Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems

Alexander Romanovsky[1] and Jörg Kienzle[2]

[1]Department of Computing Science, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK
alexander.romanovsky@ncl.ac.uk

[2]Software Engineering Laboratory, Swiss Federal Institute of Technology
CH - 1015 Lausanne Ecublens, Switzerland
joerg.kienzle@epfl.ch

Abstract. The chief aim of this survey is to discuss exception handling models which have been developed for concurrent object systems. In conducting this discussion we rely on the following fundamental principles: exception handling should be associated with structuring techniques; concurrent systems require exception handling which is different from that used in sequential systems; concurrent systems are best structured out of (nested) actions; atomicity of actions is crucial for developing complex systems. In this survey we adhere to the well-known classification of concurrent systems, developed in the 70s by C.A.R. Hoare, J.J. Horning and B. Randell, into cooperative, competitive and disjoint ones. Competitive systems are structured using atomic transactions. Atomic actions are used for structuring cooperative systems. Complex systems in which components can compete and cooperate are structured using Coordinated Atomic actions. The focus of the survey is on outlining models and schemes which combine these action-based structuring approaches with exception handling. In conclusion we emphasise that exception handling models should be adequate to the system development paradigm and structuring approaches used.

1 Introduction

System structuring is employed to successfully deal with the growing complexity of modern computer systems. The need to cope with abnormal system behavior makes system design more complicated and, as experience shows, more error-prone. Exception handling was therefore introduced as a disciplined and structured way of handling abnormal system events [7]. It is usually a very important part of any general structuring technique used in system design as it adds new ways of concern separation which are vital for dealing with abnormal situations: it allows us to separate normal code from exception handlers during system design and structuring, introduces a dynamic separation of the execution of normal code and handlers, and provides two ways of returning the control flow after the execution of a system component. This clearly shows that exception handling mechanisms should rely on the way the system is structured and be an integral part of system design. Many researchers regard exception handling as a means for achieving system fault tolerance [5, 18], and we share this view. In this context exception raising follows error detection, exception handling equals to error recovery and units of system structuring are units of exception handling and of recovery. Exception handling is used for incorporating application-specific fault tolerance.

Considerable effort has been devoted to developing exception handling models for sequential object-oriented systems, so a common understanding exists on many topics in the field. Many practical systems have been designed using these features. The situation is different in concurrent object-oriented systems. Although several schemes combining concurrency and exception handling have been proposed, research in this area is still scattered and most concurrent systems use sequential exception handling. It is our belief that this is not the way it should be as 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 a system design issue, and language features should assist in and impose proper design. Exception handling is tightly 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 object-oriented 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 in with the main principles of object-oriented 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 have yet to be found and to show likely directions of future research.

2 Concurrency and System Structuring

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

2.1 Single Method Concurrency

Concurrency in object-oriented systems is usually provided at the level of separate method calls and objects (e.g. in integrated languages [30], which unify processes and objects by defining objects as active entities). This allows object consistency to be guaranteed and concurrency aspects of object behaviour to be addressed (see Fig. 1). In this case the units of system structure and behaviour are separate method calls and objects.

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 objects and of method calls. These units should represent dynamic system execution as opposed to the static declaration of objects inside objects. For example, it clearly 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.

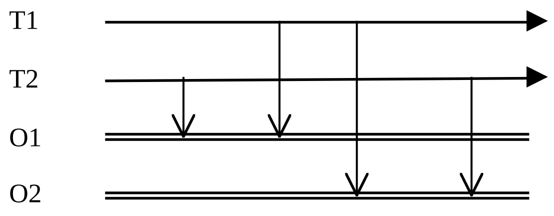

Fig. 1. Two threads T1 and T2 access objects O1 and O2 concurrently

Another concern which makes it necessary to extend the single-object view of system structuring is provision of fault-tolerance: 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 (a client or the server). 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 additional considerations that we believe should be taken into account in addressing issues of system structuring, let us consider the classification of concurrent systems in [18] (which, in its turn, follows classifications in [10, 11]). 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 have to care about other concurrent activities. They access objects as if they had them at 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 are designed together so that they can cooperate to achieve their joint goal and use each other's help and results. Existing systems sometimes provide 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. Concurrent object-oriented systems (and systems in general) are easier to understand and to analyse (see, for example [2, 17]) if their execution is built out of atomic units encapsulating several objects and method calls, provided no information crosses the borders 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 whose execution is structured out of such *atomic units* as these units confine erroneous information (see [28] for a detailed discussion).

