Concurrent Exception Handling and Resolution in Distributed Object Systems

Jie Xu, Alexander Romanovsky, and Brian Randell

Abstract—We address the problem of how to handle exceptions in distributed object systems. In a distributed computing environment, exceptions may be raised simultaneously in different processing nodes and thus need to be treated in a coordinated manner. Mishandling concurrent exceptions can lead to catastrophic consequences. We take two kinds of concurrency into account: 1) Several objects are designed collectively and invoked concurrently to achieve a global goal and 2) multiple objects (or object groups) that are designed independently compete for the same system resources. We propose a new distributed algorithm for resolving concurrent exceptions and show that the algorithm works correctly even in complex nested situations, and is an improvement over previous proposals in that it requires only $O(n_{max}N^2)$ messages, thereby permitting quicker response to exceptions.

Index Terms—Concurrent exeception handling, distributed systems, exception resolution, nested atomic actions, object-oriented programming.

1 Introduction

Concurrent and distributed computing systems often give rise to very complex asynchronous and interacting activities. The provision of error recovery is an extremely difficult problem in such circumstances [24]. In order to control this complexity and, hence, facilitate the provision of error recovery, some way of restricting interaction and communication will be required. The concept of *coordinated atomicactions* (or CA actions) that has been developed at Newcastle University [29], [30], [31] is one such mechanism for the strict enclosure of interaction and recovery activities. A CA action coordinates error recovery between multiple interacting objects in an object-oriented (OO) or object-based system by integrating two complementary concepts —*conversations* [24] (together with a technique for concurrent exception handling [5]) and *transactions* [19]. The work reported in this paper is essentially a continuation of research on CA actions, concentrating on technical details of concurrent exception handling and resolution.

Using the CA action framework as one kind of control abstraction, we establish an exception model that clarifies concepts such as exception context, declarations, and propagation in a distributed object system. We develop an abortion mechanism in order to tackle the problem of handling exceptional situations across nested levels of CA actions. In a distributed computing environment, another delicate problem is that of concurrent exceptions [5]—different processing nodes may raise different exceptions and the exceptions may be raised simultaneously. This can further complicate the process of exception handling. Take a twin-engine aircraft as an example. If just the left (or right) engine fails, the controls can be adjusted appropriately to compensate for the loss of the left (right) engine. However, if both the right and left engine fail at the same time, handling both engine-loss exceptions separately and sequentially can lead to catastrophic consequences. Another example is that of a distributed control system composed of several components. Two exceptions, Fire_Alarm and Gas_Leakage, may be raised concurrently by different components. It is obvious that handling such exceptions sequentially (in either order) is not adequate and a new method that can handle a combined occurrence of these two exceptions has to be developed.

There are a variety of reasons why several exceptions may be raised concurrently. For example, in practice, it is often difficult to interrupt the normal operations of the other nodes immediately after an exception has been raised, so new exceptions may be raised in these other nodes before they are informed of the initial exception. In addition, concurrently raised exceptions may be merely a manifestation in multiple nodes of a system-wide exception. In Section 3.2, we present a detailed discussion of the necessity of coping with concurrent exceptions and argue that a hierarchy-based approach is essential in order to find a higher-order exception that can "cover" all the exceptions raised concurrently. This further requires a distributed scheme for determining the proper recovery strategy and for involving all the related nodes in the recovery activity.

In 1986, Campbell and Randell [5] introduced the first algorithm (referred to as the CR algorithm) for concurrent exception resolution in a process-oriented system. We identify a number of issues involved in the CR algorithm and present a new distributed algorithm relying on strictly defined, practically oriented assumptions. This new, object-oriented algorithm is of lower message complexity than the original solution and it is formally proven and implemented in distributed Ada 95 [1].

- *J. Xu is with the Department of Computer Science, University of Durham, DH1 3LE, UK. E-mail: Jie.Xu@durham.ac.uk.*
- *A. Romanovsky and B. Randell are with the Department of Computer Science, University of Newcastle upon Tyne, NE1 7RU, UK.*

Manuscript received 4 Sept. 1996; revised 23 May 2000; accepted 28 Aug. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100287.

2 Faults, Errors and Exception Handling

In this paper, we consider a distributed system consisting of nodes connected by a communication network. The objects that run on network nodes communicate with each other by message passing. The CA action framework takes both *hardware and software faults* into account [29]. Hardware faults include crashes of, or transient faults in, nodes or the communication network. Erroneous information may spread through communication channels. However, as assumed by others in similar research on exceptions [8], [12], we assume that *exception handling* embedded in a CA action needs to be effective and responsible just for some specific expected conditions, i.e., *errors* detected by tests and programs running on fault-free hardware, including those detected and signaled by hardware or by lower level services such as file services.

2.1 Error Recovery and Exception Mechanisms

Fault-tolerant software detects errors produced by faults and employs *error recovery* techniques to restore normal computation. Forward error recovery (mostly exception handling schemes) is based on the use of redundant data 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.

An exception (handling) mechanism is a (more language-oriented) 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 (see [8] for rigorous and thorough discussion). There have been some considerable efforts in developing formal semantics of exception handling (e.g., ML [23] and Hoare's "$s1$ **otherwise** $s2$" construct). The exception mechanism is usually considered an essential part of a modern language (see Ada 95 [1], C++, Eiffel [22], and Java, for example). Ideally, it should be coherent with the language, the entire programming paradigm, and the design methodology.

For any given exception mechanism, *exception contexts* [5], namely regions in which the same exceptions are treated in the same way, have to be declared (very often they are blocks or procedure bodies). Each context should have a set of associated exception handlers, one of which is called when a corresponding exception is *raised*. There are different exception models: The *termination* exception model assumes that, when an exception is raised, the corresponding handler copes with the exception and completes the block execution; the *resumption* model assumes that the handler recovers the program state and the program then continues the execution from the operation following that which raised the exception. If the handler for the raised 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 of nested blocks. The appropriate handler is sought in the exception context containing the context which raised or propagated the exception.

Exception handling and the provision of error recovery are extremely difficult in concurrent and distributed systems. Much previous work has focused on the concurrency aspect of such systems (see the subsequent discussion) without addressing the other important aspect —distribution. Note that each node in a distributed system 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 [2]. The time of message passing is therefore not negligible. Obviously, implementing coordinated recovery in a loosely coupled distributed system is a more difficult problem than designing similar schemes under either a uniprocessor or a multiprocessor environment with common memory. Special protocols must be designed in order to ensure timely response to exceptions and coordinated error recovery in spite of physical distribution.

2.2 Conversations and Concurrent Exception Handling

The concept of conversations was first proposed in [24] and intended to provide joint backward error recovery of concurrent processes that have been designed to cooperate by exchanging information. Each process participating in such a conversation must save its state on entering it. While inside a conversation, a process can only communicate with other processes in the same conversation. If any process fails its acceptance test, then every participating process will roll back to the saved state and retry, perhaps using an alternate algorithm. Processes can enter a conversation asynchronously, but (at least notionally) must leave it at the same time once the acceptance test in each process has been satisfied. This general idea has been extended in several different ways in later research (see [20] for comprehensive discussion).

