program execution by an exceptional execution when occurrence of an exception is detected [5]. For any given exception mechanism, *exception contexts* are defined as regions in which the same exceptions are treated in the same way; often these contexts are blocks or procedure bodies. Each context should have a set of associated *exception handlers*, one of which will be called when a corresponding exception is *raised*. There are different models for changing the control flow, but the *termination* model is most popular. This model assumes that when an exception is raised, the corresponding handler copes with the exception and completes the program execution. If the handler for this exception does not exist in the context or it is not able to recover the program, then the exception will be propagated. Such *exception propagation* often goes through a chain of procedure calls or nested blocks where the handler is sought in the exception context (containing the context that raised or propagated the exception.)

Exception handling and the provision of fault tolerance are more difficult in concurrent and distributed systems. For example, there would be no problem in sequential programs if a client object tried to get data from an empty queue — an interface exception would be signalled by the server object. However, concurrent access to server objects, permitted by concurrent systems, greatly complicates such exceptional situations. If two clients attempt to access a non-empty queue concurrently (but the queue contains only one element), one of them may surprisingly receive an interface exception that blames it for the use of an empty queue! A more serious complication is that several exceptions can be raised concurrently in multiple concurrent activities [4][13]. Obviously, proper exception handling has to involve multiple interacting components and additional mechanisms for coordinating multiple objects are needed.

Exception propagation in concurrent programs may not simply go through a chain of nested callers, but can require an extra dimension of propagation. In the case of nested atomic actions, an exception may need to be propagated upward to the enclosing action from a nested action. Since the enclosing action can involve more components than the nested action, the exception may therefore also need to be propagated to all the components of the enclosing action in order to start a joint recovery activity. Unfortunately, no known language or systems provides appropriate support for such two-dimensional exception propagation.

Physical distribution of computing further complicates the coordination of multiple concurrent components. In a distributed system, each node may possess a separate memory; as a consequence, software segments executing on different nodes will reside in disjoint address spaces and so must communicate by the exchange of messages over relatively narrow bandwidth communication channels. The time of message passing is not negligible and the effect caused by the communication delay must therefore be taken into account.

2.2: Atomic Actions

Interacting activities in concurrent systems must be controlled carefully in order to prevent erroneous information from spreading throughout the whole system. A structuring concept which assists the confinement of interacting activities is that of *atomic actions*.

> *The activity of a group of components or objects constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity [2].*

In 1986 Campbell and Randell [4] developed a systematic approach to exception handling within an atomic action (or *conversation*) that encloses interaction of a group of processes (or execution *threads*) [12]. A set of exceptions is associated with each action. Each thread participating in the action has a set of handlers for some or all of these exceptions. When an exception is raised, the appropriate handlers (for the same exception in all participating threads) will be initiated and these handlers will be jointly responsible for recovering the system cooperatively. This means that interacting threads cooperate not only when they execute the normal program functions but also when they recover the program. Campbell and Randell introduced a mechanism for resolving multiple exceptions raised concurrently based on

the *exception tree* concept — an exception tree includes all exceptions associated with an atomic action and imposes a partial order on them in such a way that the higher exception has a handler that is intended to handle any lower level exception.

The coordinated atomic action concept is a generalized form of the basic atomic action structure and presents a general technique for achieving fault tolerance in distributed object systems by integrating conversations, transactions (that ensure consistent access to shared objects) and exception handling into a uniform structuring framework [16]. CA actions take two kinds of concurrency into account: *cooperating* and *competing*. Several execution threads can be designed collectively and executed concurrently in order to achieve certain global goal. But they must cooperate within the boundaries of a CA action. Competitive concurrency may also exist in such systems since separately designed threads can compete for the same system resources (i.e. objects). More precisely, CA actions use conversations as a mechanism for controlling concurrency and communication between threads that have been designed to cooperate with each other. Shared *external objects* are controlled by an integrated transaction mechanism that guarantees the atomicity property [10]. In other words, the transaction mechanism must ensure that during the execution of a CA action, concurrent access to its external objects from other actions and threads is in effect equivalent to absence of the interactions between the CA action and the rest of the system.

Figure 1 shows an example in which two threads enter a CA action synchronously. Within the action the threads communicate with each other and cooperate in pursuit of some common goal. However, during the execution of the CA action, an exception e is raised by one of the threads. The other thread is then informed of the exception and both threads transfer control to their respective handlers for this exception H_1 and H_2 which attempt to perform forward error recovery. The effects of erroneous operations on external objects are repaired by putting the objects into new correct states so that the CA action is able to exit with an acceptable overall outcome. (As an alternative to performing forward recovery, the two participating threads could undo the effects of operations on the external objects, roll back and then try again, possibly using diversely-designed alternates, if the aim is to provide means of tolerating residual design faults.)

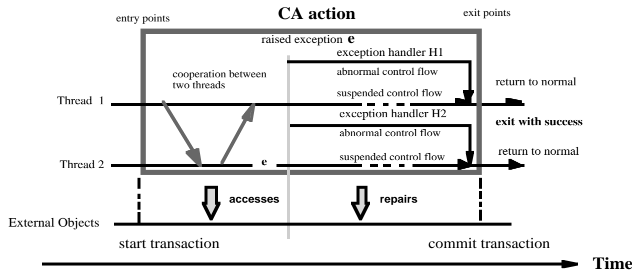

Figure 1 Error recovery performed by a CA action.

