Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems

Alexander Romanovsky — *University of Newcastle upon Tyne*
Jörg Kienzle — *Swiss Federal Institute of Technology*

1. Introduction

Considerable effort has been devoted to developing exception handling models for sequential object-oriented (OO) systems, so a common understanding exists in many topics in the field. Many practical systems have been designed using these features. The situation is different in concurrent OO systems. Although several schemes combining concurrency and exception handling have been proposed, this research is still scattered and the majority of concurrent systems use sequential exception handling. The latter is abnormal because it is our belief that exception handling features should correspond to the programming feature used in system design. The choice of a way to introduce exception handling into such systems depends on the way concurrent systems are to be developed and structured because exception handling is an issue of the system design, and language features should assist in and impose proper design. Exception handling is closely coupled with program structure and therefore the way in which the dynamic execution of concurrent systems is structured influences possible ways of introducing exception handling into such systems.

Several schemes have been proposed for introducing different units of system structuring into concurrent OO systems, but only rarely do they incorporate exception handling features. And even when they do they neither provide a general exception handling model nor fit the main principles of OO programming properly. Although this is an area of very active research, there are still many unclear points and unsolved problems here. A general common understanding does not seem to exist. The purpose of this survey is to outline the existing approaches and to compare them, to discuss problems to which satisfactory solutions yet to be found and to show likely directions of future research.

2. Concurrency and System Structuring

Many researchers view all OO systems as inherently concurrent but this is grounded only if object consistency is somehow guaranteed. In reality, concurrency adds a new dimension to system structure and design. These systems are extremely difficult to understand, design, analyse or modify. To do this successfully, we need concurrency features which correspond to the specific characteristics of both OO systems and the applications to be designed.

2.1. Single-Method Concurrency

Concurrency in OO systems is usually provided at the level of separate method calls and objects (either using the conventional procedure call or the remote procedure call mechanisms [TO99]), which allows object consistency to be guaranteed and concurrency aspects of the object behaviour to be addressed (Fig. 1). In this case the units of system structure and behaviour are separate method calls and objects.

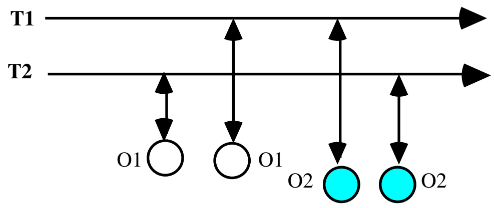

Figure 1. Two threads T1 and T2 access objects O1 and O2 concurrently

2.2. Competitive and Cooperative Systems

Complex object-oriented systems often need sophisticated and elaborate concurrency features which may go beyond the traditional concurrency control associated with separate method calls. The existing single method approaches do not scale because we deal with each single operation separately. There is a need for using units of system structuring which encapsulate complex behaviour and embrace groups of method calls and those of objects. These units should correspond to dynamic system execution as opposed to static declaration of objects inside objects. For example, it makes no sense to declare all potential clients of a server in a bigger object. System understanding, verification and modification is facilitated if system execution is recursively structured of units encapsulating several method calls or/and objects.

Another concern which makes it necessary to extend the single-object view on system structuring is fault-tolerance provision: in many situations one cannot guarantee that erroneous information is always contained inside an object. Without this strong assumption, we have to deal with very complex error containment domains consisting of several interconnected objects. For example, an error in a server can affect several client objects, so it will not be sufficient to recover only one of them. There are many applications which require such structuring units: banking systems, CSCW systems, complex workflows, control of modern production lines and cells, etc.

Various classifications of concurrent systems play an important role in identifying general approaches/techniques as they make it possible to concentrate on characteristics which are specific to different categories of systems and to develop methodologies and supports which make it easier to develop systems of different categories. To better understand the additional considerations we believe should be taken into account in understanding issues of system structuring, let us consider the classification of concurrent systems in [LA90] (which, in its turn, follows classifications in [HR73, H76]). Three categories are outlined here; they are independent (disjoint), competing and cooperating systems.

Competitive concurrency exists when two or more active components are designed separately, are not aware of each other, but use the same passive components. Programmers (would like to) live in an artificial world in which they do not care about other concurrent activities. They access objects as if they had them in their disposal. This concurrency is used, for example, when clients access a server, some of the mechanisms supporting it are the RPC and synchronisation constraints.