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 [8]. A transaction can end either by committing all updates made on the objects or by aborting them. The ACID transactions form the dynamic units of system execution and as such can be nested in many models and implementations. These transactions are oriented mainly towards tolerating hardware faults of different types: transient faults, node crashes, etc.

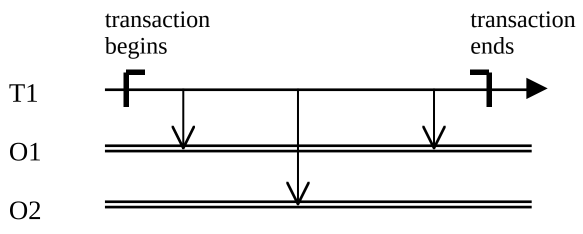

Fig. 2. Transaction incorporating several calls of several objects (O1 and O2)

This approach works well for database, client-server or simple bank 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 [20] and Arjuna [21]. Several threads can perform operations on a set of transactional objects within an MTT (Fig. 3). One of them starts a transaction, then others learn its identity, using which they can access transactional objects within the MTT. If a thread commits or aborts, the transaction does the same. This model is quite general and flexible, it has been used in many industrial applications. However, it leaves the burden of a highly labour-consuming and error-prone coordination of threads inside an MTT to application programmers as it does not impose any discipline on what these threads can do (guaranteeing the ACID properties of server objects is of 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 transactional objects do not guarantee mutual exclusion for threads of the same transaction. The thread exit from an 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.

Generally speaking, a very similar transactional model is provided by Enterprise JavaBeans architecture (EJB) [6]. EJB allows system developers to associate several client threads with the same transactional context. Unfortunately, this architecture supports only flat transactions (nesting is not allowed).

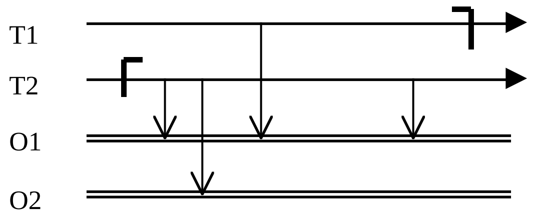

Fig. 3. CORBA multithreaded transactions

Applications built using the object-based language Argus [19] are composed of guardians, each of which provides an interface consisting of callable procedures called handlers. Handlers can fork concurrent threads which are joined when a handler is completed (see Fig. 4). Handler execution forms an atomic transaction; the execution of nested handler calls are performed as nested transactions.

The Argus approach has been very influential: Vinari/ML [9] and Transactional Drago [14] have similar computational models. Vinari/ML offers a transactional extension of SML which allows creating transactional versions of high-order functions; in this model new participants are explicitly forked by existing participants. Transactional Drago is an extension of Ada (it requires a pre-compiler and a special run-time support) which allows any program block to be declared and executed as an ACID transaction. Tasks declared inside this block are executed together with the block as additional transaction participants and they are to be completed before the transaction can end.

