Exception Handling and Resolution in Distributed Object-Oriented Systems

Alexander Romanovsky
Applied Mathematics Dept.
St. Petersburg State Technical University

Jie Xu and Brian Randell
Dept. of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU (UK)

In:
Proceedings of the
16th IEEE International Conference
on
Distributed Computing Systems (ICDCS '96),
Hong Kong, China, 27-30 May 1996 pp. 545-552

IEEE Computer Society Press 1996

© 1996 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Exception Handling and Resolution in Distributed Object-Oriented Systems

Alexander Romanovsky
Applied Mathematics Department
St. Petersburg State Technical University

Jie Xu and Brian Randell
Department of Computing Science
University of Newcastle upon Tyne, UK

Abstract
We address the problem of how to handle exceptions in distributed object-oriented systems. In a distributed computing environment exceptions may be raised simultaneously and thus need to be treated in a coordinated manner. We take two kinds of concurrency into account: 1) several objects are designed collectively and invoked concurrently to achieve a global goal, and 2) concurrent 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 \(O(N^2)\) messages, and is fully object-oriented.

Key Words — Concurrent exception handling, distributed systems, exception resolution, nested atomic actions, object-oriented programming.

1: Introduction

The coordinated atomic action (or CA action) concept has been recently developed at Newcastle University [23][24]. A CA action coordinates error recovery between multiple interacting objects in a concurrent object-oriented (OO) system by integrating two complementary concepts — conversations [19] (together with concurrent exception handling [4]) and transactions [15]. We report here a continuation of research on CA actions, concentrating upon technical details of exception handling and resolution. The major focus of this paper includes: 1) a brief survey of existing OO exception handling mechanisms, 2) exception context, declarations and propagation concepts which are coherent with the CA action framework, 3) exceptions within and between nested CA actions, and 4) resolution of concurrent exceptions in distributed OO systems. In 1986, Campbell and Randell [4] introduced the first algorithm (referred to as the CR algorithm in this paper) for exception resolution in process-oriented concurrent systems. We discuss a number of difficulties and issues involved in the algorithm and present a new distributed algorithm relying on strictly-defined, practically-oriented assumptions. The proposed algorithm is easier to implement and of lower complexity than the original solution, and is object-oriented.

2: Faults, Errors and Exception Handling

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. Both hardware and software faults are taken into account: hardware faults can be caused by crashes or transient errors of nodes or the communication network, and software faults by poor designs. We do not assume fail-stop semantics and any erroneous information may be spread through communication channels.

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 mechanisms) is based on the use of redundant data that repairs the system by analyzing the detected errors and putting the system into a correct state. In contrast, backward error recovery returns the system to a previous error-free state without requiring detailed knowledge of the errors.

An exception mechanism is a language control structure that allows programmers to describe the replacement of the standard program execution by an exceptional execution when occurrence of an exception (i.e., inconsistency with the program specification) is detected (see [7] for a rigorous and thorough discussion). It is usually considered as an essential part of a modern language (see Ada95 [11], C++, Eiffel [18] for example).

For any given exception mechanism, exception contexts [4] (i.e., regions in which the same exceptions are treated in the same way) must be declared. Very often they are blocks or procedure bodies. Each context has a set of associated exception handlers, one of which will be called when a corresponding exception is raised. There are two categories of exception mechanisms. The termination 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. 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 fault tolerance are much more 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]. Thus, the time of message passing is not negligible. Special protocols must be designed in order to ensure coordinated error recovery in spite of physical distribution.

2.2: Conversations and Concurrent Exception Handling

The concept of a conversation was first proposed in [19] 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 entry. 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 may use an alternate algorithm. Processes can enter a conversation asynchronously but must leave it at the same time when all the processes satisfy the acceptance test. This general idea has been extended in several ways in later research (see [16] for comprehensive discussion).