A systematic approach to concurrent exception handling is developed in [5] by integrating an exception mechanism into the conversation scheme and extending the well-known atomic action paradigm [14]. A set of exceptions is associated with each action (i.e., each conversation). Each process participating in an action has a set of handlers for all or some of the predefined exceptions. When any of these exceptions is raised in any process, appropriate handlers (for the same exception in all processes) will be initiated in all action participants. The notion of exception resolution and the resolution mechanism proposed in [5] are critical while considering multiple concurrent exceptions. The exception tree structure introduced in [5] has advantages over simple exception priorities for resolving these exceptions. (An exception tree consists of all exceptions associated with the action and imposes a partial order on them so that a higher exception has a handler which is intended to handle any lower level exception.) However, the identification of exceptions and the establishment of exception trees are not a simple task for an actual application. The developer of the application has to analyze typical abnormal situations to define exceptions and the exception tree based on application-specific characteristics,

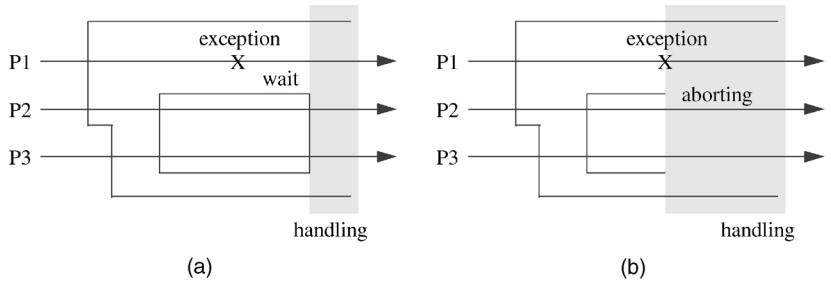


Fig. 1. Two methods for treating nested actions while an exception is raised.

fault assumptions, and available resources. The work reported in [31] demonstrated how these issues were addressed in an industrial case study.

The nesting of actions (or conversations) presents new problems not normally encountered in nonnested circumstances. For example, an exception may be raised in a process that has not yet entered a nested action, although some participants of the same containing action where the exception is raised have entered this nested action. The authors of [5] suggested two methods for overcoming this difficulty. The first method is to wait until the nested action is completed—a natural decision because the execution of the nested action is invisible and indivisible for the containing action (see Fig. 1a, where P1, P2, and P3 are participating processes). An alternative method is to raise an abortion exception in all participants of the nested action in order to abort the nested action completely, as shown in Fig. 1b. (One can implement an abortion handler in each participating process of a nested action or let the underlying support mechanism handle any abortion exception.) The second method seems to be more practical. First of all, it can be the case that a process detecting an error is expected to enter the nested action but will never be able to, so other processes in the nested action would wait forever for it to continue execution. Second, for real-time systems, it would be more predictable to abort the nested action than to wait for its completion [5].

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 [13], [32]. Of these, [32] focused on two particular conversation schemes (i.e., the name-linked recovery block and the abstract data type) and addressed various implementation issues which are specific to the chosen conversation structures. The work in [13] discussed a distributed implementation of the conversation scheme using broadcasts, assuming that all processes will enter each conversation simultaneously. Neither approach can be used directly to implement the CA action scheme because they focus on some particular schemes, with no support for forward error recovery and for OO systems. The Arche language introduced in [12] allows the application programmer to implement a function that can resolve the exceptions propagated from several objects (i.e., different implementations) of the same type. The resolution function takes all exceptions that have been raised and not handled in those objects as input parameters and returns the only "concerted" exception that will be handled in the context of the calling object. Although the Arche approach is object-oriented, it cannot be used for CA actions since it supports only a limited kind of concurrency, which relies heavily on the underlying multifunction call feature. As a result, it can be used for NVP-type schemes [3], but is not suitable for cooperative concurrency and recovery of several objects with different types. Moreover, parameterized exceptions, though suitable for Arche, cannot be used directly for CA actions because the handler for an exception which was not raised may be still called once exception resolution is required.

2.3 Exception Mechanisms in Realistic OO Languages

In reality, exceptions in OO languages can be declared either as classes, objects, or strings [10], [26], while exception handlers can be declared and attached to the level of statements, methods, classes, or objects. Some languages, such as C++, Modula-3, and Arche [12], only allow exception handlers to be attached to statements and others, such as Lore [10], Eiffel [22], Guide [4], extended C++ [21], and extended Ada [9], permit exception handlers to be attached to methods and objects or classes. Such flexible attachment has numerous advantages:

1. a clear separation of an object's abnormal behavior from its normal one, in accordance with the concept of an idealized fault-tolerant component [14];
2. object/class recovery provided at the object level;
3. exceptions associated with types; and
4. software layering which facilitates the design of fault-tolerant systems.

There is evidence from practice [9] as well: The use of object exception handlers can decrease program complexity and facilitate program design, maintenance, and reuse.

Object/class exception propagation is another important topic. Lore, Eiffel, and Guide propagate exceptions through the call chain; the exception context is associated not only with the method execution, but also with the object/class itself. In extended Ada, exceptions cannot be passed out of class handlers, whereas extended C++ propagates the class exception along the object creation chain (which may or may not coincide with the call chain).

There are only a few concurrent OO languages, such as Java, Ada 95 [1], Arche, and Guide, known to us that have exception handling features. Java has a C++-like exception mechanism, without specially coping with concurrency-related (or multithreaded) issues. Ada 95 allows handlers to be called in several concurrent tasks when an exception has

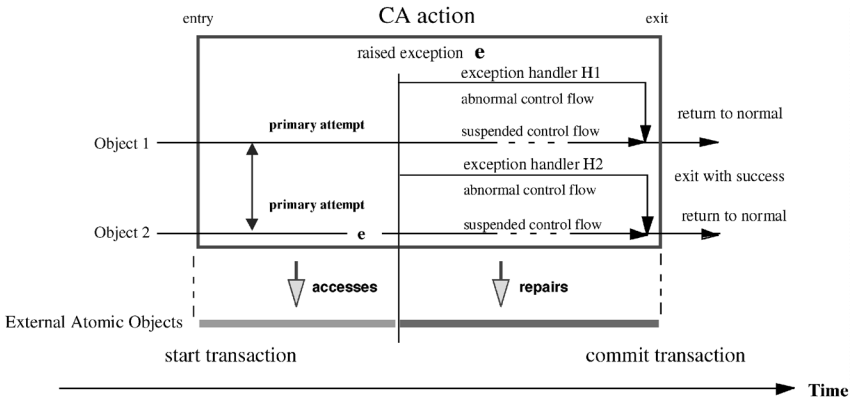


Fig. 2. Forward error recovery in a CA action.

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. Arche permits user-defined resolution of concurrent exceptions among a group of objects that belong to different implementations of a given type, which, unfortunately, is not generally applicable to the coordination of multiple interacting objects with different types. For our purposes, we need an exception model applicable to any group of interacting objects whether or not of the same type.

