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

J. Xu, A. Romanovsky, and *B. Randell*

Department of Computing Science

University of Newcastle upon Tyne, Newcastle upon Tyne, UK

Abstract

Exception handling in distributed and concurrent programs is a difficult task though it is often necessary. In many cases traditional 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 when they are cooperating to solve a global problem. 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 with from conceptual model, together with algorithms (and their correctness proofs) 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. The supporting system and the resolution mechanism are 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 [Campbell & Randell 1986]. One way to control the entire complexity, and hence facilitate error recovery, is to somehow restrict interaction and communication. Atomic actions are the usual tool 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 [Jalote & Campbell 1986][Taylor 1986].

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. Especially for distributed object systems the concept of *Coordinated Atomic Actions* (or CA actions) [Xu et al 1995], as a generalized form of the basic atomic action structure, has been developed to provide a mechanism for the strict enclosure of interaction and recovery activities. It is a natural decision to regard CA actions as one kind of 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 concurrent 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. In practice it is often difficult to interrupt the normal operations of the other components immediately after an exception has been raised, so new exceptions may be raised in these other components before they are informed of the initial exception. In fact, concurrently raised exceptions may be merely a manifestation in multiple components of a system-wide exception. A more detailed discussion of the necessity of coping with concurrent exceptions has been presented in our previous research [Romanovsky et al 1996]. This argues that for exception handling 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. An efficient distributed algorithm (and its supporting mechanisms including the exception signalling algorithm) is then developed for coordinating concurrent exception handling. The correctness of the algorithm is proved and its communication complexity is shown to be lower than existing proposals such as the algorithm in [Campbell & Randell 1986] and our original algorithm presented in [Romanovsky et al 1996]. An industrial case study is chosen to demonstrate the practical usefulness of the proposed model and algorithm. 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 Coordinated Atomic 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 fault 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 language control structure that allows programmers to describe the replacement of the normal program execution by an exceptional execution when occurrence of an exception (i.e. inconsistency with the program specification and hence an interruption to the normal flow of control) is detected [Cristian 1994]. 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 which 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 when a client object tries to get data from an empty queue — an interface exception will be signalled by the server object. However, concurrent access to server objects, permitted by concurrent systems, will greatly complicate such exceptional situations. When two clients attempt to have access to a non-empty queue concurrently (but the queue may have only an element left), one of them may surprisingly receive an interface exception which 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 [Campbell & Randell 1986][Romanovsky et al 1996]. Obviously, proper exception handling has to involve multiple interacting parts 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 situation 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 provides appropriate support for such two-dimensional exception propagation.

Physical distribution of computing further complicates 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 be therefore taken into account.

2.2 Atomic Actions

Interacting activities in concurrent systems must be controlled very carefully in order to avoid that erroneous information spreads through the whole system. An important 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 [Anderson & Lee 1980].*

Exception handling can be quite naturally incorporated into an atomic action framework because complex interaction between participating components of an atomic action can be coordinated within that action, including necessary operations for coordinating exception handling.

In 1986 Campbell and Randell [Campbell & Randell 1986] developed a systematic approach to exception handling within an atomic action, (or *conversation*) that encloses interaction of a group of processes (or execution *threads*) [Randell 1975]. A set of exceptions is associated with each action. Each thread participating in the action has the set of handlers for part 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 they will be 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, i.e. abnormal activities. 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 which is intended to handle any lower level exception.

The coordinated atomic (CA) action concept is a generalized form of the basic atomic action structure and it presents a general technique for achieving fault tolerance in concurrent software, especially for distributed object systems, by integrating conversations, transactions (that ensure consistent access to shared objects) and exception handling into a uniform structuring framework [Xu et al. 1995]. CA actions take two kinds of concurrency into account: *cooperating* and *competing*. Several execution threads can be designed collectively by different programmers (or teams) and executed concurrently in order to achieve certain joint and global goals. But they must cooperate within the boundaries of a CA action. Competitive concurrency may also exist in such systems, since two or more separately designed threads can compete concurrently 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 the associated transaction mechanism that guarantees the ACID properties (atomicity, consistency, isolation, durability [Lynch et al 1993]). In particular, objects that are external to the CA action, and can hence be shared with other actions concurrently, must be *atomic* and individually responsible for their own integrity. In a sense CA actions can be seen as a disciplined approach to using multi-threaded nested transactions, and to providing them with well-structured exception handling. For further details see [Randell et al 1997].

Figure 1 shows a simple example in which two threads enter a CA action asynchronously through different entry points. Within the CA 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 exception handlers 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 a successful outcome. (As an alternative to performing forward error recovery, the two participating threads could undo the effects of operations on the external objects, roll back and then try again, possible using diversely designed software alternates.)