A systematic approach to handling exceptions in concurrent systems is developed in [4] by extending the conversation scheme and the well-known atomic action paradigm [13]. A set of exceptions is associated with each action (i.e. conversation). Each process participating in an action has a set of handlers for (all or some of) pre-defined exceptions. When any of these exceptions is raised in a 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 in [4] are critical since several independent exceptions can be raised simultaneously, or several errors detected which could be the symptoms of a different, more serious fault. In principle, the exception tree structure [4] is more appropriate than exception priorities for resolving concurrent exceptions. (An exception tree includes all exceptions associated with an 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.)

The authors of [4] suggested two methods of coping with a nested action in the situation that an exception is raised in the process not yet entering a nested action, but some pre-defined participants of the same action where the exception is raised have entered the nested one. A natural method is to wait until the nested action is completed because the execution of the nested action is in notion invisible and indivisible for the containing action (see Figure 1 (a)). An alternative method is to implement an abortion handler in each process taking part in a nested action and to raise an abortion exception in all participants of the nested action, as shown in Figure 1(b). After the execution of a resolution algorithm, either the handlers for the same exception are started for all action participants, or a failure exception is raised if no corresponding handlers are found. 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 inside the nested action would wait forever for it to continue execution. Secondly, for real-time systems it seems to be more predictable to abort the nested action than to wait for its completion [4].

![Figure 1. Two methods for treating nested actions while an exception raised.](image)

There has been relatively little work on implementations of distributed coordinated error recovery. Implementations of distributed conversations are discussed in [12][25]. Of these [25] focused on two particular conversation schemes (i.e. the name-linked recovery block and the abstract data type). The work in [12] discussed a distributed implementation of the conversation scheme using broadcasts, assuming that all processes enter a conversation simultaneously. Both of these approaches cannot be used directly to implement the CA action scheme because they are directed into some particular schemes, with no support for forward error recovery, and not intended for concurrent OO systems.

2.3: Exception Mechanisms in OO Languages

Exceptions in actual OO languages can be declared as classes, objects or strings [9][22], while handlers can be declared and attached to the level of statements, methods, classes or objects. Some languages like C++, Modula-3 and Arche [11] only allow exception handlers to be attached to statements, and the others such as Lore [9], Eiffel [18], Guide [3], extended C++ [20], extended Ada [8] permit handlers to be attached to methods, objects and/or classes. Such flexible attachment has numerous advantages: 1) a clear separation of an object's abnormal behaviour from its normal one, in accordance with the concept of an idealised fault-tolerant component [13]; 2) object/class recovery provided at the object level; 3) exceptions associated with
types; and 4) software layering which helps the design of fault-tolerant systems. There is evidence from practice [8] 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. To do this, 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 (the resumption model), whereas extended C++ propagates the class exception along the object creation chain (which may coincide with the call chain).

There are only a few concurrent OO languages, such as Ada95 [1], Arch and Guide, that have exception handling features. Ada95 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. Arch permits user-defined resolution of multiple exceptions amongst a group of objects that belong to different implementations of a given type; however it is not generally applicable to the coordination of multiple interacting objects with different types. We need an OO exception model which can be applied to any group of interacting objects whether or not of the same type.

3: CA Actions: OO Concurrency and Coordination

The CA action scheme presents a general technique for achieving fault tolerance in concurrent (and distributed) OO software by integrating conversations, transactions and exception handling into a uniform framework [23]. 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 [13]. Several objects can be designed collectively by different programmers (or teams) and invoked concurrently in order to achieve certain joint goals. These objects operate within a CA action. Competitive concurrency may co-exist in such systems where two or more separately designed, concurrent objects can compete for the same system resources (i.e. objects).

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 [15]). Objects that are external to a CA action and can be shared with other actions and objects concurrently must be atomic and individually responsible for their own integrity.