3 CA Actions: Concurrency, Coordination, and Exception Resolution

The CA action scheme presents a general technique for achieving fault tolerance in concurrent and distributed OO software by integrating conversation-type structures, transactions, and concurrent exception handling into a uniform conceptual framework [29]. This technique allows complex OO software to be designed in a disciplined and structured way. CA actions take two kinds of concurrency into account: *cooperating* and *competing* [14]. Several objects can be designed collectively by different programmers (or teams) and invoked concurrently in order to achieve certain joint and global goals. But, these objects must cooperate within the boundaries of a CA action. Competitive concurrency may also exist in such systems since two or more separately designed objects 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 objects that have been designed to cooperate with each other (referred to as *participating objects* of the CA action). Shared external objects are controlled by the associated transaction mechanism that guarantees the ACID properties (atomicity, consistency, isolation, durability [19]). In particular, objects that are external to the CA action, and can hence be shared with other actions and objects concurrently, must be *atomic* and individually responsible for their own integrity.

Fig. 2 shows an example in which two participating objects enter a CA action in order to play the corresponding roles. Within the CA action, two concurrent roles communicate with each other and manipulate the external objects cooperatively in pursuit of some common goal. However, during the execution of the CA action, an exception e is raised by one of the roles. The other role is then informed of the exception and both roles transfer control to their respective exception handlers, H_1 and H_2 for this particular exception, 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 outcome. Two participating objects leave the CA action synchronously at the end of the action. (As an alternative to performing forward error recovery, the CA action could undo the effects of operations on the external objects, roll back, and then try again, possibly using diversely designed software alternates.)

In principle, exception handling (or forward error recovery in general) can and should be integrated into the CA action framework. But, this cannot be achieved before we find appropriate solutions to three fundamental problems: exception model, exceptions in nested actions, and exception resolution—it is these problems that are the focus of this paper.

3.1 Exception Model for CA Actions

To be generally applicable, our model assumes that exceptions may be declared in any of the ways discussed in Section 2.3. The exceptions that can be raised within a CA action must be declared, together with the action declaration, and exception handlers must be associated with participating objects of the CA action. Thus, when a participating object enters the action, it enters the corresponding exception context. A subset of these participating objects may further enter a nested CA action, which has all the properties of a nested transaction in terms of atomic objects. Note that an exception is *raised* within a CA action, but *signaled* from a nested action to its containing action. Since the nesting of CA actions causes the nesting of exception contexts, it must thus be guaranteed that each participating object of the nested action is associated with an appropriate set of handlers. In practice, such association could be done either statically or dynamically. Once the association is provided, clear semantics of exception propagation can be enforced easily. Exceptions can be propagated along nested exception contexts, corresponding to the chain of nested CA actions.

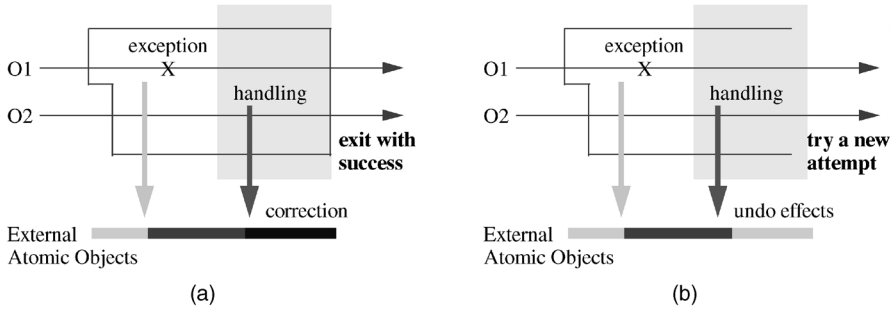


Fig. 3. Error recovery in external atomic objects. (a) Forward error recovery. (b) Backward error recovery.

However, the issue of how handlers are associated correctly with the exception context depends upon the particular way objects enter a CA action and upon peculiarities of the target OO system. If an object enters an action through an operation call and stays in it until that operation is completed, then operation-level exceptions would be appropriate for the required association. Otherwise, only object/class level exceptions can be used if the exception context cannot be changed dynamically. Here, we simply adhere to the termination model of exceptions—in any exceptional situations, handlers take over the duties of participating objects in a CA action and complete the action either successfully or by signaling a failure exception to the containing action.

Because a CA action may cope with two kinds of concurrency, external shared objects must be treated explicitly when forward error recovery is requested. We do not impose strict rules on the use of atomic objects during forward recovery, but require that these atomic objects should be always left to be in a consistent state immediately after recovery. It is particularly important to notice that an exception raised within the CA action does not necessarily cause restoration of all the atomic objects to their prior states. The appropriate exception handlers may well be able to lead them to new valid states (see Fig. 3a, where O1 and O2 are participating objects).

If any of the external shared objects fails to reach a correct state, a failure exception must be signaled to the containing CA action. A CA action can start a new attempt after backward error recovery has been performed on the external objects, as illustrated in Fig. 3b. A new associated transaction will be then issued [29]. In principle, object programmers have the freedom of choosing appropriate policies in order to guarantee the consistency of external atomic objects. There would be a wide spectrum of application-specific strategies: from simple correction of the erroneous states through handlers to the "bottom line" of relying on undoing all previous modifications.

3.2 Necessity of Concurrent Exception Resolution

One of the most basic questions as to the importance of our work concerns whether it is necessary to deal with concurrent exceptions; there may be a very low, negligible probability that multiple exceptions arise in a system at the exactly same time. However, after a careful examination of the problem, we argue the necessity of coping with concurrent exceptions for the following reasons:

- It is, in practice, difficult to interrupt the normal operations of all participating objects *immediately* after one of them has raised an exception. The probability that new exceptions are raised in other objects before they are informed of this exception is much higher in a distributed system than in a uni- or multiprocessor system with common main memory.
- Because no error detection tools are perfect, the latent period of an error is not negligible and, so, erroneous information can be spread within the boundaries of a CA action; thus, several errors occurring concurrently in different objects can be the symptoms of a different, more serious fault [5].
- Due to the nesting of CA actions and the time interval, which may be lengthy, between the first occurrence of an exception in a nested action and the end of that nested action (which may lead to further exceptions), several successive exceptions raised in different containing and nested actions can be, in effect, concurrent.
- Different participating objects can be involved in nested CA actions at different levels so that their exception contexts may be different.
- In distributed systems, the overall hardware-related failure probability is relatively higher and they are more difficult to program without design faults than centralized systems [7].
- Finally, very often there is a correlation between errors so that they happen in a very short period of time in different participating objects. On one hand, due to hardware-related operational errors, several nodes can be affected by the same bad conditions or by a channel through which traffic between several nodes can be damaged. On the other hand, because participating objects of a CA action were designed cooperatively from a given specification, an error in the specification or cooperative misunderstanding during the design could affect several or all of the participating objects.

When exceptions are raised concurrently, we cannot simply handle them sequentially in some order. A traditional priority-based method is not adequate, either. Recall the aircraft example in the Introduction section. The exceptions raised concurrently in two engines must be handled cooperatively and at the same time. A hierarchy-based approach is therefore needed in order to find a single higher-order exception that can "cover" all the exceptions

raised concurrently. This further requires a distributed resolution scheme for determining the correct recovery strategy and for involving all the participating objects in the recovery activity.