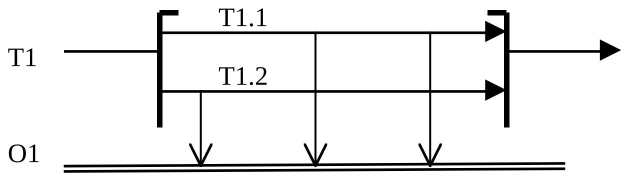

Fig. 4. Argus multithreaded transactions

A new model, called open multithreaded transactions (OMTT), has been recently proposed to allow developing systems with a richer concurrency than that of Argus yet keeping the transaction boundary on the caller (thread) side under control [15]. In the OMTT model multiple threads, called joined participants, can join a transaction, and any transaction participant can fork a thread which becomes a new transaction participant called a spawned participant (Fig. 5). The restriction is that if a participant has been created inside a transaction it has to be completed inside it. Note that such participants take part in the execution of the final commit/abort protocol. The OMTT can be nested: only participants of the containing transaction can join the nested one. Transactional support effectively consists of two parts: one guarantees the ACID properties of the objects called by transaction participants, the other coordinates transaction participants (transaction entry, exit, nesting). Transaction participants can see each other's updates of transactional objects but the entire transaction is isolated from the rest of the system. In some ways this scheme allows participants to (loosely) cooperate but the idea is that they do not depend on each other and have their own goals inside such a transaction.

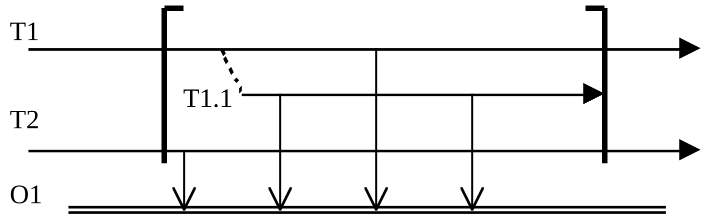

Fig. 5. Open multithreaded transactions

The concurrent object-oriented language Arche [13] allows dynamic grouping of objects. A group of N caller objects can synchronously call methods with the same names and signatures in a group of M server objects (e.g. objects of the same type); all these methods form a *multioperation*. Multioperation results are returned to all callers (Fig. 6). Some servers can synchronously call another multioperation. Arche relies on a competitive concurrency model (other multioperations compete for server objects) with a simple concurrency control based on mutual exclusion. Cooperation of servers executing a multioperation is not supported in the model, although a multioperation can issue a call to another multioperation which can only be performed jointly by all group components; this forms a basis for multioperation nesting. Arche does not use the full-fledged model of atomic transactions: multioperations are atomic only if the callees do not call external objects. This computation model has proved useful for implementing object replication and for employing diversely designed objects.

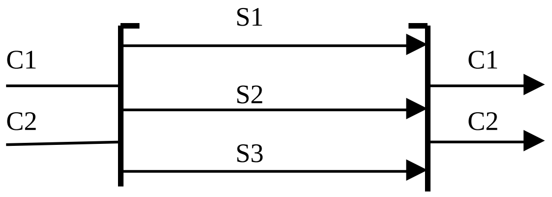

Fig. 6. Multioperation in Arche: callers C1 and C2 call a group objects S1, S2, and S3

2.4 Structuring Cooperative Systems

Many object-oriented systems provide features only for performing single acts of one-to-one cooperation. For a number of reasons, this is not sufficient when complex cooperative applications, such as complex CSCW systems or workflows, are to be developed. First of all, the approach should scale well to be useful for designing such systems in which 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 system complexity). Another concern is providing fault tolerance: we need such atomic units to keep under control erroneous information which can be smuggled between several objects (e.g. several clients of the same server). If we do not structure systems out of such units we encounter serious problems in defining the recovery region. This complex multi-participant cooperation should be a system design concern as we do not want to reason about it using single two-participant interactions (which can be done but can dramatically increase the responsibility of programmers and as such be error-prone).