3.1: Basic Exception Model for CA Actions

In our model, exceptions may be declared in any of the ways discussed in Subsection 2.3. The exceptions that can be raised within a CA action are declared together with the action declaration. Handlers should be associated with participating objects of the CA action so that 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 properties of a nested transaction in the terms of atomic objects. Note that 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 easily enforced. Exceptions can be propagated along nested exception contexts, corresponding to the chain of nested CA actions.

However, 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 the operation is completed, then operation-level exceptions would be appropriate to 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 — in any exceptional situations, handlers take over the duties of participating objects of a CA action and complete the action either successfully or by signalling a failure exception to the containing action (Note that an exception is raised within a CA action, but signalled between nested actions).

Because a CA action may cope with two kinds of concurrency, the external atomic 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 right after recovery. It is particularly important to notice that an exception within the CA action does not necessarily cause restoration of all the atomic objects to their prior states. The appropriate exception handlers may be able to put them into new valid states (see Figure 2 (a)). If the handlers fail to restore them, a failure exception must be signalled to the containing CA action. Recall that an associated transaction will be issued when a CA action starts a new attempt [23]. Thus, the exception handlers could call three functions explicitly — abort, commit and start. This allows easy use of retry operations (e.g., those used in Guide and Eiffel) in the CA action scheme. It is more straightforward to ensure the consistency of atomic objects while considering backward error recovery. The start, abort and commit functions could be called implicitly, corresponding to three different cases that an attempt of the CA action starts, or fails or passes the
acceptance test, as illustrated in Figure 2 (b). In principle, object programmers have the freedom of choosing appropriate policies in order to guarantee such consistency. 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.

![Diagram of Error Recovery](image)

Figure 2: Error recovery in external atomic objects.

### 3.2: Resolution of Concurrent Exceptions

In order to deal with concurrent exceptions within a CA action, we follow the general framework developed in [4] with some adjustments to distributed OO systems. It is assumed that a mechanism exists such that 1) a set of handlers can be associated correctly with each participating object and 2) all exceptions be structured into an exception tree, declared as part of the CA action implementation. The tree must exist at run time so as to allow concurrent exceptions to be resolved. After the resolution, the same handlers are called in all participating objects, and they either cooperatively recover the objects and fulfill the function specified by the action specifications or signal a failure exception to the containing CA action.

Our belief in the importance of the resolution mechanism for distributed systems is based on:

- It is practically difficult to interrupt 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 the initial exception is much higher in a distributed system than in a uni- or a multi-processor system with common main memory.

- Usually the latent period of an error is not negligible, and erroneous information can be easily spread within a CA action; several errors occurring concurrently in different objects can be the symptoms of a different, more serious fault [4].

- Different participating objects can be involved in nested CA actions of different levels so that their exception contexts may be different.

- In distributed systems the overall hardware failure probability is relatively higher and they are also more difficult to program without design faults than centralised systems [6].

- 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 the nodes can be damaged. On the other hand, because participating objects of a CA action were designed cooperatively from a given specification, a specification error or cooperative misunderstanding during the design could affect several or all the participating objects.

An exception hierarchy-based approach is therefore needed in order to find the exception "covering" all the exceptions raised concurrently. This is why a distributed resolution scheme is required to determine the correct recovery strategy and to involve all the participating objects of a CA action in the recovery activity. In an object-oriented fashion, an exception tree could be specified as the hierarchy of exceptions where exceptions are classes and declared by subtyping [21].