3: Coordinated Exception Handling and Resolution: Model and Algorithms

In this section we first describe a basic model for coordinated exception handling and then discuss the details of a distributed algorithm for propagating exceptions between concurrent threads and for resolving exceptions concurrently raised. A simple algorithm is also developed for signalling exceptions over nested actions.

3.1: Basic Model for Exception Handling and Resolution

We model the dynamic structure of a distributed OO system as a set of interacting CA actions. A CA action provides a mechanism for performing a group of operations on a set of objects. These operations are performed cooperatively by one or more *roles* executing in parallel within the CA action. The interface to a CA action specifies the external objects that are to be manipulated by the CA action and the roles that are to manipulate these objects. In order to perform a CA action, a group of execution threads must come together and agree to perform each role in the CA action concurrently with one thread per role. CA actions can be properly nested and exceptions may be propagated over nesting levels.

Exception Declaration: For a given CA action, there are two types of exceptions: ones that are totally internal to the CA action and that when raised are to be handled by its own handlers, and others that are known in and are to be signalled to its environment (e.g. its caller or the enclosing action).

All exceptions, $\boldsymbol{e} = \{e_1, e_2, e_3, ...\}$, that are raised within a CA action must be declared within the action definition. The corresponding exception handlers are associated with respective roles that the participating threads are to perform. The exceptions, $\varepsilon = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, ...\}$, that are signalled from a CA action to its environment should be specified in the interface to the CA action. These exceptions are signalled in order to indicate that, though internal exception handling might have been (unsuccessfully) attempted, an unrecoverable exceptional condition has occurred within the action, and/or only incomplete results can be delivered by the action. For a nested CA action and its direct-enclosing action, the definitions of $\boldsymbol{e}$ and ε are fully recursive, namely,

$$\varepsilon_{nested} \subseteq \boldsymbol{e}_{enclosing}$$

There are two special exceptions μ and f in ε. An `undo` exception, μ, implies that the action has been aborted and all of its effect have been undone. Since `undo` is not always possible, a `failure` exception, f, will indicates that the action has been aborted but that its effect may have not been undone completely.

Exception Handling and Propagation: When a thread enters the action to play a specified role, it enters the related exception context. Some or all of the participating threads may later enter nested CA actions.

Since the nesting of CA actions causes the nesting of exception contexts, each participating thread of the nested action must be associated with an appropriate set of handlers. Exceptions can be propagated along nested exception contexts, namely the chain of nested CA actions. Three terms are used here to clarify the route of exception propagation: an exception e_i in $\boldsymbol{e}$ is *raised* by a role within a CA action, other roles of the same action are then *informed* of the exception e_i and, if handling the exception within the CA action is not fully successful, a further exception ε_j in $\boldsymbol{\varepsilon}$ will be *signalled* from a nested action to its enclosing action (see Figure 2).

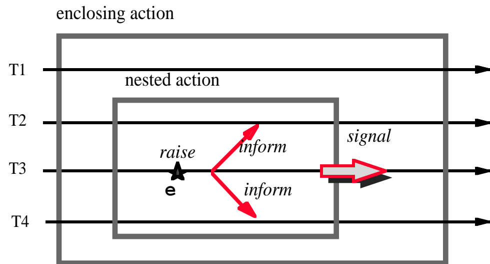

Figure 2 Exception propagation over nesting levels.

There are at least two ways of signalling an exception from a nested action to its enclosing action. One possibility is that a "leading" role has the responsibility for signalling an agreed exception to the enclosing action. Another approach however adopts a more distributed strategy: each role of the nested action is responsible for signalling its own exception. These exceptions should be the same but in fact might be different. Because an action in our model is required to have the ability of handling concurrent exceptions, the exceptions concurrently signalled from the nested action will be handled simply as if they are concurrently raised in the enclosing action.

Control Flow: The termination model of control flow is used here — in any exceptional situations, handlers take over the duties of participating threads in a CA action and complete the action either successfully or by signalling an exception ε_j to the enclosing action.

External Objects: The effect that a CA action in our system model can be observed only through the committed state of some external objects. Once an exception is raised within the CA action and hence error recovery is requested, the related external objects must be treated explicitly and in a coordinated fashion, the aim being to leave them in a consistent state, if at all possible. The standard way of doing this in transaction systems is by restoring the objects to their prior states. However, an exception does not necessarily cause restoration of all the external objects. (Indeed, external objects, particularly real ones in the computers' environment, might not be capable of state restoration.) Appropriate exception handlers may well be able to lead such objects to new valid states. But when it is detected that one or more external shared objects have failed to reach a correct state, a `failure` exception f must be signalled to the enclosing CA action in the hope that it may be able to handle the situation.

Exception Resolution: If several exceptions are raised at the same time, one simple method for resolving the exceptions is to prioritize them. The disadvantage of this scheme is that it does not allow representation of situations where the concurrently raised exceptions are merely manifestations of a different, more complicated, exception. To provide a more general method, an *exception graph* representing an exception hierarchy can be utilized. If several exceptions are raised concurrently, then the multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions [4]. In principle, each CA action should have its own exception graph.