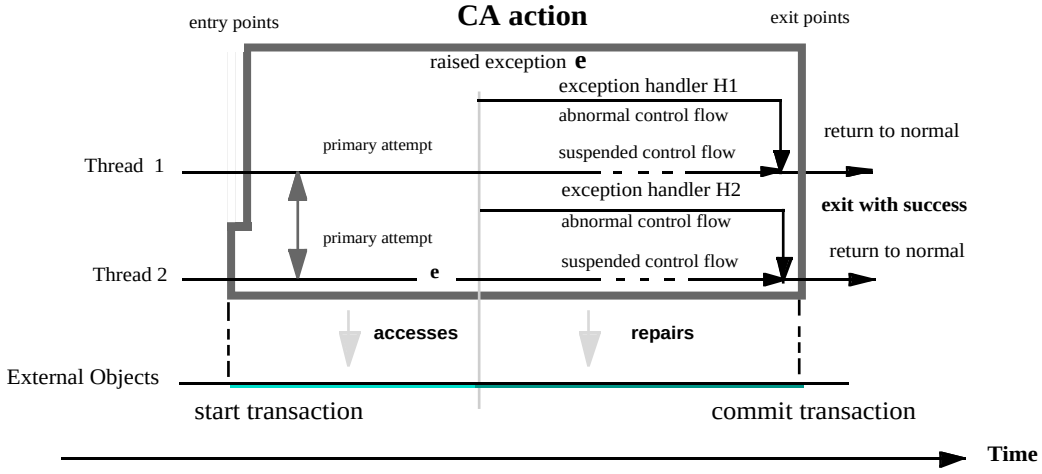

Figure 1 Coordinated error recovery performed by a CA action.

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 distributed 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 collection of, local or external atomic, 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 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 are totally internal to the CA action and must be handled by its own; the others are to be signalled to the enclosing CA action. There may be some overlapping between two types of exceptions, but they are conceptually different. All exceptions, $\boldsymbol{e} = \{e_1, e_2, e_3, ...\}$, that can be raised within a CA action must be declared with the action definition. But the corresponding exception handlers may be associated with respective roles that the participating threads are to perform. The set $\boldsymbol{e}$ of exceptions for a CA action is identical for each role of the action and a role has the corresponding handlers for those exceptions. However, for a given exception different roles may have different handlers.

The exceptions, $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \varepsilon_3,...\}$, that can be signalled from a CA action should be specified in the interface to the CA action. A pre-defined exception may indicate that an exceptional internal condition has occurred within the action, or that only incomplete results can be delivered by the action. An `undo` exception, μ, implies that the action has been aborted and all of its effect has been undone. Since `undo` is not always possible, a `failure` exception, f, will indicates that the action has been aborted but its effect may have not been undone completely. The enclosing action or the caller of the action is responsible for explicitly handling the exception. For a nested CA action and its direct-enclosing action, the definitions of $\boldsymbol{e}$ and $\boldsymbol{\varepsilon}$ are fully recursive, namely, $\boldsymbol{\varepsilon}_{nested} \subseteq \boldsymbol{e}_{enclosing}$.

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 a nested CA action. 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. In our model, three terms are used to clarify the route of exception propagation: an exception *e* is *raised* by a role within a CA action, other roles of the same action are then *informed* of the exception *e*, and if handling the exception within the CA action is not fully successful, a further exception $\boldsymbol{\varepsilon}$ will be *signalled* from a nested action to its enclosing action (see Figure 2).

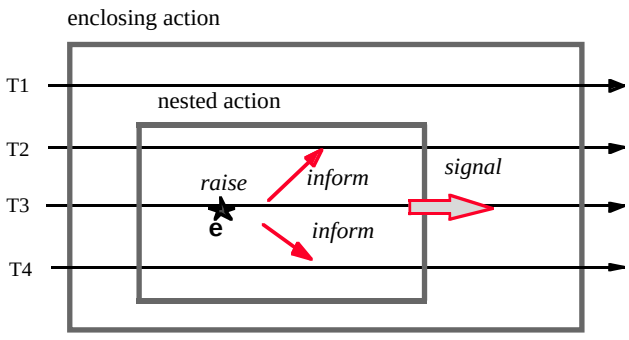

Figure 2 Exception propagation over nesting levels.

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 been pre-defined by the designer, or is determined dynamically, that is responsible for signalling an agreed exception to the enclosing action. Our model however adopts a more distributed strategy: each role of the nested action is responsible for signalling its own exception respectively. These exceptions should be the same but may be different. Because the enclosing action is also required to have the ability of handling concurrent exceptions, the exceptions concurrently signalled from the nested action will be simply handled as if they are concurrently raised in the enclosing action.

However, distributed exception signalling requires some final-stage coordination on two special exceptions μ and f. If any role of a nested action is to signal the exception f, then all other roles of the nested action must signal the same exception f to the enclosing action, indicating that some erroneous effect made by the nested action may have not been undone completely. Similarly, it makes sense only if all roles signal the same `undo` exception, μ, in order to ensure that all the effects made by the nested CA action (more precisely, made by respective roles) have been undone completely.

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 to the enclosing action.

External Objects: Since external shared objects can reflect the effect that a CA action may have on them, once an exception is raised within the CA action and hence error recovery is requested, the 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. The appropriate exception handlers may well be able to lead them to new valid states. When one or more external shared objects fail 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 remaining errors.

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 [Campbell & Randell 1986]. Each CA action should have its own exception graph. For example, the graph could be specified by the keyword **`exception hierarchy`** in the definition of a CA action. A simple hierarchy

form would be something like: e_r: e_1, e_2, ..., e_k, where e_1, e_2, ..., e_k are the direct low-level nodes of the resolving exception e_r.

3.2 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 a given $G(E, R)$, there may exist three types of nodes. The nodes with $d_{out}(e_i) = 0$ represent primitive exceptions that cover no other exceptions. The internal nodes, i.e. $d_{in}(e_i) \neq 0$ 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. A raised universal exception usually leads to the signalling of a `undo` or `failure` exception to the enclosing action.