### 3.3: Issues with the CR Algorithm

There are three sources of exceptions defined in [4]. The first source is of exceptions which are raised during the execution of the application code; the second is of exceptions signalled by the nested action; and the third is of exceptions which are raised when participants of an atomic action received information about an exception raised in some participant, but have no handler for it; so they have to examine the exception tree, find and raise an appropriate exception (for which there is a handler). This is mainly because not all of the exceptions declared in the action declaration necessarily have corresponding handlers in each action participant (although each participant could contain the use of a default handler). In [4] each participant knows only a reduced local tree of exceptions with specific handlers and has to look through the tree 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 any exception will always lead to further exceptions until the root of the exception tree is reached (see examples in [21]). 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 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. We found that this is not a trivial problem. Consider four concurrent objects, $O_0$, $O_1$, $O_2$, and $O_3$, in several nested atomic actions (see Figure 3). If $O_1$ detects an error and thus raises an exception, $O_0$, $O_2$ and $O_3$ will be informed of the exception. Since $O_1$ may know nothing about actions $A_2$ and $A_3$, $O_2$ and $O_3$ are responsible for actual abortion of these actions. Several problems (which were not adequately discussed in [4]) are as follows:

1. $A_3$ should be aborted before $A_2$;

548
2) both \( O_2 \) and \( O_1 \) 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 \( A_3 \)), \( O_2 \) and \( O_3 \) could wait for it to execute the abortion of \( A_2 \) and \( A_3 \) (so, abortion handlers must be used which 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 \( O_3 \), see the reason above) must participate in error recovery, so the lower level resolution by \( O_3 \) should be ignored when the resolution is started by \( O_1 \) within \( A_1 \); In fact, since belated participants can participate in the resolution only when they enter the nested action, the entire protocol execution for resolution should be delayed;
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 signalled 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 signalling from within the nested action \( A_3 \), but not from \( A_2 \), will be ignored for the resolution performed by Action \( A_1 \)).

We shall describe a new algorithm for exception resolution below, based on a set of precisely-defined assumptions, which allow us to overcome all the above-mentioned problems.

4: A Distributed Algorithm for OO 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 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 resolution tree (which is declared statically). Each object also has a name list of the nested actions it participates in. The currently innermost action for the object is called an active CA action.

In order for Action \( A_i \) to abort a nested action, an abortion exception is raised within the nested action and any activity of the nested action is 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 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 \( A_i \). 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 exceptional 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 \( A_n \), 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 \( A_n \). The handler for the exception is intended to perform the simple “last-will” recovery (see the discussion in [4]). It is allowed that the abortion handlers of the nested action \( A_{i+1} \) signal different exceptions to the containing action \( A_i \) (though, according to [4], we believe that in practice the same exceptions should be signalled 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 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 \( G_{CA-action} \) as the group of all participating objects \( \{O_1, O_2, \ldots, O_i, \ldots \} \), 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 amongst objects which raised exceptions, and the chosen object will be responsible for exception resolution. Let \( A \) be the active action of \( O_i \) and \( G_A \) be the corresponding set of participating objects. Assume that each object \( O_i \) has the following data structures:

- list \( LE_i \) — records exceptions that have been raised;
- list \( LO_i \) — records information about objects waiting for completion or abortion of their nested CA actions;
- list \( LP_i \) — records information about participating objects which have acknowledged the message sent by \( O_i \); and
- stack \( SA_i \) — stores the exception context and the exception tree corresponding to each of nested CA actions.
In the interests of simplicity and brevity, we assume that application-related message passing is treated independently. In our algorithm 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 in it;

**HaveNested**\( (O_i, A) \) is sent by each object \( O_i \) that is in a nested action of Action \( A \) to all participating objects of Action \( A \), and \( O_i \) then starts abortion of nested actions;

**NestedCompleted**\( (A, O_i, E) \) is sent by each object \( O_i \) which sent message **HaveNested** earlier. This message is sent to all participating objects of \( A \) and informs them of the exception \( E \) which may be signalled by abortion handlers of a nested CA action;

**ACK**\( (O_i) \) is sent by each object \( O_i \) to the object which sent either **Exception** or **NestedCompleted** to it earlier;

**Commit**\( (E) \) is sent by a chosen object to all participating objects after it completes resolution of all exceptions, where \( E \) is the resulting exception. A handler for \( E \) will be called by each object once it receives **Commit**.