3.2: Exception Graphs

The exception tree concept (first proposed in [4]) is a simplified form of specifying the relationship between multiple exceptions. We have however found that in practice an exception hierarchy often has a more complicated form than a simple tree. We formalize a general form below, called exception graphs.

An exception graph is a directed graph $G(E, R)$ where the exception set $E = \{e_1, e_2, ..., e_n\}$. Each exception $e_i \in E$ is represented by a node and each directed edge $(e_i, e_j) \in R$ represents a simple relationship in which $e_i \in E$ is the direct high-level node, or parent node of $e_j \in E$. We define the in-degree of node e_i, $d_{in}(e_i)$, as $|\Gamma^{-1}(e_i)|$ and the out-degree $d_{out}(e_i)$ as $|\Gamma(e_i)|$, where $\Gamma(e_i) = \{e_j : (e_i, e_j) \in R\}$ and $\Gamma^{-1}(e_i) = \{e_j : (e_j, e_i) \in R\}$. (For example, in Figure 3 $d_{in}(e_1) = 2$ and $d_{out}(e_1) = 0$.)

For a given $G(E, R)$, there may exist three types of nodes. Nodes with $d_{out}(e_i) = 0$ represent primitive exceptions that cover no other exceptions. Internal nodes with $d_{in}(e_i) \neq 0$ and $d_{out}(e_i) \neq 0$ represent resolving exceptions that cover some other exceptions. The node with $d_{in}(e_i) = 0$, called the root of $G(E, R)$, represents a special *universal* exception. The raising of a universal exception usually leads to the signalling of an `undo` or `failure` exception to the enclosing action. Figure 3 shows a four-level exception graph containing three primitive exceptions e_1, e_2, e_3 at the level 0.

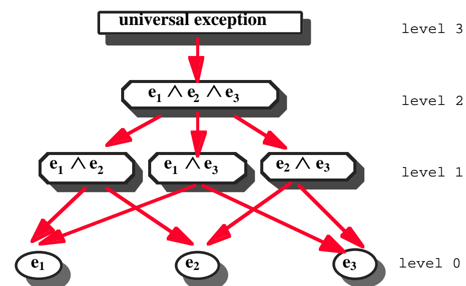

Figure 3 Example of a four-level exception graph.

In Figure 3 the resolving exception $e_1 \wedge e_2$ at level one will be raised when e_1 and e_2 are raised concurrently. Similarly, the exception $e_1 \wedge e_2 \wedge e_3$ at level two will be raised in order to cover all the three primitive exceptions. This resolving exception may still be handled by the current action, or otherwise the universal exception at level three will be further raised. In general, an $(n+1)$-level exception graph can be defined with n primitive exceptions at level 0. The first level can contain up to $n \times (n - 1)/2$ resolving exception nodes. Level two could consist of up to $n \times (n - 1)(n - 2)/6$ nodes, and so on. Level n-1 has only one resolving exception that covers all the primitive exceptions when level n-2 may have at most n exception nodes. This general definition makes the automatic generation of an exception graph possible. However, for an actual application a simplified exception graph may be required from the space and performance point of view.

3.3: Concurrent Exception Propagation and Resolution

Assumptions and Definitions: For a given CA action it is assumed that each participating thread knows the set of all other threads participating in the action and uses the same exception graph which is statically declared. Every thread has a name list for the nested actions it is to enter. For a specified thread the currently innermost action is called the *active* CA action. Let *CA-action* be the outermost (or top-level) CA action. We define $G_{CA\text{-}action}$ as the group of threads $\{T_1, T_2, ..., T_i, ..., T_j, ...\}$ participating in *CA-action*, where each thread T_i has a unique identifier and the threads are ordered (e.g. thread names and the lexicographic ordering could be used).

During the execution of the algorithm, a thread T_i may be in one of the following states (denoted by $S(T_i)$): `N` = Normal, `X` = Exceptional (if an exception was raised in T_i), or `S` = Suspended (if T_i has to stop normal computation due to exceptions raised by other threads).

Let A be the active action of T_i and G_A be the corresponding set of participating threads. We assume that each T_i keeps the following data structures:

list LE_i — records exceptions that have been raised, and suspended states, `S`, of threads that have halted normal computation;

stack SA_i — stores names of the nested actions T_i is currently in.

It is also assumed that application-related message passing is treated independently, and only the following specific messages are used in our algorithm:

`Exception`(A, T_i, E) is sent by thread T_i to all the other threads of action A when an exception E is raised by T_i;

`Suspended`(A, T_i, `S`) is sent by the thread T_i that does not raise an exception but has received an `Exception` or `Suspended` message from another thread, where `S` indicates T_i is in the "Suspended" state;

`Commit`(A, E) is sent by a chosen thread in action A to all the other threads after it completes resolution of exceptions, where E is the resolving exception. A corresponding handler for E will be called by each thread after it receives this `Commit` message.

It is further assumed that an exception in an enclosing action will stop or abort any activity of its nested actions (including any nested resolution in progress and execution of any handlers.)

In the interests of simplicity and brevity, our algorithm is designed not to tolerate node or communication line crashes, though a fault-tolerant version of this algorithm would be non-trivial, especially when addressing omissions and Byzantine faults. The proposed algorithm attempts to handle certain forms of software bugs, transient hardware faults and hardware design faults, but the disastrous crash of a processing node or a communication line must be masked at the appropriate underlying or hardware level, e.g. by using modular redundancy. (Our model described in section 3.1 is however general, and it is supposed to cope with exceptions that may be caused by various types of faults.)