The general concept of *atomic actions*, proposed in [4], answers all these concerns. Several participants (threads, processes, objects, etc.) enter an action and cooperate inside it to achieve joint goals (Fig. 7). They are designed to cooperate inside the action and are aware of this cooperation. These participants share work and explicitly exchange information in order to complete the action successfully. Atomic actions structure dynamic system behaviour. To guarantee action 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 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. Atomic actions provide a sound framework for developing schemes intended for tolerating faults of different types: hardware faults, software design faults, transient faults, environmental faults, etc. The *conversation* scheme [23] was the first atomic action scheme proposed: it uses software diversity and participant rollback to tolerate design faults. A number of atomic action schemes incorporating different fault tolerance techniques have been developed since then for different languages: CSP, Concurrent Pascal, Ada, OCCAM, Java (with and without extensions); for distributed, multiprocessor and single computer settings; for different application requirements [24].

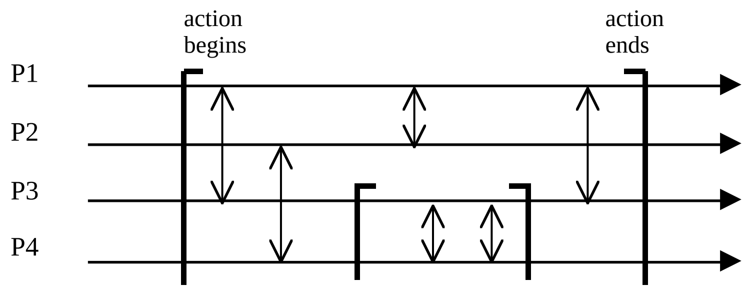

Fig. 7. Atomic actions: participants P1-P4 take part in the containing action, participants P3 and P4 in the nested action

There could be several structuring ways of incorporating atomic actions into object-oriented and object-based systems. The first approach is to introduce actions as classes or objects with methods representing participants, one each (as, for example, in the schemes [16, 32]). The computation model allows all participants to be active at the same time. The downside is that in this case we are losing the ability to treat participants as classes. Another approach is to view actions as sets of participant objects. For example, in scheme [26] 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 on the action entry, exit and nesting. In both scenarios we need a special support to coordinate participant execution. These ideas allow us to make use of the many advantages of object-oriented programming while designing new object-oriented atomic action schemes (including their supports) and applying them.

2.5 Structuring Systems with Cooperative and Competitive Concurrency

Developers of the Coordinated Atomic action (CA action) concept [25, 33] realised that many realistic systems to be modelled/controlled by software have elements of both cooperation and competition and that it is important to allow them to be combined within one system. CA actions provide a framework for dealing with different kinds of concurrency and achieving fault tolerance by integrating and extending two complementary concepts - atomic actions [4] and atomic transactions [8]. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency (Fig. 8).

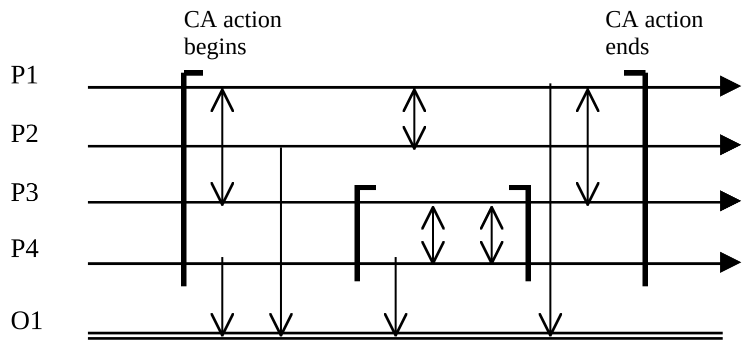

Fig. 8. CA atomic actions: action participants access transactional objects

Each CA action is designed as a stylised multi-entry procedure with roles which are activated by action participants cooperating within the CA action. Logically, the action starts when all roles have been activated and finishes when all of them reach the action end. CA actions can be nested. The state of the CA action is represented by a set of local and transactional objects. Transactional objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the start and completion of the CA action has the ACID properties with respect to other sequences. The execution of a CA action looks like an atomic transaction for the outside world. Action participants explicitly cooperate (interact and coordinate their executions) through local objects (for example, those of message, mailbox or buffer classes). All participants are involved in recovery if an error is detected inside an action since conceptually it makes no difference which of them detects an error and the whole action represents the recovery region. Object-orientation plays an important role in the

CA action concept and in developing different Java and Ada implementation schemes: concrete actions, action roles, local object and transactional objects are viewed as instances of classes.