Figure 3 shows an example of a three-level exception graph containing three primitive exceptions e_1, e_2, e_3 at the level 0. The resolving exception $e_1 \cap e_2$ at level one will be raised when e_1 and e_2 are raised concurrently. Similarly, the exception $e_1 \cap e_2 \cap e_3$ 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-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 method for defining exception graphs makes the automatic generation of an exception graph possible.

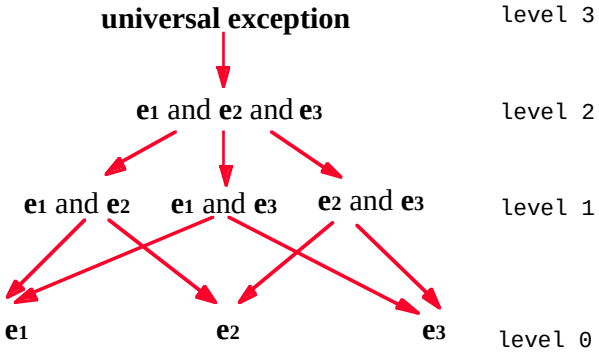

Figure 3 Example of a three-level exception graph.

There are however several ways of simplifying an exception graph by removing certain nodes for an actual application:

- For some particular exceptions, it may not be possible that they are raised concurrently. The corresponding internal nodes in an exception graph between level 0 and level *n* (or resolving exceptions) can be thus removed.
- At a given level of the exception graph, there may exist other order relationship between exceptions, i.e. an exception may be able to cover another exception of the same level. In this case the higher order exception can be moved to a higher level.
- An exception graph can be structured to contain only part of resolving exceptions that cover certain combination of the primitive exceptions. Other concurrently raised, primitive exceptions will simply cause the raising of the universal exception.
- The resolving exceptions may correspond to other logical relationships. For example, the resolving exception $e_1 \cup e_2 \cup e_3$ may cover more exceptional situations than the exception $e_1 \cap e_2 \cap e_3$, but may lead to a more complicated handler.
- Finally, it is clear that the size or complexity of an exception graph depends upon the fault model that a specific application has chosen to handle.

3.3 Concurrent Exception Propagation and Resolution

3.3.1 Algorithm-Related Assumptions and Definitions

For a given CA action it is assumed that each participating thread knows the set of all participating threads and uses the same exception graph, which is statically declared. Every thread has a name list of the nested actions it is to participate in. The currently innermost action for a specified thread 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 all participating threads $\{T_1, T_2, ..., T_i, ..., T_j, ...\}$, where each thread T_i has a unique identifier and all threads are ordered (e.g. thread names and the lexicographic ordering could be used). 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 keeps the following data structures:

list LE_i — records exceptions that have been raised or suspended states of threads that have halted normal computation;

stack SA_i — stores the exception context and the exception graph corresponding to each of nested CA actions.

During the execution of our algorithm, a participating 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), and `S` = Suspended (if T_i has to stop the normal computation due to the exceptions raised in other threads). It is 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 each thread T_i that does not raise any exception but has received `Exception` or `Suspended` messages from the others;

`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 once it receives this `Commit` message.

It is further assumed that an exception in an enclosing action will simply stop or abort any activity of its nested actions (including any nested resolution in progress and execution of any handlers). In order for action A_i to abort one of its nested action, an `abortion` exception must be raised within that nested action. Each thread of this nested action then starts the corresponding abortion handler. In general, when a participating thread in its active action A_{i+k} needs to take part in the abortion of a chain of the nested actions A_{i+1} (the outermost), A_{i+2}, ..., A_{i+k} (the innermost), it must execute abortion handlers in the order $(i + k)$, $(i + k – 1)$, ..., $(i + 1)$, ignoring any exception which may be signalled to a containing action. During the process of abortion, only the exception signalled by abortion handlers of action A_{i+1} is allowed to be raised in the containing action A_i.

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 could be developed. The proposed algorithm can clearly handle 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 the modular redundancy technique. (Our model described in section 3.1 is however general, and it is supposed to cope with exceptions that may be caused by node or communication line crashes.)

3.3.2 Algorithm for Coordinated Exception Handling

Our algorithm is based on the general support provided by the underlying system, including FIFO message sending/receiving between threads/objects and calls to abortion handlers. In addition, "$\rightarrow$" stands for "put in" and "$\Rightarrow$" for "sent to" in the description of our algorithm.

Algorithm:

For any T_i, $S(T_i)$: = N; and empty LE_i, SA_i;

loop

if T_i enters A **then**

<A> → SA_i; consume messages having arrived;

end if;

if T_i completes A **then**

delete last element in SA_i;

leave A (synchronously) —— $S(T_i)$: = N if leave A with success or

$S(T_i)$: = X if leave A with failure;

end if;

if E_i is raised in T_i **then**

$S(T_i)$: = X; <A, T_i, E_i> → LE_i;

Exception(A, T_i, E_i) ⇒ all T_j in G_A;

inform external objects (used by T_i within A) of the exception;

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> is the top element in SA_i*

<A^*, T_j, E_j> or <A^*, T_j, S> → LE_i;

exception information ⇒ uninformed external objects (used by T_i within A^*);

if A^* contains A **then**

abort all nested actions until A^*;

delete the elements in SA_i until <A^*>;

remove all elements except <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}> → LE_i;

Exception(A^*, T_i, E_{ab}) ⇒ all T_j in G_{A^*};

else $S(T_i)$: = S; <A^*, T_i, S> → LE_i;

Suspended(A^*, T_i, S) ⇒ 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> → LE_i;

Suspended(A^*, T_i, S) ⇒ all T_j in G_{A^*};

end if;

end if;

else retain the Exception or Suspended message till T_i enters A^*;

end if;

end if;

if T_i has all exceptions, or state S, of other threads within A *//<A> is the top element in SA_i*

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) ⇒ 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

Figure 4 illustrates how the new algorithm works when two concurrent exceptions `E1` and `E2` are raised in a nested CA action that contains four participating threads.