Cooperative concurrency exists when several components cooperate, i.e. do some job together and are aware of this. They can communicate by resource sharing or explicitly, but the important thing is that they have been designed together so that they cooperate to achieve their joint goal and use each other's help and results. Existing systems sometimes provide a support for single one-to-one communications, a direct cooperation of equal partners: rendezvous, signals, message send/receive.

Many researchers rely on the concept of *atomicity* in developing structuring approaches to system design. OO systems are easier to understand and to analyse if their execution is built out of atomic units encapsulating several objects and method calls provided no information crosses the border of such units. The ability to nest such units is vital for dealing with system complexity in a scalable way (we say that a unit is *nested* if it contains a subset of objects or/and method calls from the containing one). Providing fault tolerance is essentially facilitated in systems which execution is structured out of such *atomic units* as these units confine erroneous information.

2.3. Structuring Competitive Systems

Atomic transactions, incorporating several object calls, are the main approach to structuring competitive systems (Fig. 2). Atomicity, consistency, isolation and durability (ACID) are the fundamental properties of such units. A transaction can be completed either by committing all updates made or by aborting them. The ACID transactions form the units of system execution and as such can be nested.

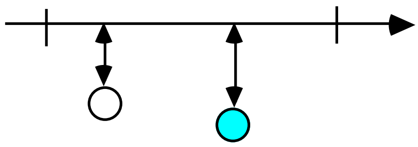

Figure 2. Transaction incorporating several calls of several objects

This approach works well for big database systems but many applications nowadays require more sophisticated features. The original transaction concept has been further developed, in particular, additional concurrency at the caller side is often allowed.

The concept of a *multithreaded transaction* (MTT) has been used in different transactional models for quite a long time. Very typical examples are the CORBA transaction service and Arjuna [PS95]. Several threads can perform operations on a set of transactional objects within a MTT (Fig. 3). One of them starts a transaction, then others learn its identity, using this identity they can access transactional objects within the MTT. If a thread commits or aborts, the transaction commits or aborts. This model is quite general and flexible, it has been used in many industrial applications. Although, from our point of view, it leaves a burden of a very careful and error-prone coordination of threads inside a MTT on application programmers as it does not impose any discipline on what these threads can do (providing the ACID properties of the server objects is the paramount concern here). For example, any thread can decide to leave the MTT without knowing whether it is committed or aborted. In this model threads do not actually join the transaction because the transaction support is not aware of the concurrency, and the transactional objects do not guarantee mutual exclusion for threads of the same transaction. The thread exit from a MTT is not coordinated. Another problem with the MTT model is that programmers have to start and commit/abort transactions explicitly because transactional structure is separate from method/object structure.

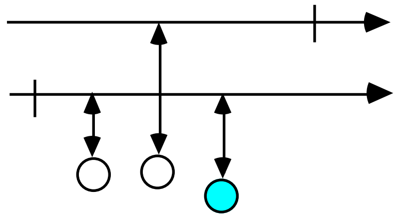

Figure 3. CORBA multithreaded transaction

In the object-based language Argus [L88] applications are composed out of guardians, each of which provides an interface consisting of callable procedures, called handlers. Handlers can fork concurrent threads which are joined when the handler is completed (Fig. 4). Handler execution forms an atomic transaction, nested handler calls form nested transactions. Vinari/ML [HK94] and Transactional Drago [JP00] have similar computational models.

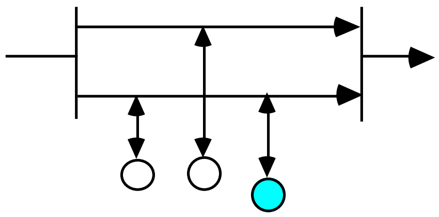

Figure 4. Argus multithreaded transaction

The concurrent OO language Arche [I93] allows dynamic grouping of objects. A caller object can call synchronously methods with the same names and signatures in a group of M server objects; these methods form a *multioperation*. The multioperation results are returned to the caller (Fig. 5). This is a competitive concurrency model (other multioperations compete for these servers) with a simple concurrency control based on mutual exclusion. Servers do not cooperate while executing a multioperation although a multioperation can issue a call to another multioperation which can be only done jointly by all group components; this forms a basis for multioperation nesting. Multioperations are atomic only if the callees do not call external objects.