3 Exception Handling

Exceptions are abnormal events which can happen during program execution. Many languages and systems provide special features for handling them in a disciplined way. These features allow programmers to *declare* exceptions and enable programmers to treat a program unit as the *exception context* and to associate exceptions and *exception handlers* with such context, so that when an exception is raised in this context, execution stops and a corresponding handler is searched for 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 provided the following conditions are met: contexts are associated with program units which have interfaces and the concept of *context nesting* is defined. Most 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 - 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 the client/server or remote procedure call paradigms well and which is used in most systems and languages (e.g. in C++, Ada, Java, CLU).

External exceptions allow programmers to pass (in a disciplined, unified and structured fashion) different outcomes to the containing context. This can be used to inform it of the reasons for abnormal behaviour and of the state in which the context has been left, to pass partial results, etc. Another important issue which exception handling models have to address is defining the state in which the context is left when an external exception is propagated. Some systems provide an automatic support which guarantees the "all-or-nothing" semantics: if an exception is propagated outside, all modifications made inside the context are cancelled. Another possibility (which originates in the Inscape software development environment [22]) is to allow the context to be left in several states: an initial state (an abort exception is propagated); successfully committed state (if no exception is propagated outside); and several "partial" committed states, when the requested result cannot be achieved but partial (or degraded, alternative) results are still acceptable (external exceptions are propagated). It is clear that developing supports to provide such functionalities is a difficult task, this is why in many systems all responsibility of leaving the context in a known and consistent state rests entirely with application programmers.

The model of exception handing in object-oriented programming follows all fundamental principles of building such mechanisms. Exception handling is usually associated with either dynamic (method calls) or static (object/class declaration) system structuring: exception contexts are methods or classes, interface exceptions are declared in the type (often in method signatures). Unfortunately, in many concurrent object-oriented systems exception handling is, in essence, sequential as it is related to single classes or separate methods.

4 Single Method Exception Handling in Concurrent Systems

In many concurrent object-oriented systems (e.g., Java, Guide, Arche and Ada) exceptions are propagated through nested (and, sometimes, remote) method calls and exception contexts are either separate methods or objects. These systems provide features for guaranteeing object consistency when several clients issue concurrent calls. This is a very important issue but in our opinion this type of exception handling is not sufficient for many reasons. If only mechanisms of this type are employed exception handling is effectively separated from concurrent programming. Moreover, such mechanisms rely on a very simplistic view of concurrent system structuring and of handling abnormal events in such systems. Some researchers (e.g. [3]) argue that special features for involving several concurrent objects in exception handling are so difficult to develop and use that object-oriented system developers should use only sequential exception handling. Thus, an essential but a most difficult feature to provide is random interruption of a thread when an exception is raised in another thread. We believe that this misunderstanding is due to the fact that exception handling issues are being considered separately from those of system structuring, which is clearly wrong for many reasons: first, exception contexts are (should be viewed as) units of system structuring; secondly, dynamic system structure is defined by exception context nesting and, thirdly, interface exceptions have to be part of structuring units.

In our opinion, there are no reasons why exception handling should have to be sequential in concurrent systems. Concurrency clearly adds a new dimension to system design and execution. And exception handling should keep up with this new feature. Moreover, concurrent exception handling should be associated with the way a 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 vital for dealing with the complexity of concurrent systems. Ideally, exception contexts (i.e. structuring units) should encapsulate complex behaviour consisting of several operations on several objects.

There have been some attempts to address this problem. For example, the Oz language [31] allows associating a handler with a thread. This handler is initiated before the thread is terminated, which can be used for handling any exceptions raised in any threads as well as for those propagated out of the outmost context in the thread. Language Facile (an extension of SML) [29] 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 a handler for this exception). Another example is Ada, in which an exception propagated out of the accept body during rendezvous is signalled in the context of the caller and of the callee containing the accept body.