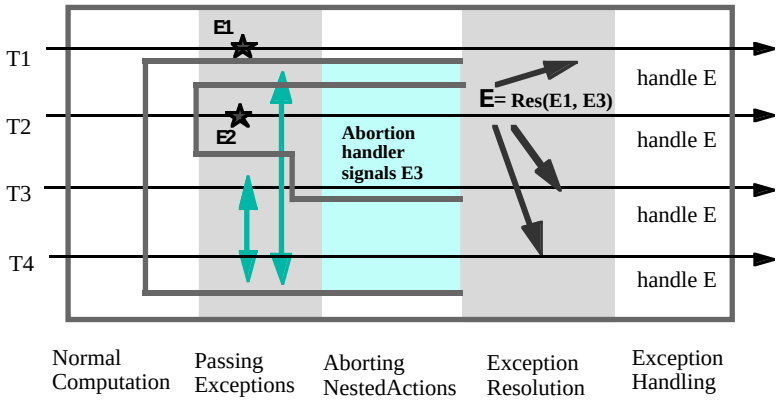

Figure 4 Concurrent exception handling and resolution.

3.2.3 Correctness and Communication Complexity

In order to prove the correctness of our algorithm, we re-state the following assumptions.

Assumption 1: Dependable communication between threads/objects is guaranteed, i.e. no message loss or corruption.

Assumption 2: FIFO message passing is supported by the target system, i.e. two messages from thread T_i will arrive at thread T_j in the same order as they were sent.

For a specific distributed system, 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 an (resolving) exception. We now show that no deadlock is possible in our proposed algorithm.

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

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

Proof: In order to prove the above bound, let us consider the worst case, i.e. a thread that raises an exception is in the innermost CA action and each time the abortion of a nested action occurs right at the end of exception handling within that nested action.

Without loss of generality, assume that a thread T_i in the innermost action raises an exception and changes its state into X. It will send the exception message to all the other participating

threads, by assumption 1, which will reach them in T_{mmax}. Since there are no further nested actions within the innermost action, any message from the other threads about an exception or suspended state will come to T_i in at most $2T_{mmax}$. Note that actual exception resolution may take T_{reso}. Therefore, T_i will receive a resolving exception and then complete exception handling in at most ($3T_{mmax} + T_{reso} + \Delta_{max}$).

If T_i has not yet left the innermost action, but a further exception occurs in its direct containing action, then the abortion of the innermost action will have to be performed. After the abortion, T_i will send either an abortion exception or suspended message to other threads, which will arrive at them in ($T_{abort} + T_{mmax}$). T_i will then receive the resolving exception (or resolve the exceptions by itself) in at most ($T_{reso} + T_{mmax}$) and complete exception handling within Δ_{max}. The whole process costs at most ($2T_{mmax} + T_{abort} + T_{reso} + \Delta_{max}$).

In the worst case, the above process could be repeated n_{max} times until the outermost CA action is reached. Totally the repeated process will cost at most $n_{max}(2T_{mmax} + T_{abort} + T_{reso} + \Delta_{max})$. Adding the time spent in the innermost action, we therefore have that

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

namely, thread T_i will complete exception handling ultimately and leave the outermost CA action. Q.E.D.

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.

Proof: Assume that, to the contrary, a new exception message arrives at the resolving thread after it has started the resolution. Note that, from the proposed algorithm, the resolving thread must know all the states (X or S) of the participating threads in *A* before it can begin any actual resolution. Hence, by assumption 2, the only possibility is that the newly arriving exception is caused by an abortion event, namely, *A* must be aborted by some containing action, contradicting the assumption that no exception is raised in any containing action of *A*.

Q.E.D.

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.

Proof: An exception that is raised in the containing CA action will abort any effect the nested action may have made or be making (even if a resolving exception for the nested action has been identified and the corresponding exception handling has been in operation). Note that however the number of nesting levels is finite and bounded by n_{max}. Abortion will be no longer possible if the current active action A is the outermost (or top-level) CA action. By Lemma 2, the exception resolution will start finally and no more new exception will be raised.

Q.E.D.

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.

Without the nesting of CA actions, it is obvious that the message complexity of our algorithm is $O(N^2)$ messages, where N is the number of the threads participating in the outermost CA action. More precisely,

1) when only one exception is raised and there are no nested actions, then the number of messages is $(N + 1) \times (N - 1)$, i.e. $(N - 1)$ `Exceptions`, $(N - 1)^2$ `Suspendeds`, and $(N - 1)$ `Commit` messages;

2) when all N participating threads have the exceptions raised simultaneously, the number of messages is also $(N + 1) \times (N - 1)$, i.e. $N \times (N - 1)$ `Exceptions` and $(N - 1)$ `Commit` messages.