The Main Algorithm: Our algorithm assumes the existence of general support mechanisms including FIFO message sending/receiving between threads/objects and calls to abortion handlers. In addition, "< >" indicates a data item with one or more elements, "A*" is the active action of thread T_j, "$\rightarrow$" stands for "put in", and "$\Rightarrow$" stands for "sent to" in the description of our algorithm.

Figure 4 illustrates how the algorithm works when two exceptions `E1` and `E2` are raised concurrently in several nested CA actions. The proposed algorithm first informs all four participating threads of the two exceptions by message passing between those threads. Secondly, the algorithm aborts two nested actions because of exception `E1` that occurred outside the nested actions. (During the abortion, a further exception `E3` is signalled to the outermost enclosing action.) Finally, the algorithm determines a resolving exception `E` that covers both `E1` and `E3` and starts the corresponding handler for `E`.

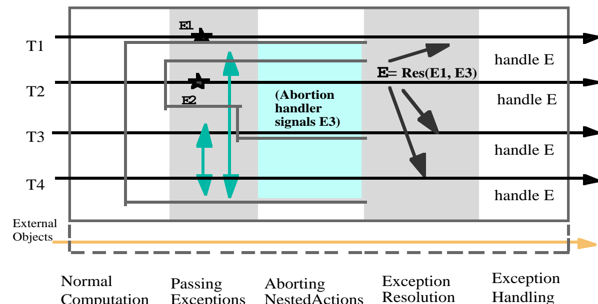

Figure 4 Exception propagation and resolution.

Algorithm:

For any T_i, $S(T_i)$: = N; and empty LE_i, SA_i;
loop
if T_i enters A **then**
  $<A> \rightarrow SA_i$; consume messages having arrived;
end if;
if T_i completes A **then**
  delete last element in SA_i;
  $S(T_i)$: = N if end A with success, or otherwise $S(T_i)$: = X;
  leave A;   //synchronously
end if;
if E_i is raised in T_i **then**
  $S(T_i)$: = X; $<A, T_i, E_i> \rightarrow LE_i$;
  `Exception`$(A, T_i, E_i) \Rightarrow$ all T_j in G_A;
  inform external objects (used by T_i in A) of E_i;
end if;
if T_i receives `Exception`(A^*, T_j, E_j) **or**
  `Suspended`$(A^*, T_j,$ S$)$ **then**
  if A^* contains or equals A **then**
   $<A^*, T_j, E_j>$ or $<A^*, T_j, S> \rightarrow LE_i$;
   $E_j \Rightarrow$ uninformed external objects (used by T_i in A^*);
   if A^* contains A **then**
    abort all nested actions until A^*;
    delete the elements in SA_i until $<A^*>$;
    remove items but $<A^*, T_j, E_j>$ or $<A^*, T_j, S>$ in LE_i;
    if E_{ab} is raised by the abortion handler **then**
     $S(T_i)$: = X; $<A^*, T_i, E_{ab}> \rightarrow LE_i$;
     `Exception`$(A^*, T_i, E_{ab}) \Rightarrow$ all T_j in G_{A^*};
    else $S(T_i)$: = S; $<A^*, T_i, S> \rightarrow LE_i$;
     `Suspended`$(A^*, T_i,$ S$) \Rightarrow$ all T_j in G_{A^*};
    end if;
   else **if** $S(T_i)$ = N **then**   *//here $A^* = A$*
      $S(T_i)$: = S; $<A^*, T_i, S> \rightarrow LE_i$;
      `Suspended`$(A^*, T_i,$ S$) \Rightarrow$ all T_j in G_{A^*};
     end if;
   end if;
  else retain the message until T_i enters A^*;
  end if;
end if;
if T_i has all E_i, `or` state S, of other threads within A **and** T_i has the biggest identifying number among threads with the state X **then**
  resolve exceptions in LE_i;
  //find E in the exception graph
  `Commit`$(A, E) \Rightarrow$ all T_j in G_A;
  empty LE_i and handle E;
end if;
if T_i receives `Commit`(A^*, E) **then**
  if $<A^*>$ = the top element in SA_i **then**
   empty LE_i and handle E;
  end if;
end if;
end loop

Correctness and Communication Complexity: For a specific distributed system, we assume that the following time elements can be bounded if no a fault occurs. Let T_{mmax} be the maximum time of message passing between two concurrent execution threads in the system; T_{reso} be the upper bound of the time spent in resolving current exceptions, T_{abort} be the maximum possible time for a thread to abort one nested CA action, n_{max} be the maximum number of nesting levels of CA actions (if no nesting, then $n_{max} = 0$), and Δ_{max} be maximum possible time of handling a (resolving) exception. We now show that no deadlock is possible in our proposed algorithm. (For complete proofs, see [18].)

Lemma 1: Consider N execution threads that interact within nested CA actions. For any thread T_i, if it reaches the state X (exceptional) or S (suspended), it will complete exception handling ultimately in at most $\boldsymbol{T}$, where

$$\boldsymbol{T} \le (2n_{max} + 3)T_{mmax} + n_{max}T_{abort} + (n_{max} + 1)(T_{reso} + \Delta_{max})$$

By Lemma 1, we know that any thread will complete exception handling within a finite time bound. Therefore, deadlock during the process of exception handling will be impossible while executing the proposed algorithm. However, in order to prove the entire correctness of the proposed algorithm, we must show that any resolving exception is a proper cover of the multiple exceptions that have been raised concurrently so far.