3.3 Concurrent Exception Resolution: Issues and Difficulties

There are three sources of exceptions defined in [5]. The first source is of exceptions that are raised during the execution of the application code; the second is of exceptions signaled by the nested action; the third is of exceptions that are raised because participants of the atomic action received information about an exception raised in some other participant, but have no handler for it, so they have to examine the exception tree and find and raise a new appropriate exception (for which there is a handler). This is mainly because not all of the exceptions declared in the action declaration will necessarily have associated handlers in each participant of the action (although each participant could contain the use of the default handler). In [5], each participant knows only a reduced local tree of exceptions with specific handlers and has to look through it after raising each exception and after each resolution. However, repeated search of the local tree could cause a kind of "domino effect"; in certain cases, the CA action will fail in spite of there being several handlers implemented in respective participating objects. This could happen if the exception tree is organized as a directed chain: Consider an action A which has the exception tree TA and two participating objects O_1 and O_2, with two reduced trees (i.e., sets of exceptions with specific handlers) `TO1` and `TO2`, respectively.

```
TA : e1 − − > e2 − − > e3 − − > e4 − − > e5 − − > e6
     − − > e7 − − > e8
TO1 : e1 − − > e3 − − > e5 − − > e7
TO2 : e2 − − > e4 − − > e6 − − > e8
```

Note that if exception e8 is raised in O_2 and O_1 is informed of it, then O_1 has to find the appropriate exception to raise (in this example it is `e7`). When O_2 is informed of `e7`, it has to raise the further exception `e6`, in which case `e5` will be raised in O_1, and so on. Therefore, in this example, any exception will always lead to further exceptions until the root of the exception tree is reached. To solve this problem, we will assume in our new mechanism that each participating object has handlers for all exceptions declared in a given action. It is, we argue, a natural assumption since participating objects are implemented cooperatively and all of them should be involved in any activity of exception handling.

Another important issue is how to raise an abortion exception in nested actions. The CR algorithm relies heavily on such abortion, but assumes that the related operations can be provided by the underlying system. However, we found that this is not a trivial problem. Consider four concurrent objects, O_1, O_2, O_3, and O_4, in several nested atomic actions (see Fig. 4). If O_1 detects an error and thus raises an exception, O_2, O_3, and O_4 will be informed subsequently of the exception. Since O_1 may know nothing about nested actions A_2 and A_3, O_2, O_3, and O_4 are

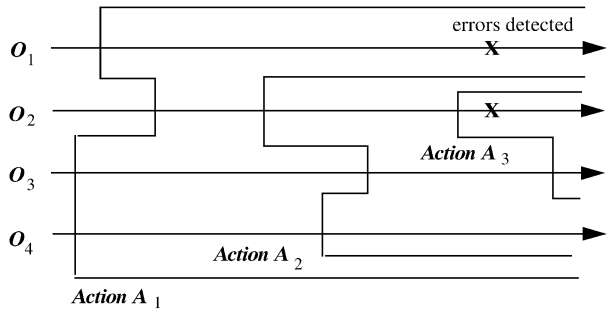


Fig. 4. An example of four objects in nested actions.

responsible for actual abortion of these actions. Several problems (which were not adequately discussed in [5]) are as follows:

1. A_3 should be aborted before A_2;
2. O_2, O_3, and O_4 are responsible for aborting A_2;
3. If O_1 was supposed to enter A_2 and A_3 but failed to do so due to an error (O_1 is thus a *belated* participant for A_2 and A_3), O_2, O_3, and $O4$ could wait for it to complete the abortion of A_2 and A_3 (so abortion handlers must be used that have been implemented in a very special way in order to avoid deadlocks);
4. If O_2 raises an exception as well, all A_3 participants (maybe including O_1 as a belated participant, see the reason above) must participate in error recovery; any lower level resolution performed by O_2 should be ignored when the resolution is started by O_1 within A_1 (however, since belated participants can participate in the resolution only when they enter the nested action, the entire protocol execution for resolution has to be delayed); and
5. To abort nested actions, only abortion handlers should be executed because the execution of other handlers would not guarantee correct abortion.

Hence, all exceptions signaled by abortion handlers in a nested action have to be ignored unless the action is nested directly in the action where an exception was raised (e.g., all signaling from within the nested action A_3, but not from A_2, will be ignored for the resolution performed by Action A_1).

In fact, from the point of view of a single object that cooperates with other objects in a distributed system, our exception model can be described in the following way: If an exception is raised in the object, it is guaranteed that a handler for the exception or for an exception with a higher order in the resolution tree will be executed. An object may be interrupted at any time and forced to perform an exception handler even if it did not encounter any problem of its own. This includes the situation in which the current action the object participates in is aborted and the object is thus forced to execute its corresponding abortion handler. We shall describe a new algorithm for exception resolution below, based on a set of carefully defined assumptions, which allow us to overcome all the above-mentioned problems.

4 Distributed Algorithm for Exception Resolution

According to our model, objects may enter a CA action asynchronously. A (centralized or decentralized) manager of CA actions has 1) to guarantee that all participating objects will wait for each other on the acceptance test line while using backward error recovery or 2) to invoke exception handlers for the same exception in these objects in order to provide coordinated forward error recovery.

4.1 Assumptions and Definitions

It is assumed that, for a given CA action, each participating object knows all other participating objects of the same action and has the same exception tree (which is statically declared). Each object also has a name list of the nested actions it participates in. The currently innermost action for the object is called the *active* CA action (for that object). Again, note that nested actions can end their executions by signaling an exception to the containing CA action.

In order for action A_i to abort a nested action, an abortion exception must be raised within the nested action and any activity of the nested action stopped (including any nested resolution in progress and execution of any handlers). Each object in this nested action then starts the corresponding abortion handler. In general, when an object 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 signaled to a containing action. During the process of abortion, only the exception signaled by abortion handlers of directly nested action A_{i+1} is allowed to be raised in the containing action A_i. This is simply because any handler for a specific exception cannot be called in those actions which have to be aborted.

An object transits from the normal state to the abnormal state when 1) an exception is raised within the object or 2) it receives the message concerning an exception in one or more other objects. Again, it is important to notice that an object in the exceptional state, of action A_i, may raise a further exception which is signalled by abortion handlers of the nested action A_{i+1}. However, we assume that only one such exception can be raised within action A_i. The handler for the exception is intended to perform the simple "last-will" recovery (see the discussion in [5]). We allow for the possibility that the abortion handlers of the nested action A_{i+1} signal different exceptions to the containing action A_i (though, in accordance with [5], we believe that, in practice, the same exceptions should be signaled from all participating objects of action A_{i+1}). If there exists any belated participating object of action A_{i+1}, the abortion handlers of other participating objects will not have to wait for it in order to carry out abortion promptly.