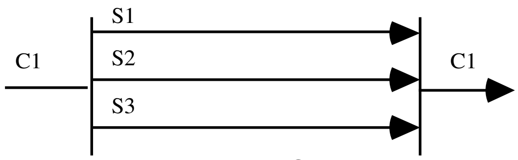

Figure 5. Multioperation in Arche: a caller C calls a group objects S1, S2, S3

2.4. Structuring Cooperative Systems

Many OO systems provide features only for performing single acts of one-to-one cooperation. This is not sufficient for many reasons when complex cooperative applications are to be developed (e.g. complex CSCW systems or workflows). First of all, the approach should scale well to be useful for designing such systems in which several (more than two) objects have to cooperate to achieve joint goals. Secondly, it should rely on structuring units which can be made atomic and nested (to cope with the system complexity). Another concern is providing fault tolerance: we need such atomic units to keep under control the erroneous information which can smuggle between several objects (e.g. several clients of the same server). If we do not structure systems out of such units we cannot know the recovery area. This complex multi-participant cooperation should be a concern of the system design as we do not want to reason about it using single two-participant interactions (which is possible but can dramatically increase the responsibility of programmers).

The general concept of *atomic actions*, proposed in [CR86], answers all these concerns. Several participants enter an action and cooperate inside it to achieve joint goals (Fig. 6). They are designed to cooperate inside the action and are aware of this cooperation. They share work and exchange information in order to complete the action successfully. To guarantee atomicity no information is allowed to cross the action border. Actions can be nested (a subset of the participants of the containing action can join in a nested action). Participants leave the action together when all of them have completed their job. If an error is detected inside an action all participants take part in a cooperative recovery. The *conversation* scheme [R75] was the first atomic action scheme: it uses software diversity and participant rollback to tolerate design faults.

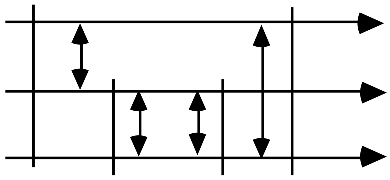

Figure 6. (Nested) atomic actions

There could be several structuring ways for incorporating atomic actions into OO systems. The first approach is to introduce actions as classes with methods representing participants (as, for example, in scheme in [WB97]). The computation model allows all participants to be active at the same time. Although, in this case we are loosing the ability to treat participants as classes. Another approach is to view actions as sets of participant objects. For example, in the scheme [R95] a set of objects takes part in an atomic action by executing one method each, the action here is formed as a set of separate methods. Interfaces of participant objects have to be extended to allow their synchronisation. In both scenarios we need a special support which coordinates participant execution. These ideas allow us to make use of many benefits of OO programming while designing new OO atomic action schemes (including their support) and while applying them.

3. Exception Handling

Exceptions are abnormal events which can happen during the program execution. Many languages and systems provide special features for handling them in a disciplined way. These features allow programmers to *declare* exceptions and provide them with the ability to treat a program block as the *exception context* and to associate *handlers* with such context, so that when an exception is raised in this context the execution stops and the corresponding handler is searched among the handlers (there are some models in which one can propagate an exception straight outside the context). In our opinion, the vital feature of any exception handling mechanism is its ability to differentiate between *internal exceptions* to be handled inside the context and the *external exceptions* which are propagated outside the context: these exceptions are not clearly separated in many languages although it is obvious that they are intended for different purposes. This separation can be done under two conditions: contexts are associated with program units which have interfaces and the concept of the *context nesting* is defined. The majority of the existing exception handling mechanisms use dynamic exception context nesting in which case the execution of the context can be completed either successfully or by interface exception propagation, in which case this exception is treated as an internal exception raised in the *containing context*. The simplest example of the dynamic nested context is nested procedure calls. Actually this is the dominating approach which suits well the client/server or remote procedure call paradigms and which is used in the majority of the systems.

The model of exception handing is basically the same in OO programming: the exception contexts are methods or classes, interface exceptions are declared in the type (often in the method signatures). Exception handling is, in essence, sequential here as it is related to single classes or separate methods.

4. Single-Method Exception Handling in Concurrent OO Systems

In many concurrent OO systems (e.g., Java and Ada) exceptions are propagated through nested (and, sometimes, remote) method calls and the exception contexts are either separate methods or objects. Effectively this means that these systems separate exception handling from concurrent programming. Some researchers argue that special features for involving several concurrent objects into exception handling are so difficult to develop and to use that OO system developers should use only sequential exception handling. One of the problems from their point of view is that it should be possible to randomly interrupt threads when an exception is raised in another thread. We believe that this misunderstanding is caused by the fact that issues of exception handling are considered separately form issues of system structuring, which is clearly a mistake because of many reasons: first, exception contexts are units of system structuring; secondly, dynamic system structure is described by context nesting and, thirdly, interface exceptions have to be a part of the structuring units.