From the proposed algorithm, we can see that the number of messages is in fact independent of the number of concurrent exceptions, which is a great improvement over our previous algorithm in [Romanovsky et al 1996]. Taking the nesting of actions into account, we have the theorem below.

Theorem 2: In the worst case, our proposed algorithm requires exactly $n_{max} \times (N^2 - 1)$ messages.

Note that the algorithm in [Campbell & Randell 1986] is of complexity $O(n_{max} \times N^3)$. Our previous algorithm in [Romanovsky et al 1996] could use $n_{max} \times 3N \times (N - 1)$ messages. Our new algorithm is less complex because 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 the last section 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 sucessful, or fail completely. In these cases a thread must signal a further exception ε 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 when exception handling has failed. We therefore need a further algorithm for coordinating those exceptions to be signalled.

Again, 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:

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).

A specific message is used in our algorithm:

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

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. Clearly, 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 an `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 caller.

Algorithm:

```
//after handling the resolving exception E
For any T_i of A, empty listSignal_i and undo = 0;
loop
  if T_i is to signal ε then
    <T_i, ε> → listSignal_i; where ε ∈ {φ, ε_1, ε_2, ε_3, ..., μ, f}
    toBeSignalled(T_i, ε) ⇒ all T_j in G_A;
  end if;
  if T_i receives toBeSignalled(T_j, ε) then
    <T_j, ε> → listSignal_i;
  end if;
  if |listSignal_i| = |G_A| then      //T_i has 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 <T_i, ε>;
          case 2: μ but no f in listSignal_i
                  if undo = 1 then
                  T_i signals μ;
                  else
                  empty listSignal_i and undo = 1;
                  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
```

4 Case Study: A Production Cell

Many practical systems that interact with their environments typically are 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 [Lewerentz & Lindner 1995], 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, [Zorzo et al 1997] 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 applied to 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 extendible 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.

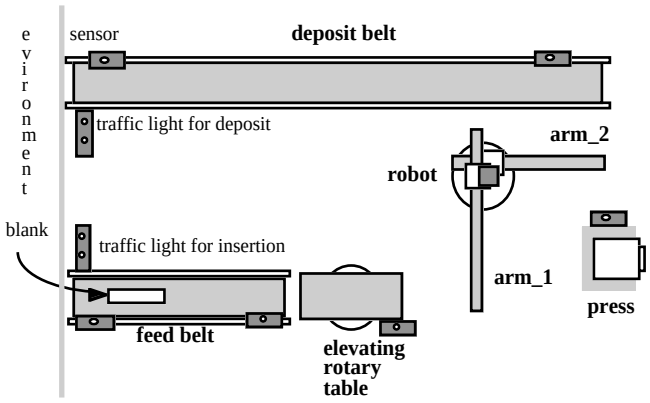

Figure 5 Production Cell (top view).

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 places the forged plate on the deposit belt, and 7) if the traffic light for deposit is green, the plate may be forwarded further and carried to the environment where a container may be used, e.g. by the blank consumer, to store the forged pieces.

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

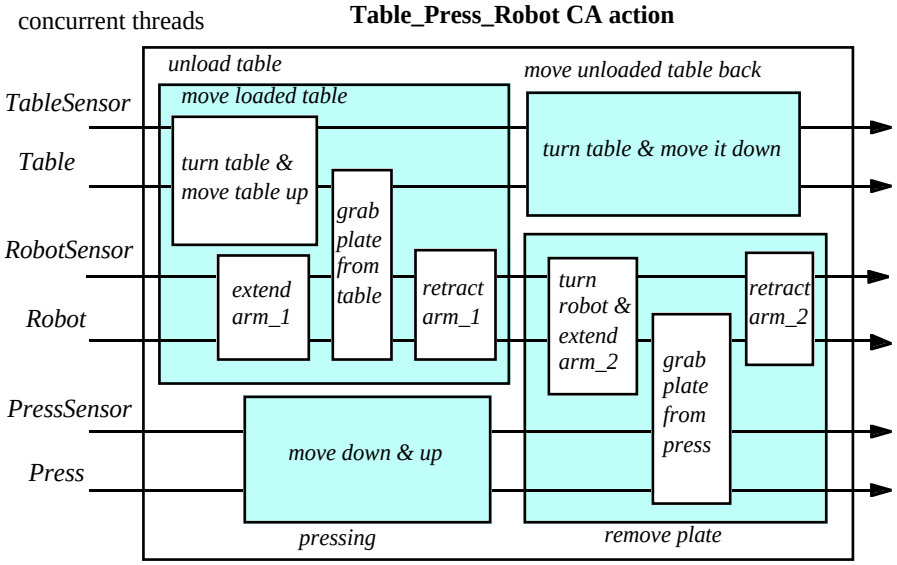

Figure 6 The Table_Press_Robot Action.

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 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 simply lead to the raising of the `universal` exception.