Let *CA-action* be the outermost CA action. We define $G_{CA\text{-}action}$ as the group of all participating objects $\{O_1, O_2, ..., O_i, ..., O_j, ...\}$, where each object O_i has a unique number and all objects are ordered (e.g., object names and the lexicographic ordering could be used). Such ordering helps to dynamically identify a unique object among objects that raised exceptions and the chosen object will be responsible for performing actual exception resolution. Let A be the active action of O_i and G_A be the corresponding set of participating objects. We assume that each object O_i has the following data structures:

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

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

In the interests of simplicity and brevity, we assume that application-related message passing is treated independently. In our algorithm, only the following specific messages are used:

`Exception`(A, O_i, E) is sent by object O_i to all participating objects of action A when an exception E is raised within O_i;

`Suspended`(A, O_i, S) is sent by each object O_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 object in action A to all participating objects after it completes resolution of exceptions, where E is the resolved exception. A corresponding handler for E will be called by each object once it receives this `Commit` message.

4.2 The Algorithm

Our algorithm is based on the general support provided by the underlying system, including FIFO message sending/receiving between objects and calls to abortion handlers. During its execution, a participating object O_i may be in one of the following states (denoted by $S(O_i)$): `N` = Normal, `X` = Exceptional (if an exception was raised in O_i), and `S` = Suspended (if O_i has to stop the normal computation due to the exceptions raised in other objects). In addition, "$\rightarrow$" stands for "put in" and "$\Rightarrow$" for "sent to" in the description of our algorithm. (For a possible implementation, a "process" running this algorithm may be associated with each pair (O_i, A) of objects and CA actions.)

Algorithm 1:

For any O_i,$S(O_i)$: = `N`; and empty LE_i, SA_i;
loop
if O_i enters A **then**
  $<A> \rightarrow SA_i$; consume messages having arrived;
end if;

if O_i completes A **then**
  delete the top element in SA_i;
  $S(O_i)$: = `N` if A ends with success or $S(O_i)$: = `X` if A ends with failure;
  leave A (synchronously)
end if;

if E_i is raised in O_i **then**
$S(O_i)$: = `X`; $<A, O_i, E_i> \rightarrow LE_i$;
  `Exception`(A, O_i, E_i) all O_j in G_A;
  exception information $\Rightarrow$ external objects (used by O_i within A);
end if;

if O_i receives `Exception`(A^*, O_j, E_j) **or**
`Suspended`(A^*, O_j, S) **then**
if A^* contains or equals A **then** *//<A> is the top element in SA_i*
<A^*, O_j, E_j> or <A^*, O_j, S> LE_i;
exception information $\Rightarrow$ uninformed external objects (used by O_i within A^*);
if A^* contains A **then**
abort all nested actions until A^*;
delete the elements in SA_i until <A^*>;
delete all elements except <A^*, O_j, E_j> or <A^*, O_j, S> in LE_i;
if E_{ab} is raised by the abortion handler **then**
$S(O_i)$: = X; <A^*, O_i, E_{ab}> $\rightarrow$ LE_i;
`Exception`(A^*, O_i, E_{ab}) $\Rightarrow$ all O_j in $G_A{}^*$;
else $S(O_i)$: = S; <A^*, O_i, S> $\Rightarrow$ LE_i;
`Suspended`(A^*, O_i, S) $\Rightarrow$ all O_j in $G_A{}^*$;
end if;
else if $S(O_i)$ = N **then**
$S(O_i)$: = S; <A^*, O_i, S> $\rightarrow$ LE_i;
`Suspended`(A^*, O_i, S) $\Rightarrow$ all O_j in $G_A{}^*$;
end if;
end if;
else retain the `Exception` or `Suspended` message till O_i enters A^*;
end if;
end if;

if O_i has all states, X or S, of other objects within A
//<A> is the top element in SA_i
and O_i has the biggest number among objects with the state X **then**
resolve exceptions in LE_i; *//find E in the exception tree*
`Commit`(A, E) $\Rightarrow$ all objects in $G_A - \{O_i\}$;
empty LE_i and handle E;
end if;

if O_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

4.3 Two Examples

Before we present the proof of the algorithm, let us consider two examples which demonstrate how our algorithm works.

Example 1. Assume that three objects, O_1, O_2, and O_3, participate in the action A. If exceptions E_1 and E_2 are raised in O_1 and O_2 concurrently, then the three objects will undertake the following steps:

O_1: sends `Exception` to O_2 and O_3. Later on, O_1 may receive `Exception` from O_2 and `Suspended` from O_3. It then waits for the `Commit` message. Once it receives `Commit`(A, E), it will start handling the resolving exception E.

O_2: sends `Exception` to O_1 and O_3. Later on, it may receive `Exception` from O_1 and `Suspended` from O_3. O_2 then resolves the exceptions E_1 and E_2 (because name(O_2) > name(O_1)), finds the resolving exception E, sends `Commit`(A, E) to O_1 and O_3, and starts handling E.

O_3: receives `Exception` from O_1 or O_2 (no matter who arrived first) and sends the `Suspended` message to O_1 and O_2 while suspending any normal computation. Once O_3 receives `Commit`(A, E) from O_2, it will start handling E.

Example 2. Assume that four objects, O_1, O_2, O_3, and O_4, participate in a set of nested CA actions (see Fig. 4). If two errors are detected in both O_1 and O_2 and, hence, exceptions E_1 and E_2 are raised simultaneously, then the four objects will undertake the following steps, respectively (this example demonstrates how a resolution started in the nested action A_3 is eliminated by the resolution performed by the containing action A_1):

O_1: sends `Exception` to O_2, O_3, and O_4. Later on, it receives suspended from O_3 and O_4 and `Exception`(A_1, O_2, E_3) from O_2 (assuming that an exception E_3 was signaled by the abortion handler in O_2 within Action A_2). O_1 then waits for `Commit` message (because name(O_2) > name(O_1)). Once O_1 receives `Commit`(A_1, E) from O_2, it will start handling E.

O_2: sends `Exception` to O_3 (but O_3 is a belated participant for Action A_3 in Fig. 4) and waits for some message(s) from O_3. Because O_3 is not yet in A_3, this `Exception` message cannot reach O_3. When O_2 receives `Exception` from O, it has to abort nested CA actions A_3 and A_2. Since the abortion handler in A_2 signals a further exception E_3 to A_1, O_2 will send `Exception`(A_1, O_2, E_3) to O_1, O_3, and O_4. During or after the abortion process it should receive `Suspended` from O_3 and O_4. O_2 then resolves the exceptions E_1 and E_3 (because name($O2$) > name(O_1)), finds the resolved exception E, sends `Commit`(A_1, E) to O_1, O_3, and O_4, and starts handling E.

O_3: receives `Exception` from O_1 and has to abort A_2. We assume no further exception is signaled by the abortion handler in O_3. Thus, O_3 sends `Suspended` to O_1, O_2, and O_4. It should also receive `Suspended` from O_4 and `Exception`(A_1, O_2, E_3) from O_2. Once O_3 receives `Commit`(A_1, E) from O_2, it will start handling E.

O_4: takes the action similar to that of O_3, but it is not a belated participant for Action A_3. Once O_4 receives `Commit`(A_1, E) from O_2, it will start handling E.