A more sophisticated example is an extension of the concurrent object-oriented language ABCL/1 by a concurrent exception handling mechanism [12]. 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 object (method) execution. Complaints (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).

Language $\mathcal{E}_{CSP}$ is another interesting attempt to introduce exception handling into concurrent systems [1]. In this language, if a process cannot continue its normal execution because of an exception, it signals a global exception so that any process which will be communicating with this process in the course of its normal execution will get an exception raised in its context.

Unfortunately, the schemes above neither relate exception handling to structuring concurrent systems, nor scale well. They do not provide any support for leaving the exception context in a known consistent state. Usually all responsibility for transferring information about exceptions among several processes and their coordinated handling is left with programmers.

5 Action-Oriented Exception Handling

Structuring complex concurrent systems using atomic actions offers us a straightforward choice of exception contexts. (By atomic actions we mean all types of atomic units of structuring system behaviours discussed above in Section 2: atomic transactions, atomic actions, CA actions.) Treating such units as contexts seems the most beneficial way because these atomic units have clearly defined borders, can be nested and no information can cross the unit border. It is important that this approach is compatible with the way we structure sequential systems for exception handling, which is based on nested method calls. The general exception handling model can easily be applied here to allow internal exceptions and corresponding handlers to be associated with such structuring unit. Actions can have interfaces enriched by external exceptions which the unit can propagate into the containing exception context (i.e. into the containing structuring unit). Atomicity of actions (i.e. of exception contexts) is vital for dealing with abnormal events (i.e. exceptions) as it guarantees the containment of all (potentially erroneous) information which should be involved in exception handling and recovery. Clearly, the atomicity of action execution has a general importance for all phases of system development: it facilitates reasoning about the system, system understanding, verification and development, tolerating faults of different types, etc. In addition, it guarantees the most beneficial way of information and behaviour encapsulation, when no intermediate results can be seen from the outside and the execution of units is indivisible. This is why we believe that exception handling in concurrent systems should be action-oriented.

There is an important question which should be addressed while developing support for such atomic units. There is a lot of evidence indicating that it is very likely that multiple exceptions are raised at the same time in a concurrent (and, in particular, distributed) system [27, 35]. These complex situations have to be correctly resolved, and atomic actions give a simple and well-structured way of dealing with them. First of all, concurrent exceptions raised in concurrent (sibling) actions are handled separately. To deal with exceptions raised inside an atomic action, paper [4] proposes the concept of *exception tree* which includes all exceptions associated with this action and imposes a partial order on them in such a way that a higher exception in the tree has a handler which is capable of handling any lower-level exception or any combination of them. The idea is to handle the *resolved* exception which corresponds to the tree node that is higher than nodes of all concurrent exceptions raised in the action. Recently this approach has been further developed to allow action exceptions to be ordered by a resolution graph and to provide an improved decentralised resolution algorithm [35].

Generally speaking, atomicity of actions means that the intermediate results of action execution are not seen from the outside; we will adhere to this understanding in the following discussion of different action schemes. Some of these schemes allow partial (but consistent) action results to be achieved and the system to be moved in a new consistent state when an exception is propagated outside this action; others subscribe to the idea that if any exception is signalled outside an action, the "nothing" semantics should be provided.

5.1 Exception Handling in Competitive Systems

The designers of transactional systems often do not incorporate exception handling but use return error codes instead. There are many problems with this approach. Firstly, the use of return codes has always been described as a canonical example of bad practice caused by the absence of the exception handling mechanism [7]. Secondly, even if the core language has exception handling, it is completely separated from transactions and, as a result, application exception handling (including the exception context, exception propagation, etc.) is separated from the transactional structure. The CORBA transaction service [20] (Fig. 3) is a typical example of this: it offers a very sophisticated MTT model but programmers can use only sequential exceptions (e.g. those of C++ or Java): any exception raised in an MTT transaction can cross its border unnoticed, each MTT participant deals with its exceptions separately, the MTT transaction is not the exception context, one cannot define or handle exceptions at the transaction level. Actually, the transaction border is not clearly defined in this model as participant threads are not coordinated in any way.