Lemma 2: For a given CA action A, if no exception is raised in any containing action of A, then no more new exceptions will be raised within A once the exception resolution starts.

Lemma 3: Consider N execution threads that interact within nested CA actions. If multiple exceptions are raised concurrently, an ultimate resolving exception that covers all the exceptions will be generated by the proposed algorithm.

From Lemmas 2 and 3, we know that a resolving exception will always cover all the concurrently raised exceptions. Any further exception will cause the abortion of any effect of previous resolutions and trigger the new exception resolution. Because deadlock is not possible, the final resolving exception will be raised in the end. We therefore have the conclusion below.

Theorem 1: The proposed algorithm is deadlock-free and always performs correct exception resolution.

Note that the algorithm in [4] is of complexity $O(n_{max} \times N^3)$. Our previous algorithm in [13] could use $n_{max} \times 3N \times (N-1)$ messages. Our new algorithm is less complex and requires exactly $n_{max} \times (N^2 - 1)$ messages because 1) the number of messages for informing exceptions or suspended states is reduced and no reply is required, and 2) only one thread (rather than all the threads) resolves multiple exceptions and only one thread needs to send the `Commit` message. In the interest of fault tolerance, the algorithm can be easily extended to the use of a group of threads that are responsible for performing

resolution and producing the `Commit` messages. But this only contributes a constant factor to its total message complexity.

3.4: Exception Signalling

The algorithm described in section 3.3 ensures that a resolving exception e_r is identified and all the threads start handling this exception by invoking the appropriate handlers. However, such exception handling may be only partially successful, or fail completely. In these cases a thread must signal a further exception ε_j to the enclosing CA action. Following our model introduced in section 3.1, participating threads of a nested action may signal different exceptions, but they must signal the same exception μ or f if exception handling fails. We therefore need a further algorithm for coordinating those exceptions to be signalled. Let A be the active action of T_i and G_A be the corresponding set of participating threads. We assume that each thread T_i has a list and uses a specific message:

list $listSignal_i$ — records exceptions that are to be signalled by the participating threads of action A (if a thread is to signal no exception, ϕ will be recorded in the list instead);

`toBeSignalled`(T_i, ε) is sent by T_i to all threads of action A when an exception ε is to be signalled by it, where $\varepsilon \in \{\phi, \varepsilon_1, \varepsilon_2, \varepsilon_3, ..., \mu, f\}$.

Algorithm:

//after handling the resolving exception E
For any T_i of A, empty $listSignal_i$ and `FALSE` $\Rightarrow$ *undo*;
loop
if T_i is to signal ε **then**
  $\langle T_i, \varepsilon \rangle \rightarrow listSignal_i$; // $\varepsilon \in \{\phi, \varepsilon_1, \varepsilon_2, \varepsilon_3, ..., \mu, f\}$
  `toBeSignalled`$(T_i, \varepsilon) \Rightarrow$ all T_j in G_A;
end if;
if T_i receives `toBeSignalled`(T_j, ε) **then**
  $\langle T_j, \varepsilon \rangle \rightarrow listSignal_i$;
end if;
if $|listSignal_i| = |G_A|$ **then**
  //received all exceptions of other threads to be signalled
  switch$(listSignal_i)$
   case 1: no μ or f in $listSignal_i$
    T_i signals ε of $\langle T_i, \varepsilon \rangle$;
   case 2: μ but no f in $listSignal_i$
    if *undo* = `TRUE` **then**
     T_i signals μ;
    else
     empty $listSignal_i$ and `TRUE` $\Rightarrow$ *undo*;
     T_i executes appropriate `undo` operations;
     T_i is ready to signal a new exception ε;
    end if;
   case 3: f in $listSignal_i$
    T_i signals f;
end if;
end loop

The correctness of the algorithm is obvious. In the case that neither μ nor f is to be signalled by any participating thread, no coordination will be needed; each thread simply signals its own exception or signals no exception at all. If a thread is to signal the exception f, other threads just ignore their own exceptions and signal f instead. In these simple cases just $N \times (N-1)$ messages are required where $N = |G_A|$. In the complicated case that one thread is to signal the exception μ, all the threads must execute appropriate `undo` operations to ensure the removal of previous effects. Because some `undo` operations may fail, in this case f, rather than μ, must be signalled and messages must be passed again to guarantee that all threads signal the same f. However, after the second round of message passing no more operations will be executed and all threads will simply signal an appropriate exception μ or f. In the worst case, $2N \times (N-1)$ messages will be used.

This simple algorithm can be easily extended to cope with crashes of nodes or communication lines. The corrupted message or lost message can be simply treated as a `failure` exception and f is then recorded in $listSignal_i$. Therefore all the threads that run on fault-free nodes can still signal correct, coordinated exceptions to the enclosing action or the calling thread.