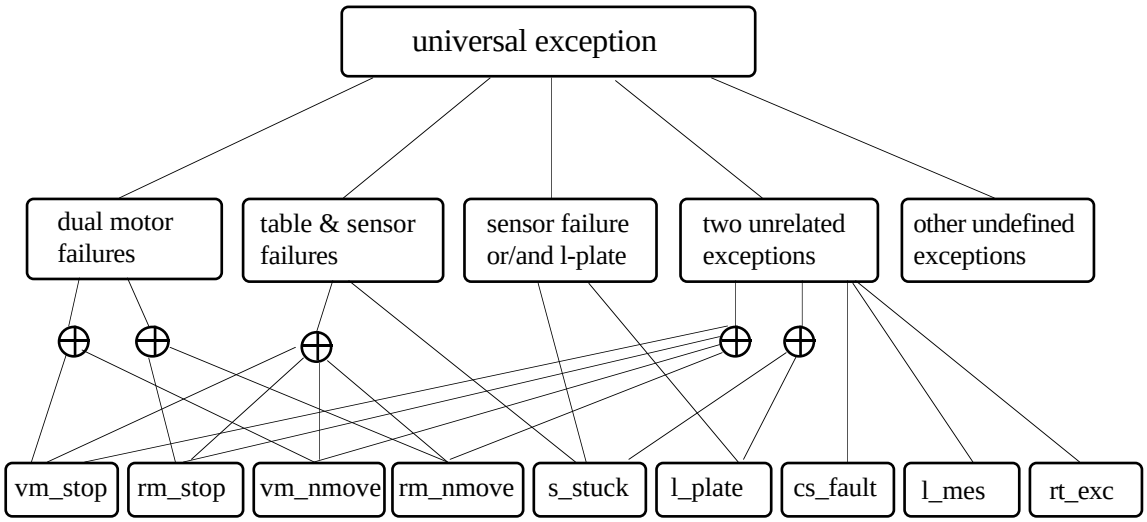

Figure 7 Exception graph for the `Move_Loaded_Table` action.

Some internal exceptions can be handled within an action while 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 undoing). These exceptions and the exceptions raised by the action `Unload_Table`, together with all exceptions signalled by actions `Extend_Arm1`, `Grabe_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` (one arm_1's sensor failure), in the hope that the outermost action may be able to handle them.

5 Implementation and Experimentation

In Ada 95 [Ada 1995] we have recently accomplished a prototype implementation of the resolution mechanism and the related CA action supporting system (with the standard features

of the Distributed Annex) in order to identify and tackle implementation and performance-related issues. We have chosen Ada 95 (the GNAT Ada 95 compiler, public release 3.04, on SunOS 5.4) because it is one of few standard OO languages that have features for distributed programming. Besides, its elaborate features for concurrent programming, such as protected objects, asynchronous transfer of control and conditional entry calls, greatly simplify the task of programming the run time support and ensuring the 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 simple, and hence 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 massage passing subsystem. Every partition has a copy of the run-time system, including the subsystems for concurrent exception handling and resolution where our new algorithms are realized. This basic CA action support offers the main CA action features: (nested) action entry points and exits, raising and signalling of exceptions, abortion of (nested) actions and calls to handlers. A simple protocol is also implemented for participating threads to leave a CA action synchronously.

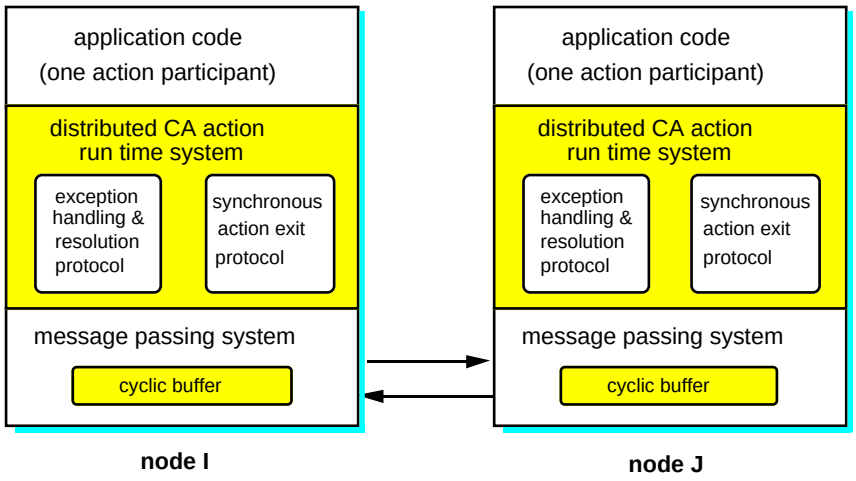

Figure 8 Prototype architecture for the CA action and resolution system.

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 action handlers. Every partition has a copy of the resolution function and of the exception graph so as to ensure that the handlers for the same exception are called in all the 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_{abo} 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} = 0.2*s*, T_{abo} = 0.1*s*, and T_{reso} = 0.3*s*; the execution of the system will take about 94.36*s*. In the tables of Figure 6 we present some of the experimental results with varying T_{mmax} (message passing), varying T_{abo} (abortion) and varying T_{reso} (resolution) values.

Message Passing	Total Execution Time	Abortion Time	Total Execution Time	Resolution Time	Total Execution Time
0.2	94.361391	0.1	94.361391	0.3	94.361391
0.4	98.586050	0.3	98.991825	0.5	98.352511
0.6	102.150904	0.5	101.939318	0.7	102.547776
0.8	106.774196	0.7	106.150075	0.9	107.164660
1.0	110.984972	0.9	110.154827	1.1	110.338507
1.2	125.078084	1.1	113.937682	1.3	114.729476
1.4	140.826807	1.3	118.147893	1.5	118.928022
1.6	161.766956	1.5	122.573297	1.7	122.483917
1.8	188.284787	1.7	128.461646	1.9	127.117187
2.0	214.519403	1.9	130.362452	2.1	131.816326
2.2	226.543372	2.1	134.165025	2.3	135.123453
2.4	237.934833				
2.6	249.744183				
2.8	261.768559				

Figure 9 Results of performance-related experiments.