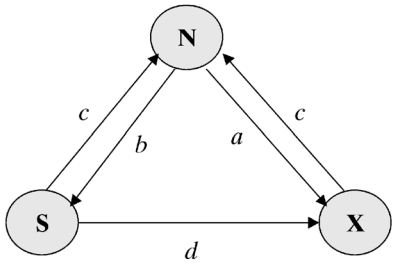
4.4 Correctness and Complexity

In order to prove the correctness of our algorithm, we restate the following assumptions:

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

Assumption 2. *FIFO message passing between objects is supported by the target system, i.e., two messages from object O_i will arrive at object O_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 objects in the system, T_{reso} be the upper bound of the time spent in resolving current exceptions, T_{abort} be the maximum



a: the object raises an exception
b: the object is suspended by another object
c: error recovery is successful
d: error recovery or abortion operation fails

Fig. 5. State changes of a participating object.

possible time for an object 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 (resolved) exception.

The correctness criteria for the proposed algorithm are two-fold: 1) There is no deadlock and 2) the resolved exception does cover all the exceptions raised concurrently within the outermost action. The no deadlock property will guarantee that the algorithm will produce a result eventually and the second criterion will ensure that the delivered result by the algorithm is actually the intended result. To further facilitate the understanding of our proofs, Fig. 5 shows a simple state transition diagram that illustrates possible state changes of a participating object.

We now show that no deadlock is possible in our proposed algorithm.

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

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

Proof. In order to prove the above bound, let us consider the worst case, i.e., an object 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 an object O_i in the innermost action raises an exception and changes its state into X. It will send the exception message to all the other objects, by Assumption 1, which will arrive at them in T_{mmax}. Since there are no further nested actions within the innermost action, any message from the other objects about an exception or suspended state would come to O_i in at most $2T_{mmax}$. Note that actual exception resolution may take T_{reso}. Therefore, O_i will receive a resolved exception and then complete exception handling in at most $(3T_{mmax} + T_{reso} + \Delta_{max})$.

If O_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 be required. After the abortion, O_i will send either an abortion exception or suspended message to other objects, which will arrive at them in $(T_{abort} + T_{mmax})$. O_i will then receive the resolved 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 thus have that

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

namely, object O_i will complete exception handling ultimately and leave the outermost CA action. □

By Lemma 1, we know that any object 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 resolved exception is a proper cover of the multiple concurrent exceptions that have been raised 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 object after it has started the resolution. Note that, from the proposed algorithm, the resolving object must know all the states (X or S) of the participating objects 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. □

Lemma 3. *Consider N objects that interact within nested CA actions. If multiple exceptions are raised concurrently, an ultimate resolved 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 resolved 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 *nmax*. Abortion will be no longer possible if the current active action A is the outermost CA action. By

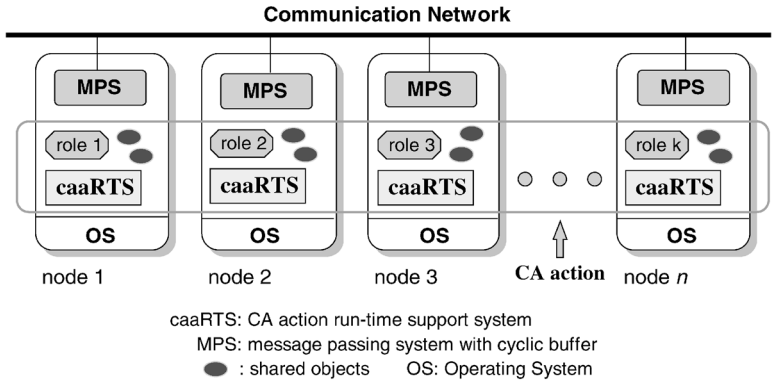


Fig. 6. Architecture for a prototype implementation.

Lemma 2, the exception resolution will start finally and no more new exceptions will be raised. □

From Lemmas 2 and 3, we know that a resolved exception will always cover all the currently existing 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 resolved 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 objects 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., $(N-N)$`Exceptions`, $(N-1)^2$`Suspendeds`, and $(N-1)$`Commit` messages;
2. When all N objects have the exceptions raised simultaneously, then the number of messages is still $(N+1) \times (N-1)$, 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 [26]. 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 CR algorithm [5] is of complexity $O(n_{max} \times N^3)$. Our previous algorithm in [26] could use $n_{max} \times 3N \times (N-1)$ messages. Our new algorithm is less complex because only one object (rather than all the objects) resolves multiple exceptions and only one object 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 objects that are responsible for performing resolution and producing the Commit messages. This only contributes a constant factor to its total complexity.

4.5 Implementation and Performance-Related Analysis

In order to implement the resolution algorithm and support reliable message passing, a practical way could be to use group communication and a group membership service [16]. Participating objects in a CA action could be treated as members of a closed group which multicasts service messages to all members. Another way would be the use of reflection and meta-objects [21]. The algorithm can be programmed as a meta-protocol connecting a set of meta-objects: one for each CA action participant. Exceptions, handlers, exception contexts should be first class objects. Such implementation would allow the dynamic change of different resolution algorithms (e.g., centralized or decentralized) to be transparent to the application programmer. We can use Open C++ [6] as a testbed which offers `ObjectCommunities` as a group communication feature and simplifies transactions as a particularly practical system for small experiments. Practical experiments of using this language to implement distributed replicated objects have been very successful [11].

We have recently implemented a prototype of the resolution mechanism and the 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 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.

Fig. 6 shows the system architecture for our prototype implementation. For each given CA action, its participating objects, called roles, are located on separate computing nodes. Communication and interaction between these roles are supported by a message passing subsystem (MPS). Received messages are first kept in a cyclic buffer before being consumed. A run-time system (caaRTS) that supports the execution of CA actions is established, together with MPS. This support system is decentralized in the sense that every distributed node has a copy of caaRTS. This basic CA action support offers the main CA action features: (nested) action entry points and exits, raising and signaling of

TABLE 1
Performance-Related Results

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

exceptions, abortion of (nested) actions, and calls to handlers. In addition, we have also implemented a basic protocol for participating objects to leave a CA action synchronously.

Note that 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 objects. The exception context in a participating object has an abortion handler and a set of action handlers. Every partition has a copy of the resolution function and of the resolution tree so as to ensure that the handlers for the same exception are called in all participating objects. The types common to all participating objects are declared in package Pure, which is used in compiling all packages; it includes the names of all exceptions, the lists of participants of all actions, the types declaring all object states, and all types of messages.

This prototype shows that the protocol is easy to implement: The entire implementation has about 1,000 lines of code, 800 of which form the partition executive and only 300 of those deal with exception handling and resolution. The protocol fits well with the structure of the modern distributed systems. It demonstrates how to extend the basic CA action executive by just adding new functionalities to it. The implemented protocol is general and can be easily moved to other distributed systems, perhaps with minor adjustment as to performance enhancement.