4: Case Study: A Production Cell

Many practical systems that interact with their environments are typically incapable of simple backward recovery. Exception handling and forward error recovery are the major means of improving the reliability of such systems. An industrial production cell model, taken from a metal-processing plant in Karlsruhe, Germany, was specified (and a controllable graphical Tcl/Tk simulator provided) as a challenging case study by the FZI in 1993 [9], within the German Korso Project. This case study has attracted wide attention and has been investigated by over 35 different research groups and universities. At Newcastle, [20] used CA actions as a structuring tool to design a control program for the model and implemented it in Java. The developed control program was then run against the simulator, demonstrating a good guarantee of functional and safety-related requirements.

The production cell consists of six devices: two conveyor belts — feed belt and deposit belt, an elevating rotary table, a press and a rotary robot that has two orthogonal extensible arms equipped with electromagnet (see Figure 5). These devices are associated with a set of sensors that provide useful information to a control program and a set of actuators through which the control program can have control over the whole system. The task of the cell is to get a metal blank (or plate) from its "environment" via the feed belt, transform it into the forged plate by using a press, and return it to the environment via the deposit belt.

More precisely, the production cycle for each blank is as follows: 1) if the traffic light for insertion shows green,

a blank may be added, e.g. by the blank supplier, to the feed belt from the environment, 2) the feed belt conveys the blank to the table, 3) the table rotates and lifts to the position where the robot can magnetize the blank, 4) the arm_1 of the robot picks the blank up and places it into the press, 5) the press forges the blank, 6) the arm_2 picks up the forged plate and places it on the deposit belt, and 7) if the traffic light for deposit is green, the deposit belt carries the plate forward to the environment where a container may be used, e.g. by the blank consumer, to store the forged pieces.

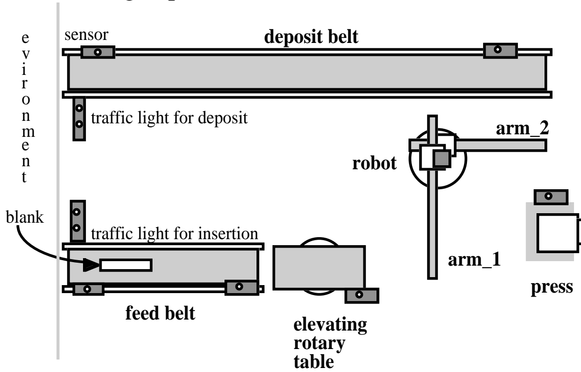

Figure 5 Production cell (top view).

The entire control program can be organized as a set of CA actions that coordinate the concurrent activities of the various devices. Figure 6 shows a set of nested CA actions for coordinating the activities of the table, the robot and the press.

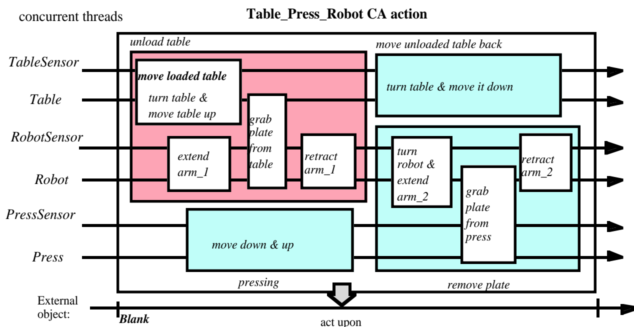

Figure 6 The `Table_Press_Robot` action.

For each (enclosing or nested) action, various exceptions are defined and an exception graph for resolution is declared. Take the `Move_Loaded_Table` action as an example: it may contain internal exceptions such as `vm_stop` (vertical table motor stops unexpectedly), `rm_stop` (rotation table motor stops), `vm_nmove` (vertical motor can't move), `rm_nmove` (rotation motor can't move), `s_stuck` (sensor(s) stuck at 0), `l_plate` (lost plate), `cs_fault` (control software fault(s)), `l_mes` (lost or corrupted message) and `rt_exc` (run time exceptions like underflow or overflow). An exception graph for this action is shown in Figure 7, permitting no more than two exceptions to be concurrently raised. For example, when both vertical and rotation motors fail, the exception graph will be searched and the resolving exception `dual_motor_failures` will be raised. Three or more concurrent exceptions as well as other undefined exceptions will not be resolved and will simply lead to the raising of the `universal` exception.

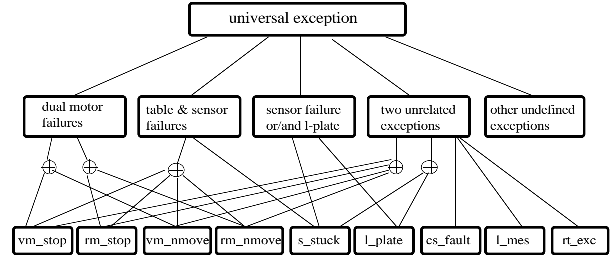

Figure 7 Exception graph for action `Move_Loaded_Table`.

Some internal exceptions can be handled within an action while other more serious exceptions are signalled to the enclosing action. For the `Move_Loaded_Table` action, four types of exceptions may be signalled to the `Unload_Table` action: `L_PLATE` (lost plate), `NCS_FAIL` (non-critical sensor failure), μ (undo) and f (failure without undo). These exceptions and the exceptions raised by the action `Unload_Table`, with exceptions signalled by actions `Extend_Arm1`, `Grab_Plate_from_Table` and `Retract_Arm1`, constitute the internal exceptions of the action `Unload_Table`. An exception graph can be structured in the form similar to the graph of Figure 7. Certain exceptions can be further signalled from the action `Unload_Table` to the action `Table_Press_Robot` action, including `T_SENSOR` (non-critical table sensor failure) and `A1_SENSOR` (arm_1's sensor failure), in the hope that the outermost action may handle them.