There are no reasons why exception handling has to be sequential in concurrent systems. Concurrent exception handling should be associated with the way the concurrent system is structured in the same manner in which this works for sequential systems. We consider such support for exception handling in concurrent programming to be vital for dealing with the complexity of concurrent systems.

There have been some attempts to address this problem. For example, the Oz language allows associating a handler with a thread; this can be used for handling any exceptions raised in any threads and for handling exceptions propagated out of the outmost context in the thread: this handler is initiated before the thread is terminated. Language Facile (an extension of SML) allows us to declare the same exception in several processes, when this exception is raised in any of them the execution of all processes which declared this exception is interrupted and handlers are called (the process terminates if it has not got a handler for this exception). An exception propagated out of the accept body in Ada rendezvous is signalled in the context of the caller and of the callee containing the `accept` body.

Another interesting example is an extension of the concurrent OO language ABCL/1 by a concurrent exception handling mechanism [IY91]. This extension relies on the ABCL/1 computational model within which method calls are viewed as message transmissions between concurrent objects, and methods as operations initialised by accepting the corresponding messages. Exceptions are treated here as signals that can be transmitted between objects. Any method call can be accompanied by a special tag indicating the reply destination: the tag is the name of the object which will receive the method results (the reply). The exception context is a block of statements or a method body. In the extended ABCL/1 a new notion of *complaint* is introduced. It is similar to the notion of reply but intended for informing another object (complaint destination) of any unexpected things occurring during the object (method) execution. Complaints (actually, a type of failure exceptions) can be of four kinds: unaccepted messages, time-outs, system-defined (predefined) and user-defined complaints. Complaint destination can be declared in each object, which changes the direction of exception propagation from methods (objects).

The schemes above neither relate exception handling with structuring concurrent system nor scale well, leaving all responsibilities for transferring information about exceptions among several processes and their coordinated handling with programmers.

5. Action-Oriented Exception Handling in Complex Concurrent Systems

Complex concurrent systems that are structured using atomic units offer us a straight-forward choice of exception contexts Treating such units as the contexts seems to be the most beneficial way. It is important that this approach is compatible with the way we structure sequential system for exception handling, based on nested method calls. The general exception handling model allows internal exceptions and corresponding handlers associated with the structuring unit. These units have interfaces enriched by the external exceptions which the unit can propagate into the containing exception context (that is into the containing structuring unit). Atomicity of such units (that it of exception contexts) is vital for dealing with abnormal events (i.e. exceptions) as it guarantees the containment of all information which is needed for exception handling and recovery.

There is an important question which should be addressed while developing support for such atomic units. Many evidence indicate that multiple exceptions can be raised at the same time in a concurrent system. These situations have to be correctly resolved. Paper [CR86] proposes a concept of the *exception tree* which includes all exceptions associated with an atomic unit and imposes a partial order on them in such a way that the higher exception has the handler which is intended to handle any lower-level exception. The idea is to handle the *resolved* exception corresponding to the tree node which is higher than all concurrent exceptions raised.

6. Exception Handling in Competitive Systems

The designers of transactional systems often do not incorporate exception handling and use return error codes instead. There are many problems with this approach. Firstly, using return codes have been always described as a canonical example of bad practice caused by the absence of the exception handling mechanism. Secondly, even if the code language has an exception handling it is completely separated from transactions and, as a result, the application exception handling (including the exception context, exception propagation, etc.) is separated from the transactional structure. The CORBA transaction service (Fig. 3) is an example of this: it offers a very sophisticated MTT model but only sequential exceptions (that of C++ or Java): any exception raised in a MTT can cross the MTT border unnoticed, each MTT participant deals with its exceptions separately, the MTT is not the exception context. Actually, the transaction border is not clearly defined here as participant threads are not coordinated in any way.

The most general approach to incorporating exception handling into competitive systems is to allow each transaction to have internal exceptions with handlers inside and external exceptions described in the transaction interface. Generally speaking, interface exceptions are to be propagated to the containing transactions. It is important to be able to associate some external exceptions with the abort outcome; when the other exceptions are signalled, the state of all objects involved should be known and committed. The problem here is to introduce transactional exception handling into the OO context and to avoid having different exception mechanisms for sequential and concurrent programming within the same system.