### 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 of a nested CA action. During a given CA action, 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 \( O_i \)), \( s \) = Suspended (if \( O_i \) in normal state receives a message about an exception in other objects), and \( r \) = Ready (if \( O_i \) in the \( x \) state receives **ACK** from all objects in \( G_A \)). In addition, "\( \rightarrow \)" stands for "put in" and "\( \Rightarrow \)" for "sent to" in the description of our algorithm.

**Algorithm:**

```plaintext
For any \( O_i, S(O_i) := n \); and empty \( LE_i, LO_i, LP_i, SA_i \);
loop
  if \( O_i \) enters A then
    \(<A> \rightarrow SA_i \); consume messages having arrived;
  end if;
  if \( O_i \) completes A then
    delete last element in \( SA_i \); 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\( (A, O_i, E_i) \Rightarrow \) all \( O_j \) in \( G_A \);
  end if;
  if \( O_i \) receives Exception\( (A, O_i, E_i) \) or
    HaveNested\( (O_i, A) \) then
    if \( O_i \) is in the action nested within \( A \) then
      HaveNested\( (O_i, A) \Rightarrow \) all \( O_j \) in \( G_A \),
      abort all nested actions until \( A \);,
    else \( E_i \) may be raised by abortion handlers
    empty \( LE_i, LO_i, LP_i \);
    NestedCompleted\( (A, O_i, E_i) \Rightarrow \) all \( O_j \) in \( G_A \),
    if \( E_i \) \# null then \(<A, O_i, E_i> \rightarrow LE_i \);
    \( S(O_i) := x \); else \( S(O_i) := s \);
  end if;
end loop
```