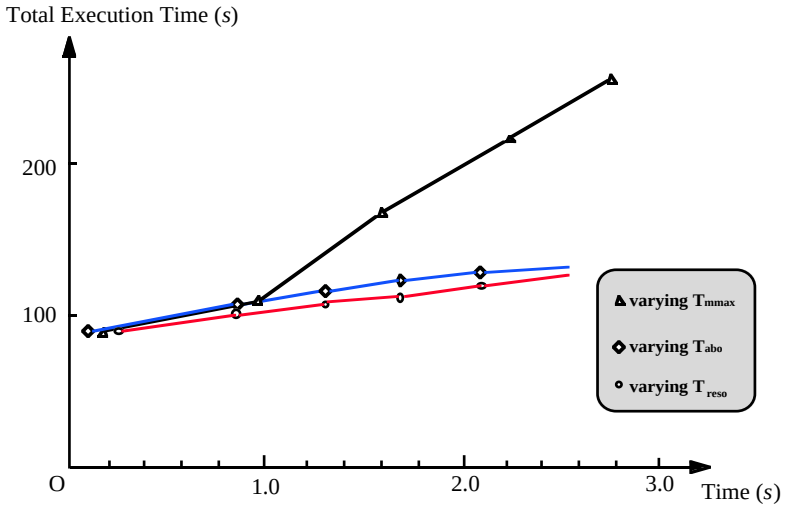

Figure 10 Sensibility of the total execution time.

The experimental data obtained are essentially consistent with the theoretical analysis presented in the previous sections. Figure 10 shows the sensibility of the total execution time of the application system. When T_{mmax} is limited within 1.0*s*, 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_{abo}, the total execution time has just 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.

5.3 Experimental Comparison of Different Resolution Algorithms

Another set of our experiments was intended for comparing the CR algorithm in [Campbell & Randell 1986] and our algorithms. We used the previously discussed implementation of our algorithm together with the CA action run time support. We modelled the CR algorithm by updating our algorithm and kept the rest of the CA action support unchanged. The application software and the resolution graph were the same for both resolution algorithms. The total execution time was measured for different T_{res} and T_{mmax}.

We have chosen to make a set of experiments with a simple application system. Three threads enter a CA action and after some period of computation all of them raise different exceptions nearly at the same time, so exception resolution is required. Figure 12 gives our experimental results: varying T_{mmax} given T_{res} = 0.3*s*, and varying T_{res} given T_{mmax} = 1.0*s*.

The results of these two experiments correspond well to the mathematical analysis of the algorithms. First of all, the dependencies on T_{mmax} and T_{res} for both algorithms are linear but

the coefficients differ a lot. Because the number of messages is $O(N^2)$ in our system and $O(N^3)$ in the CR algorithm we can see a big difference in the execution time even with the fixed N (N = 3 in our case) and, in particular, when the time of the message passing grows (see Figure 13(a)). In fact, the resolution procedure is called $N \times (N-1) \times (N-2)$ times in CR algorithms and only once in our approach. This can be clearly seen in Figure 13(b).

T_{mmax}	Time_our algo.	Time_CR algo.	T_{res}	Time_our algo.	Time_CR algo.
1.0	9.153302	11.770973	0.3	9.153302	11.770973
1.2	9.938735	12.978797	0.5	9.348575	12.358930
1.4	10.758318	14.168119	0.7	9.581770	12.984660
1.6	11.548076	15.397075	0.9	9.762674	13.604786
1.8	12.356180	16.558536	1.1	9.981335	14.212014
2.0	13.164378	17.757369	1.3	10.177758	14.817670
2.2	13.931107	18.967081	1.5	10.414642	15.288979
2.4	14.720373	20.188518			

Figure 12 Experimental results for comparison.

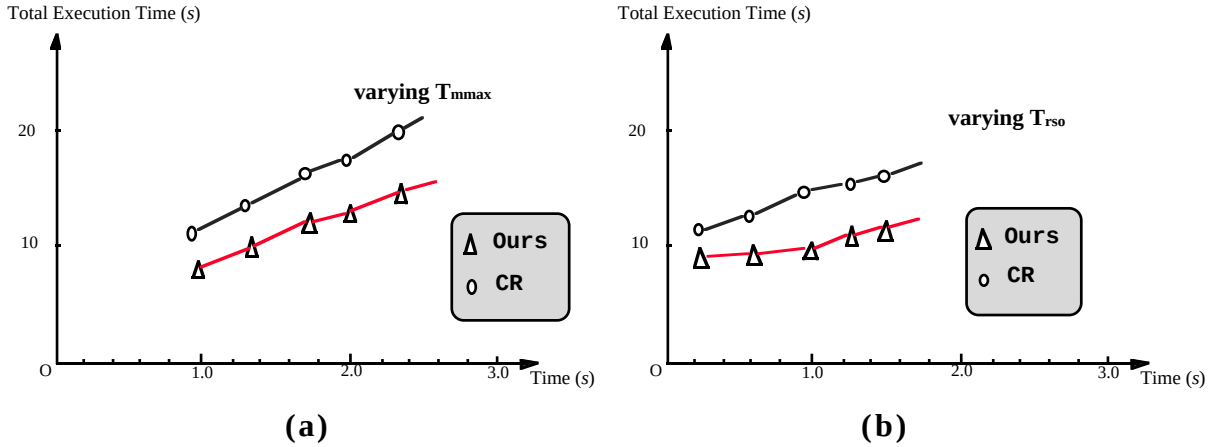

Figure 13 Comparison of two algorithms with respect to execution time.