A simple application using nested CA actions was run on the prototype implementation. The application program was executed in a 20 times loop and the execution time measured (in seconds). One of the experiment sets was set up as follows: One role of a containing action raises an exception and its nested actions have to be aborted. A further exception is raised by the abortion handler. A high-order exception (covering both exceptions) is then identified and raised in all the roles. We varied three parameters, T_{mmax}, T_{abort}, and T_{reso}, in order to examine different effects of these factors on the execution time (where T_{mmax} is the maximum time of message passing between two concurrent execution threads in the system, T_{abort} is the maximum possible time for a thread to abort a nested action, and T_{reso} is the upper bound of the time spent in resolving multiple exceptions).

The values for these three parameters in our experiments are chosen based on two main considerations: 1) their reality and 2) our ability to analyze the overheads introduced by exception resolution by varying the parameters. The values chosen for T_{mmax} correspond well to several types of practical distributed systems. The interval of values for T_{abort} is decided based on the fact that the abortion of atomic actions in practice requires some access to discs in order to restore the state of affected objects. The values for parameter T_{abort} are chosen for the situations in which exception trees have a fairly large size so that any search in them can take a considerable time. For example, let T_{mmax} = 0.2*s*, T_{abort} = 0.1*s*, and T_{reso} = 0.3*s*; the execution of the application will take about 94.36*s*. Table 1 presents some typical data with varying values of T_{mmax}, T_{abort}, and T_{reso}.

The experimental data obtained are essentially consistent with the theoretical analysis presented previously. Fig. 7 illustrates effect on the total execution time of the application system. When T_{mmax} is limited within 1.0*s*, the cost of

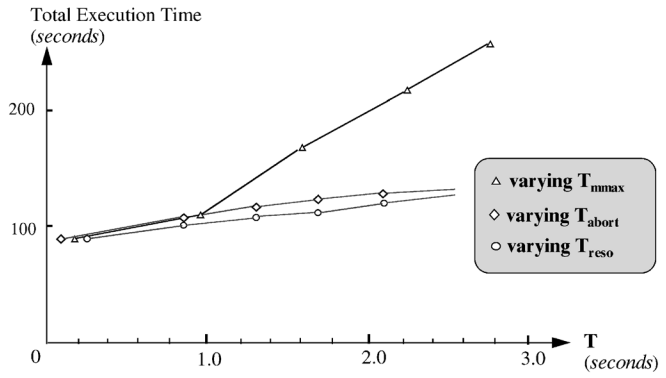


Fig. 7. Effect on total execution time.

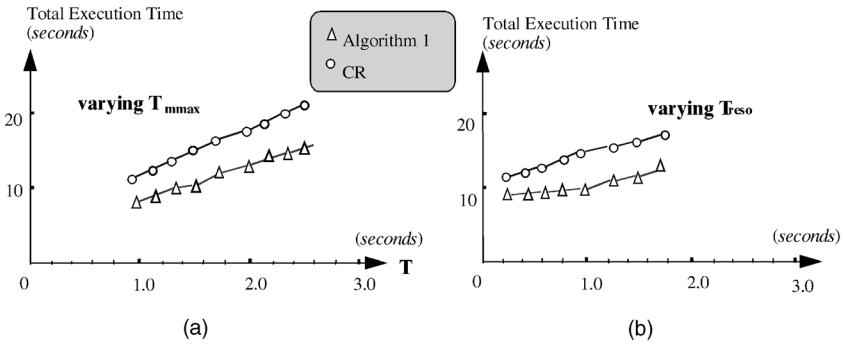


Fig. 8. Performance-related comparison of Algorithm 1 and the CR algorithm.

message passing has a minor impact on the total execution time. 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 by this given 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 is to compare Algorithm 1 with the original CR algorithm developed in [5]. The CR algorithm was implemented by modifying our algorithm appropriately and the same application program was used to collect the related data. The total execution time was then measured with respect to different T_{reso} and T_{mmax}. Fig. 8 demonstrates the major change of the total execution time when varying T_{mmax} with $T_{reso} = 0.3s$ and when varying T_{reso} with $T_{mmax} = 1.0s$. A big difference in the execution time can be observed even with a fixed N ($N = 3$ in our case). In particular, the difference becomes more obvious with an increase of the time of message passing (see Fig. 8a). The procedure for exception resolution is called $N \times (N - 1) \times (N - 2)$.

5 Conclusions

The concept of CA actions offers a general and convenient framework for designing distributed and concurrent OO software. This paper has focused on important technical details of concurrent exception handling and resolution under the CA action framework (though the results are generally applicable to other atomic action schemes). Our solutions are intended for a wide set of OO languages and for practical systems that interact with their environments; 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 nonconcurrent models for 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 [5], the authors presented just a draft of their resolution algorithm, without discussing assumptions under which the algorithm may work. The semantics of the operation of raising abortion exceptions in nested actions and of the resuming/suspending mechanism in such nested actions were not addressed clearly. We have developed an abortion mechanism that coordinates recovery measures used in both participating objects of nested actions and external atomic objects. A new distributed algorithm for exception resolution has been designed to handle concurrent raising of multiple exceptions in interacting objects.

We have applied our approach to realistic industrial applications by developing robust software to control different types of production cells. A production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI (Forschungszentrum Informatik) in 1993 [15] in order to evaluate different formal methods and to explore their practicability for industrial applications. Since then, this case study has attracted wide attention and has been investigated by over 35 different research groups. In 1996, the FZI presented the specification of two extended models, called the Fault-Tolerant Production Cell [17] and Real-Time Production Cell [18]. We have recently designed and implemented two control software systems for these case studies [27], [31]. Analytical and experimental results have demonstrated the feasibility of our approach to concurrent exception handling and showed that the approach makes it possible to reason about complex exceptional behaviour of a system in a very disciplined and structured way.

Future research directions would be in two primary areas in terms of the further development of the CA action concept. The first is the introduction of an appropriate linguistic mechanism for specifying (nested) CA actions in a distributed environment. Second, it is important to investigate the implementation-related issues with CA action's run-time support mechanisms.

Acknowledgments

This work was supported by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa) and IST Programme RTD Project 1999-11585 on Dependable Systems of Systems (DSoS) and by the EPSRC Flexx Project on Flexible Software.

REFERENCES

[1] *Information Technology—Programming Langauages—Ada.* Language and Standard Libraries, ISO/IEC 8652:1995(E), Intermetrics, Inc., 1995.

[2] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley, 1991.

[3] A. Avizienis, "The *N*-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.*, vol. 11, no. 12, pp. 1,491-1,501, Dec. 1985.

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

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

[6] S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," *Proc. European Conf. Object-Oriented Programming '93*, pp. 482-501, 1993.

[7] F. Cristian, "Understanding Fault-Tolerant Software," *Comm. ACM*, vol. 34, no. 2, pp. 56-78, 1991.

[8] F. Cristian, "Exception Handling and Tolerance of Software Faults," *Software Fault Tolerance*, M. Lyu, pp. 81-107, Wiley, 1995.

[9] Q. Cui and J. Gannon, "Data-Oriented Exception Handling," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 393-401, May 1992.

[10] C. Dony, "Exception Handling and Object-Oriented Programming: Towards a Synthesis," *Proc. European Conf. Object-Oriented Programming/Object-Oriented Programming, Systems, Languages, and Applications '90*, pp. 322-330, 1990.