if \( S(O_i) = n \) then \( S(O_i) := s \);
end if;
if \( O_i \) receives Exception\( (A, O_i, E_i) \) then
  \(<A, O_i, E_i> \rightarrow LE_i \); ACK\( (O_i) \Rightarrow O_j \);
else \( <O_j, A> \rightarrow LO_i \);
  clean up messages related to nested actions;
end if;
if \( O_i \) receives NestedCompleted\( (A, O_i, E_i) \) then
  ACK\( (O_i) \Rightarrow O_j \);
  if \( E_i \# \) null then \(<A, O_i, E_i> \rightarrow LE_i \);
end if;
if \( O_i \) receives ACK\( (O_i) \) then \( <O_j> \rightarrow LP_i \);
end if;
if \( S(O_i) := x \) and
  \( O_i \) received NestedCompleted from all \( O_j \) in \( LO_i \) and
  \( O_i \) received ACK\( (O_i) \) from all \( O_j \) in \( G_A \) then \( S(O_i) := r \);
end if;
if \( S(O_i) := r \) and
  \( O_i \) has the biggest number among all objects that
  raised exceptions then
  resolve exceptions in \( LE_i \);
  // find \( E \) in the exception tree
  commit\( (E) \Rightarrow \) all \( O_j \) in \( G_A \),
  empty \( LE_i, LO_i, LP_i \); start handler for \( E \);
end if;
if \( S(O_i) := r \) and \( O_i \) receives commit\( (E) \) then
  empty \( LE_i, LO_i, LP_i \); start handler for \( E \);
end if;
if \( S(O_i) := s \) and \( O_i \) receives commit\( (E) \) then
  wait until all exception messages are handled;
  empty \( LE_i, LO_i, LP_i \); start handler for \( E \);
end if;
end loop
```

### 4.3: Examples

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 \( A_1 \). If exceptions \( E_1 \) and \( E_2 \) are raised in \( O_1 \) and \( O_2 \) concurrently, then the three objects will take the following actions:

- \( O_1 \) sends **Exception**\( E_2 \) and \( O_3 \), then receives **ACKs** for the exception message from them. During this message passing, it may receive **Exception** from \( O_2 \) and send an **ACK** to \( O_2 \). Then \( O_1 \) waits for the **Commit** message. Once it receives **Commit**\( (E) \) it will start the handler for \( E \).

- \( O_2 \) sends **Exception**\( E_1 \) and \( O_3 \), then receives **ACKs** for the exception message from them. During this message passing, it may receive **Exception** from \( O_1 \) and send an **ACK** to \( O_1 \). Then \( O_2 \) resolves the exceptions \( E_1 \) and \( E_2 \) (because name\( (E_2) > \) name\( (E_1) \)); finds the resolving exception \( E \), sends **Commit**\( (E) \) to \( O_1 \) and \( O_3 \), and starts the handler for \( E \).

- \( O_3 \) receives **Exceptions** from \( O_1 \) and \( O_2 \), sends **ACKs** for two exception messages to them. Once \( O_3 \) receives **Commit**\( (E) \) from \( O_2 \), it will start the handler for \( E \).

**Example 2:** Assume that four objects \( O_1, O_2, O_3 \) and \( O_4 \) participate in nested CA actions (see Figure 4). If two errors are detected in \( O_1 \) and \( O_2 \) and exceptions \( E_1 \) and \( E_2 \)
are raised simultaneously, the four objects will take the following actions respectively (this example demonstrates how a resolution started in the nested action \(A_3\) was eliminated by the resolution of the containing action \(A_1\):

![](image)

**Figure 4. Four objects participating in nested actions.**

- **\(O_1\):** sends `Exception` to \(O_2, O_3,\) and \(O_4,\) and receives `HaveNested` from them. Later on, it should receive `NestedCompleted` from all \(O_2, O_3,\) and \(O_4,\) and then send `ACKs` to them. \(O_1\) should also receive `ACKs` from \(O_2, O_3,\) and \(O_4\) for its `Exception` messages. (Suppose that an exception \(E_1\) was signalled by the abortion handler in \(O_2\) within \(A_2\).) After message passing, \(O_1\) waits for `commit` message (since \(name(O_2) > name(O_1))\). Once \(O_1\) receives `commit(E)` from \(O_2,\) it will start the handler for \(E\).

- **\(O_2\):** sends `Exception` to \(O_1\) (but \(O_2\) is a belated participant for action \(A_3\) in Figure 4), and waits for the `ACK` from \(O_1\). Because \(O_1\) is not yet in \(A_3\), this `Exception` message cannot reach \(O_1\). However, \(O_2\) receives `Exception` from \(O_1\) and has to send `HaveNested` to \(O_1, O_3,\) and \(O_4,\) It then aborts nested CA actions \(A_1, A_2,\) and \(A_3.\) Since the abortion handler in \(A_2\) has signalled an exception \(E_2\) to \(A_1, \) \(O_2\) will send `NestedCompleted` to \(O_1, O_3,\) and \(O_4,\) with the information about \(E_2.\) It also sends an `ACK` to \(O_1,\) for the previous `Exception` message. \(O_2\) should then receive `ACKs` for its `NestedCompleted` from all \(O_1, O_3,\) and \(O_4,\) After the message exchange, \(O_2\) resorces the exceptions \(E_1, E_2\) (because \(name(O_2) > name(O_1)),\) finds the `resolve` decision `commit(E)` to \(O_1, O_3,\) and \(O_4,\) and starts the handler for \(E.\)

- **\(O_3\):** receives `Exception` from \(O_1\) and has to send `HaveNested` to \(O_2, O_3,\) and \(O_4,\) it then aborts \(A_2,\) Assume no further exception is signalled by the abortion handler in \(O_3.\) Thus \(O_3\) sends `NestedCompleted` to \(O_1, O_3,\) and \(O_4,\) and an `ACK` to \(O_1,\) for the previous `Exception`. When \(O_3\) receives `HaveNested` from \(O_2,\) it will clean up the message `Exception` sent by \(O_2.\) Later on, it should receive `NestedCompleted` from \(O_2\) and send an `ACK` to \(O_2.\) It should receive `ACKs` for its `NestedCompleted` from \(O_1, O_2,\) and \(O_4,\) as well. Once \(O_3\) receives `commit(E)` from \(O_2,\) it will start the handler for \(E,\)