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 [Campbell & Randell 1986] 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.

6.1 Related Work

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 [Jalot 1986][Yang & Kim 1992]. The Arche language introduced in [Issarny 1993] allows the programmer to implement a function that can resolve the exceptions propagated from several objects of the same type. Such resolution is however only suitable for a limited form of concurrency. Wellings and Burns [Wellings & Burns 1997] have recently shown how Ada 95 can be used to implement atomic actions, but without addressing exceptions concurrently raised. The work in [Miller & Tripathi 1997] discusses many important problems arising when exception handling mechanisms are used in object-oriented systems. It summarises previous research and presents new unsolved problems. The authors concluded that this is a difficult and delicate task because exception handling mechanism cannot be simply mapped into the OO paradigm. No actual solutions to these problems have been provided yet.

There are only a few concurrent and/or distributed OO languages, such as Ada 95, Java and Guide, known to us that have exception handling features. Ada 95 allows handlers to be called in several concurrent tasks when an exception has been raised in one of them. This language 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 is a significant new feature of Java [Java 1995]; it is similar to but not quite the same as exception handling in C++. The type of an exception may be specified with a `throws` clause in the method declaration, and the `try/catch/finally` is Java's exception handling mechanism without specially coping with concurrency-related (or multi-threaded) issues. Guide [Bolter et al 1994] has one of the most object-oriented exception mechanisms among existing languages: handlers are associated not only with exception names but also with type and method names; exceptions that can be raised by a method must appear in its interface. It is possible to ensure the consistency of objects by defining a restoration block to be executed just after the raising of an exception. However, Guide does not address in any way concurrent exceptions: its concurrency model is completely separated from the exception mechanism.

6.2 Future Research

Future research directions would be in three primary areas in terms of the further development of the CA action concept and of the method for handling concurrent exceptions. 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 of Newcastle, and J. Vachon of EFPL, Switzerland.

References

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

[Anderson & Lee 1980] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, Prentice-Hall International, 1980.

[Balter et al 1994] R. Balter, S. Lacourte, and M. Riveill, "The Guide language," *Computer J.* vol.37, no.6, pp.521-530, 1994.

[Campbell & Randell 1986] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Soft. Eng.*, vol. SE-12, no.8, pp.811-826, 1986.

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

[Issarny 1993] V. Issarny, "An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software," *Journal of Object-Oriented Programming*, vol.6, no.6, pp.29-40, 1993.

[Jalote 1986] P. Jalote, "Using Broadcast for Multiprocess Recovery," In *Proc. 6th Distributed Computing Systems Symposium*, pp.582-589, 1986.

[Jalote & Campbell 1986] P. Jalote and R.H. Campbell, "Atomic Actions for Software Fault Tolerance Using CSP," *IEEE Trans. Soft. Eng.*, vol. SE-12, no.1, pp.59-68, 1986.

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

[Lewerentz & Lindner 1995] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell"*, LNCS-891, Springer-Verlag, Jan. 1997.

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

[Miller & Tripathi 1997] R. Miller and A. Tripathi, "Issues with Exception Handling in Object-Oriented Systems," in *Proc. ECOOP'97 — Object-Oriented Programming*, pp.85-103, LNCS-1241, Finland, 1997.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no.2, pp.220-232, 1975.

[Randell et al 1997] B. Randell, A. Romanovsky, R. Stroud, J. Xu and A.F. Zorzo, "Coordinated Atomic Actions: from Concept to Implementation," Technical Report, Department of Computer Science, University of Newcastle upon Tyne, no.595, 1997.

[Romanovsky et al 1996] A. Romanovsky, J. Xu and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems," in *Proc. 16th IEEE International Conference on Distributed Computing Systems*, pp.545-552, Hong Kong, May 1996.

[Taylor 1986] D.J. Taylor, "Concurrency and Forward Recovery in Atomic Actions", *IEEE Trans. Soft. Eng.*, vol. SE-12, no.1, pp.69-78, 1986.

[Wellings & Burns 1997] A.J. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95", *IEEE Trans. Soft. Eng.*, vol. 23, no.2, pp.107-123, 1997.

[Xu et al 1995] 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. 25th Int. Symp. on Fault-Tolerant Computing*, pp.499-508, Pasadena, June 1995.

[Yang & Kim 1992] S.M. Yang and K.H. Kim, "Implementation of the Conversation Scheme in Message-Based Distributed Computer Systems," *IEEE Trans. Parallel and Distributed Sys.*, vol.3, no.5, pp.555-572, Sept. 1992.

[Zorzo et al 1997] 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," Technical Report (obtained on request), Department of Computer Science, University of Newcastle upon Tyne, 1997.