5: Implementation and Experimentation

We have recently accomplished a prototype implementation of the resolution mechanism and the related CA action supporting system in Ada 95 [1] (with the standard features of the Distributed Annex) in order to identify and tackle implementation and performance-related issues. We chose Ada 95 (the GNAT Ada 95 compiler, public release 3.04, on SunOS 5.4) because it is one of the few standard OO languages that have features for distributed programming. In addition, its elaborate features for concurrent programming, such as protected objects, asynchronous transfer of control and conditional entry calls, simplify the task of programming the run time support for CA actions and ensuring data consistency.

5.1: Prototype System Architecture

For a given CA action, each participating thread is located in its own node (or *partition* in the Ada 95 terminology), as shown in Figure 8. A portable subsystem for message passing is implemented that uses asynchronous remote procedure calls (without **out** parameters). Messages are first kept in the cyclic buffer of the receiver and then processed afterwards. A distributed

run-time system that supports CA actions is then established on the top of the message passing subsystem. Every partition has a copy of the run-time system, including the subsystems for exception handling and resolution where our new algorithms are realized. This basic CA action support offers the main CA action features: (nested) action entrances and exits, raising and signalling of exceptions, abortion of (nested) actions and calls to handlers. A protocol is also implemented for participating threads to leave a CA action synchronously.

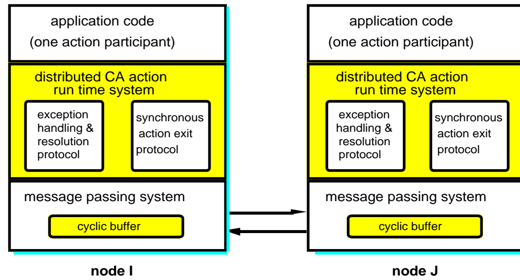

Figure 8 Prototype system architecture.

An exception may interrupt the normal computation or cause the abortion of the nested actions. We use the Ada 95 asynchronous transfer of control (ATC) to interrupt the action execution; the exception context of each CA action consists of the ATC blocks of its participating threads. The exception context in a thread has an abortion handler and a set of exception handlers. Every partition has a copy of the exception graph so as to ensure that the handlers for the same exception are called in all participating threads. The types common to all participating threads are declared in package `Pure`, which is used in compiling all packages; it includes names of all the exceptions, lists of all participating threads of each action, types declaring all object states and types of messages.

This prototype shows that the developed protocol fits well with the structure of modern distributed systems and is easy to implement: the entire implementation consists of about one thousand lines of code, 800 of which form the partition executive, and only 300 of those deal with exception handling and resolution. It demonstrates how to extend the basic CA action executive by just adding new functionalities to it. Our implementation method is general and can be easily applied to other systems, perhaps with minor adjustment to performance enhancement.

5.2: Performance Analysis

A simple application system was also developed for our experimental evaluation in which three threads take part in a CA action and two of them enter a further nested action. This system was executed in a loop (20 times) and the execution time measured. One of the experimental scenarios was as follows: one thread of the containing action raises an exception and the nested action has to be aborted. Another exception is raised by the abortion handler and the resolving exception (covering both exceptions) is then raised in all the threads. We varied three parameters, T_{mmax}, T_{abort} and T_{reso}, in order to examine the sensibility of the application execution time with respect to communications and exception handling. For example, let $T_{mmax} = 0.2s$, $T_{abort} = 0.1s$, and $T_{reso} = 0.3s$; the execution of the system will take about $94.36s$. In the tables of Figure 9 we present some of the experimental results with varying T_{mmax} and varying T_{reso} values.

T_{mmax}	Total Execution Time	T_{reso}	Total Execution Time
0.2	94.361391	0.3	94.361391
0.4	98.586050	0.5	98.352511
0.6	102.150904	0.7	102.547776
0.8	106.774196	0.9	107.164660
1.0	110.984972	1.1	110.338507
1.2	125.078084	1.3	114.729476
1.4	140.826807	1.5	118.928022
1.6	161.766956	1.7	122.483917
1.8	188.284787	1.9	127.117187
2.0	214.519403	2.1	131.816326
2.2	226.543372	2.3	135.123453
2.4	237.934833		
2.6	249.744183		
2.8	261.768559		

Figure 9 Results of performance-related experiments.

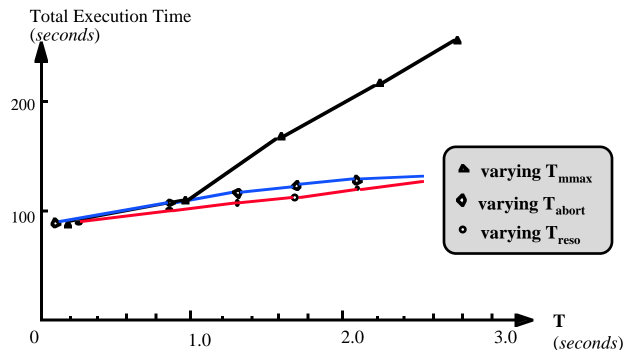

Figure 10 Effect on the total execution time.