Argus (Fig. 4) provides a very powerful extension of sequential OO exception handling. Methods (handlers) are atomic transactions which have external exceptions declared in their interface. Threads can be forked inside, allowing a very rich computation to be performed concurrently. As opposed to CORBA MTT, all Argus threads have to be synchronised and joined when the transaction commits or aborts. An interface exception is propagated to a single-threaded caller when any thread inside the transaction signals it. Any thread may decide to signal an exception with or without transaction abort. Internal thread exceptions have to be dealt with separately by different threads as the system does not provide any coordination in dealing with such exceptions. Argus offers a special construct for handling interface exceptions rather than making it possible for the containing transaction to deal with them explicitly at its level.

Vinari/ML offers a transactional extension of SML which allows creation of transactional versions of high-order functions. Its computational model is similar to that of Argus but it is not possible to declare external exceptions in transactional functions.

Transactional Drago is an extension of Ada and as such requires a pre-compiler and a special run-time support. It is based on forking/joining threads inside transactions and, as opposed to Argus and Vinari/ML, allows concurrent exception resolution before signalling. Any exception propagated outside a transaction causes the transaction abort.

In Arche (Fig. 5) each multifunction member represents an isolated exception context and as such can signal an external exception. When all members have completed their execution, a resolution function is applied, and the resolved exception is propagated to all callers' contexts (unless appropriate action is not taken by programmers, this approach can leave members in inconsistent states).

7. Exception Handling in Cooperative Systems

Exception handling in cooperative systems can be quite naturally incorporated into the atomic action framework [CR86]. An OO scheme which is based on this research is described in [XR98]. A set of internal and external exceptions is associated here with each action. Each object participating in the action has a set of handlers for (some or all of) internal exceptions. When an exception is raised in any participant, the appropriate handlers are initiated in all of them, and they will recover the action cooperatively. In this approach action participants cooperate not only when they execute program functions (i.e. during normal activity) but also when they handle abnormal events. This is mainly due to the fact that ,when an atomic action is executed, an error can spread to all participants, and the system can be returned into a consistent state only if all participants are involved in handling. An action can be completed either by signalling an interface exception into the context of the containing action or normally. In scheme [XR98] concurrent internal exceptions are resolved using a resolution graph.

Although several OO schemes incorporating this kind of exception handling have been proposed, there are still many theoretical and practical problems to be addressed. It is not clear, for example, how to make an ordinary object also capable of performing, when required, the functions of an action participant: the computational models and object interfaces are very different for these two entities. There are unclear points in how to properly combine sequential exception handling and atomic action exception handling to allow compatibility. The problems of inheriting and refining action and participant classes or types have not been addressed (let alone the refinement of action exceptions, action handlers, etc.).

References:

[CR86] R. H. Campbell, B. Randell. Error Recovery in Asynchronous Systems. IEEE TSE-12, 8.

[H76] C.A.R. Hoare. Parallel Programming: an Axiomatic Approach, LNCS-46.

[HK94] N. Haines, D. Kindred, J.G. Morrisett, A.M. Nettles, J.M. Wing. Composing First-Class Transactions. ACM TOPLAS, 16, 6.

[HR73] J.J. Horning, B. Randell. Process Structuring. Comp. Surveys, 5.

[I93] V. Issarny. An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software. JOOP, 6, 6.

[IY91] Y. Ichisugi, A. Yonezawa. Exception Handling and Real Time Features in Object-Oriented Concurrent Language. LNCS-491.
[JP00] R. Jimenez-Peris, M. Patino-Martinez, S. Arevalo. TransLib: An Ada 95 Object Oriented Framework for Building Transactional Applications. Computer Systems: Science & Engineering J., 15, 1.
[L88] B. Liskov. Distributed Programming in Argus. CACM, 31, 3.
[LA90] P.A. Lee, T. Anderson. Fault Tolerance: Principles and Practice.
[PS95] G.D. Parrington, S.K. Shrivastava, S.M. Wheater, M.C. Little. The Design and Implementation of Arjuna. USENIX Computing Systems J., 8, 3.
[R75] B. Randell. System Structure for Software Fault Tolerance. IEEE TSE-1, 6.
[R95] A. Romanovsky. Conversations of Objects. Computer Languages, 21, 3/4.
[TO99] A. Tripathi, J. Van Oosten. R. Miller. Object-Oriented Concurrent Programming Languages and Systems. JOOP, 12, 6.
[WB97] A.J. Wellings, A. Burns. Implementing Atomic Actions in Ada 95. IEEE TSE-23, 2.
[XR98] J. Xu, A. Romanovsky, B. Randell. Exception Handling and Resolution in Distributed Object-Oriented Systems. ICDCS-18.