- **\(O_4\):** 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(E)` from \(O_2,\) it will start the handler for \(E.\)

4.4: Analysis and Comparison

The 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 an exception is raised without nested actions, the number of messages is \(3 \times (N - 1),\) i.e., \((N - 1)\) `Exception`s, \((N - 1)\) `ACKs`, and \((N - 1)\) `Commit` messages;
2. when one exception is raised and all other objects have nested actions, then the number of messages is \(3 \times (N - 1),\) i.e., \((N - 1)\) `Exception`s, \((N - 1)\) `ACKs`, \((N - 1)\) `nestedCompleted`s, \((N - 1)\) `Commit` messages; and
3. when all \(N\) objects have the exceptions raised simultaneously, then the number is \((N - 1) \times (2N + 1),\) i.e., \(N \times (N - 1)\) `Exception`s, \(N \times (N - 1)\) `ACKs`, and \((N - 1)\) `Commit` messages.

Now let \(P: [1, N]\) be the number of objects in which exceptions have been raised, and \(Q: [2, N - 1]\) the number of the objects with the nested actions. Then the number of total messages is: \((N - 1) \times (2P + 3Q + 1).\)

Note that the CA algorithm [4] is of complexity \(O(N^2)\). Our algorithm is less complex because only one object resolves multiple exceptions and 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. In general, our algorithm (and the CA algorithm as well) will have no overhead if an exception is not raised.

However, during the abortion of nested CA actions the associated transaction supporting system should abort the corresponding operations on external atomic objects. Abortion handlers must be responsible for sending the related abortion messages to the supporting system. It is therefore worth notice that the proposed algorithm may suffer some delays because of the execution of abortion handlers in nested actions. This is because levels of nesting cannot be estimated in any way (their structures are hidden and atomic with respect to the containing actions) and also due to possible belated participants.

In [4] 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, especially with respect to distributed systems. In contrast, our proposal clarifies several key points of the original algorithm and presents a new strictly-defined algorithm with lower complexity.

The Arche language [11] 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 raised but 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 directly for CA actions since it
supports only a limited kind of concurrency and is not suitable for cooperative concurrency and recovery of several objects with different types. Moreover, parameterized exceptions, perhaps suitable for Arche, cannot be correctly resolved because the handler for a combined exception which was not raised may still be needed after resolution. As opposed to the Arche solution, our algorithm is transparent to programmers (like the CR algorithm) — only responsibility of the programmer is to implement abortion handlers for each nested CA action.

4.5: Implementation Issues

Due to the limitation of space, we have to omit the correctness proof of our algorithm. Here we address the related implementation issues briefly. To implement the resolution algorithm and support reliable message passing a practical way could be to use group communication and group membership services [14]. Participating objects in a CA action could be treated as members of a closed group which multicasts service messages to all members. If a reliable multicast [2] can be used, ACK messages will be no longer necessary and so communications in our algorithm would consist of only several multicasts (Exception, Commit, HaveCompleted, and NestedCompleted).

Another way of implementing the resolution algorithm would be the use of reflection and meta-objects [17]. 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, transparent to the application programmer. We are considering Open C++ [5] 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 [10].

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 exception handling and resolution under the CA action framework. Our solutions are intended for a wide set of OO languages and for practical systems that interact with their environments which typically are incapable of simple backward recovery.

How to correctly cope with nested CA actions in exceptional situations is a delicate problem, especially in a distributed computing environment. 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 in order to handle concurrent raising of exceptions in interacting objects.

Acknowledgements

This work was supported by the ESPRIT BRA 3092 and 6362 on Predictably Dependable Computing Systems (PDCS), by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa), and by the Royal Society under Project 638072.

References