The experimental data obtained are essentially consistent with the theoretical analysis presented in the previous sections. Figure 10 shows effect on the total execution time of the application system. When T_{mmax} is limited within $1.0s$, the cost of message passing has a minor impact on the total execution time. However, the execution time will increase dramatically once the time of message passing becomes longer than one second. On the other hand, with an increase in T_{reso} or T_{abort}, the total execution time has a very gentle and linear change. This demonstrates, at least in our prototype implementation, that the cost of message exchanges is still of the major concern, while concurrent exception handling does not introduce a high run-time overhead. (Another set of our experiments was performed in order to compare the CR algorithm in [4] with our algorithms. The related details can be found in [18].)

6: Conclusions

This paper has focused on the topic of exception handling in concurrent and distributed object systems. Our solutions are intended to be applicable to a wide set of OO languages and to practical systems that interact with their environments (e.g. the production cell application); such systems typically are incapable of simple backward recovery. The OO exception model developed in this paper extends and improves the models which may be found in sequential OO languages, and the non-concurrent models used in some concurrent OO languages.

How to correctly cope with nested CA actions in exceptional situations is a significant and delicate problem, especially in a distributed computing environment. In [4] the authors presented just a draft of their resolution algorithm, without discussing conditions and assumptions under which the algorithm may work. We have developed a mechanism that coordinates recovery measures used in both participating threads of nested actions and external atomic objects. New distributed algorithms have been designed and implemented to handle multiple exceptions raised concurrently and to signal exceptions over nested actions.

There has been relatively little work on implementations of coordinated error recovery in a distributed system. Implementations of distributed process-oriented conversations are discussed in [7][19]. The Arche language introduced in [6] allows the programmer to implement a resolution function. Such resolution is however only suitable for a limited form of concurrency. Wellings and Burns [15] have recently shown how Ada 95 can be used to implement atomic actions, but without addressing exceptions concurrently raised. There are only a few concurrent OO languages, such as Ada 95, Java and Guide [3], known to us that have exception handling features. Ada 95 has a limited form of concurrent-specific exception propagation — an exception will be propagated to both calling and called tasks if it is raised during the rendezvous. Exception handling in Java is similar to that in C++ without specially coping with concurrency-related (or multi-threaded) issues. Guide has one of the most object-oriented exception mechanisms among existing languages but its concurrency model is completely separated from the exception mechanism.

Future research directions would be in three primary areas. The first is the introduction of an appropriate linguistic mechanism for specifying nested CA actions in a distributed environment. Secondly, the exception graph concept requires more formal research into graph (automatic) generation, simplification, and efficient search. Finally, it is necessary to further implement a mechanism for supporting forward and backward error recovery of external atomic objects.

Acknowledgements

This work was supported by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa), and has been benefited greatly from discussions with a number of colleagues within the project, in particular R.J. Stroud, I. Welch, and A.F. Zorzo at Newcastle, A. Burns, S. Mitchell, and A. Wellings of the University of York, and J. Vachon of EFPL, Switzerland.

References

[1] "Ada. Language and Standard libraries" ISO/IEC 8652:1995(E), Intermetrics Inc., 1995.

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*, Prentice-Hall International, 1981.

[3] R. Balter, S. Lacourte, and M. Riveill, "The Guide language," *Computer J.* 37(6), pp.521-530, 1994.

[4] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Soft. Eng.*, SE-12(8), pp.811-826, 1986.

[5] F. Cristian, "Exception Handling and Tolerance of Software Faults," In *Software Fault Tolerance* (ed. M. Lyu), Wiley, pp.81-107, 1994.

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

[7] P. Jalote, "Using Broadcast for Multiprocess Recovery," In *Proc. IEEE ICDCS-6*, pp.582-589, 1986.

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

[9] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell"*, LNCS-891, Springer-Verlag, Jan. 1997.

[10] N.A. Lynch, M. Merrit, W.E. Wehil, and A, Fekete. *Atomic Transactions*, Morgan Kaufmann, 1993.

[11] R. Miller and A. Tripathi, "Issues with Exception Handling in Object-Oriented Systems," in *Proc. ECOOP'97* , pp.85-103, Finland, 1997.

[12] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Soft. Eng.*, SE-1(2), pp.220-232, 1975.

[13] A. Romanovsky, J. Xu, and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems," in *Proc. IEEE ICDCS-16,* pp.545-552, Hong Kong, 1996.

[14] D.J. Taylor, "Concurrency and Forward Recovery in Atomic Actions", *IEEE Trans. Soft. Eng.*, SE-12(1), pp.69-78, 1986.

[15] A.J. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95", *IEEE Trans. Soft. Eng.*, 23(2), pp.107-123, 1997.

[16] 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. IEEE FTCS-25*, pp.499-508, Pasadena, 1995.

[18] J. Xu, A. Romanovsky, and B. Randell, "Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation," Tech. Report, Dept. of Comput. Sci., Univ. of Newcastle, no.612, 1997.

[19] S.M. Yang and K.H. Kim, "Implementation of the Conversation Scheme in Message-Based Distributed Computer Systems," *IEEE Trans. Parallel and Distributed Sys.*, 3(5), pp.555-572, 1992.

[20] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud, and I. Welch, "Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study," to appear in *Software—Practice & Experience*, John Wiley & Sons.