[11] J. Fabre, V. Nicomette, T. Perennou, R.J. Stroud, and Z. Wu, "Implementing Fault Tolerant Application Using Reflective Object-Oriented Programming," *Proc. 25th Int'l Symp. Fault-Tolerant Computing*, pp. 489-598, June 1995.

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

[13] P. Jalote, "Using Broadcast for Multiprocess Recovery," *Proc. Sixth Distributed Computing Systems Symp.*, pp. 582-589, 1986.

[14] P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, second ed. Wien: Springer-Verlag, 1990.

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

[16] L. Liang, S.T. Chanson, and G.W. Neufeld, "Process Groups and Group Communications: Classifications and Requirements," *Computer*, vol. 23, no. 2, pp. 56-66, Feb. 1990.

[17] A. Lötzbeyer, "Task Description of a Fault-Tolerant Production Cell," version 1.6, available from http://www.fzi.de/prost/projects/korsys/korsys.html, 1996.

[18] A. Lötzbeyer and R. Mühlfeld, "Task Description of a Flexible Production Cell with Real Time Properties," Forschungszentrum Informatik, Karlsruhe, Germany (ftp.fzi.de/pub/prost/projects/korsys/task_descr_flex_2_2.ps.gz), 1996.

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

[20] *Software Fault Tolerance*, M.R. Lyu, ed. Wiley, 1995.

[21] P. Maes, "Concepts and Experiments in Computational Reflection," *Proc. conf. Object-Oriented Programming, Systems, Languages, and Applications 87, ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 147-155, 1987.

[22] B. Meyer, *Eiffel: The Language.* New York: Prentice Hall, 1992.

[23] L.C. Paulson, *ML for the Working Programmers.* Cambridge Univ. Press, 1996.

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

[25] A. Romanovsky, I. Shturtz, and V. Vassilyev, "Designing Fault-Tolerant Objects in Object-Oriented Programming," *Proc. Seventh Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS EUROPE 92)*, pp. 199-205, 1992.

[26] A. Romanovsky, J. Xu, and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems," *Proc. 16th IEEE Int'l Conf. Distributed Computing Systems*, pp. 545-552, May 1996.

[27] A. Romanovsky, J. Xu, and B. Randell, "Coordinated Exception Handling in Real-Time Distributed Object Systems," *Int'l J. Computer Systems Science and Eng.*, special issue on object-oriented real-time distributed systems, vol. 14, no. 4, pp. 197-208, 1999.

[28] C.M.F. Rubira, "Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation," PhD thesis, Univ. of Newcastle upon Tyne, 1994.

[29] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R.J. Stroud, and Z. Wu, "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," *Proc. 25th Int'l Symp. Fault-Tolerant Computing*, pp. 499-508, June 1995.

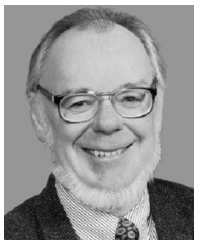
[30] J. Xu, B. Randell, C.M.F. Rubira-Calsavara, and R.J. Stroud, "Toward an Object-Oriented Approach to Software Fault Tolerance," *Recent Advances in Fault-Tolerant Parallel and Distributed Systems*, D.K. Pradhan and D.R. Avresky, eds., IEEE CS Press, pp. 226-233, Sept. 1995.

[31] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver, and F. von Henke, "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions," *Proc. 29th Int'l Symp. Fault-Tolerant Computing*, pp. 68-75, June 1999.

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

Jie Xu received the PhD degree from the University of Newcastle upon Tyne in advanced fault-tolerant software. He is a lecturer in Computer Science at the University of Durham, United Kingdom. From 1990 to 1998, Dr. Xu was with the Computing Laboratory at Newcastle, where he was promoted to senior researcher in 1995. He moved to a lectureship at Durham in 1998 and cofounded the Distributed Systems Engineering Group and the DPART Laboratory supporting highly dependable distributed computing. Dr. Xu has published more than 100 book chapters and research papers in areas of system-level fault diagnosis, exception handling, software fault tolerance, and large-scale distributed applications. He has been involved in several research projects on dependable distributed computing systems, including two EC-sponsored ESPRIT BRA projects and one ESPRIT Long Term Research project. He is principal investigator of the FTNMS project on fault-tolerant mechanisms for multiprocessors and co-Investigator of the EPSRC Flexx project on highly dependable and flexible software. Dr. Xu is an editor of *IEEE Distributed Systems OnLine*, chair of the IEEE SRDS Workshop on Reliable Distributed Object Systems, and tutorial speaker for the IEEE/IFIP International Conference on Dependable Systems and Networks. He has given invited lectures in international colloquiums and served as session chair and PC member of various international conferences and workshops.

Alexander Romanovsky received an MSc degree in applied mathematics from Moscow State University in 1976 and a PhD degree in computer science from St. Petersburg State Technical University in 1988. He was with this university from 1984-1996, doing research and teaching. In 1991, he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993, he was a visiting researcher at the Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-1994, he was a postdoctoral fellow in the Department of Computing Science, the University of Newcastle upon Tyne (United Kingdom). In 1992-1998, he was involved in the Predictably Dependable Computing Systems ESPRIT Basic Research Action and the Design for Validation ESPRIT Basic Project. In 1998-2000, he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. He is now a senior research associate with the Department of Computing Science, University of Newcastle upon Tyne, working on the Dependable Systems of Systems EC IST RTD Project. Dr. Romanovsky coedited the *IEEE Transactions on Software Engineering* special issue on current trends in exception handling and organized ECOOP 2000 Workshop on Exception Handling in Object-Oriented Systems; he is now a program committee member of the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. His research interests include software fault tolerance, software diversity, concurrent programming, concurrent object-oriented and object-based languages, real time systems, exception handling, operating systems, software engineering, and distributed systems. He has coauthored more than 120 scientific papers, book chapters, reports, and a patent in these and related areas.



Brian Randell graduated in mathematics from Imperial College, London, in 1957 and joined the English Electric Company, where he led a team which implemented a number of compilers, including the Whetstone KDF9 Algol compiler. From 1964 to 1969, he was with IBM, mainly at the IBM T.J. Watson Research Center in the United States, working on operating systems, the design of ultra-high speed computers, and system design methodology. He then became a professor of computing science at the University of Newcastle upon Tyne, where, in 1971, he set up the project which initiated research into the possibility of software fault tolerance, and introduced the "recovery block" concept. Subsequent major developments included the Newcastle Connection and the prototype Distributed Secure System. He has been Principal Investigator on a succession of research projects in reliability and security funded by the Science Research Council (now Engineering and Physical Sciences Research Council), the Ministry of Defence, and the European Strategic Programme of Research in Information Technology (ESPRIT). Most recently, he has had the role of Project Director of DeVa, the ESPRIT Long Term Research Project on Design for Validation, and of CaberNet, the ESPRIT Network of Excellence on Distributed Computing Systems Architectures, and is now leading an IST Research Project on Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) and an IST RTD Project on Dependable Systems of Systems (DSOS). He has published nearly 200 technical papers and reports and is the coauthor or editor of seven books.