It is symptomatic that the designers of EJB [6] have made a serious efforts to combine exception handling with transactions. This model allows us to develop a system in which any exception signalled by a transactional object can affect the execution of the whole transaction. For example, one can mark the transaction for abort, re-raise the same or another exception, try to recover the situation and continue the transaction, abort the transaction and re-raise the same exception, etc. However, it is clear that MTTs are not full-fledged exception contexts because multiple participants are not coordinated (e.g. they are not informed when the transaction is aborted) and because such transactions cannot be nested.

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 transaction. It is important to be able to associate some external exceptions with the abort outcome; when 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 object-oriented context and to avoid having different exception mechanisms for sequential and concurrent programming (i.e. for individual threads and for transactions) within the same system.

Argus [19] (Fig. 4) provides a very powerful extension of sequential object-oriented exception handling. Methods (called handlers in this model) are atomic transactions which have external exceptions declared in their interfaces. Threads can be forked inside, allowing very rich computations to be performed concurrently. Unlike the 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, which makes it possible to commit partial results and to associate different results with different exceptional outcomes. Internal thread exceptions have to be dealt with separately by individual threads as the system does not provide any coordination for dealing with such exceptions (which suits the competitive nature of this model well). 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.

The exception handling model of Vinari/ML [9] is in many ways similar to that of Argus but it does not differentiate between external and internal exceptions: it is not possible to declare external exceptions in transactional functions; a transaction is always aborted if any exception is propagated outside the transactional function; if there is no local thread-level handler for an exception, it gets propagated outside the transaction.

Transactional Drago [14], unlike Argus and Vinari/ML, resolves concurrent exceptions raised by several participating threads before signalling a resolved exception outside the transaction. In this model, external exceptions cannot be declared in the transaction interface, and any exception which is not handled by a thread locally aborts the transaction and gets propagated outside it.

The OMTT model clearly separates internal and external exceptions. Each participant has to have handlers for all of its local exceptions. If it cannot handle it, it has to explicitly signal an external exception which always causes the transaction abort. External exceptions propagated by a joined participant are raised in the containing context of the caller thread. There is a predefined exception Abort_Transaction which can be signalled by spawned participants if they decide to abort the transaction. This exception is propagated to the callers of all joined participants if they do not signal their external exceptions concurrently.

In Arche (Fig. 6) 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 caller contexts (unless an appropriate action is taken by programmers). This approach can clearly leave member objects in inconsistent states.

5.2 Exception Handling in Cooperative Systems

Exception handling in cooperative systems can be quite naturally incorporated into the atomic action framework [4, 35]. A set of internal and external exceptions is associated here with each action, and these exceptions are clearly separated. The model is recursive, and all external exceptions of an action are viewed as internal ones of the containing action (Fig. 7). Each object participating in the action has a set of handlers for all internal exceptions. 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. This is why, when an exception is raised in any participant, appropriate handlers are initiated in all of them. An action can be completed either by signalling an interface exception into the context of the containing action or normally (without internal exceptions being raised or after a successful cooperative handling of such exceptions). Concurrent internal exceptions are resolved using a resolution graph, so that handlers for the resolved exception are called in all participants (see Fig. 9).

Even though several object-oriented schemes incorporating this kind of exception handling have already been proposed (several of them will be mentioned in Section 5.3 as this research has been mainly conducted in the context of developing the CA action concept), there are still some 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 still unclear points as to how properly combine sequential exception handling and atomic action exception handling in order to allow compatibility. The problems of inheriting and refining action and role classes or types have not yet been addressed (let alone the refinement of action exceptions, exception handlers, etc.).

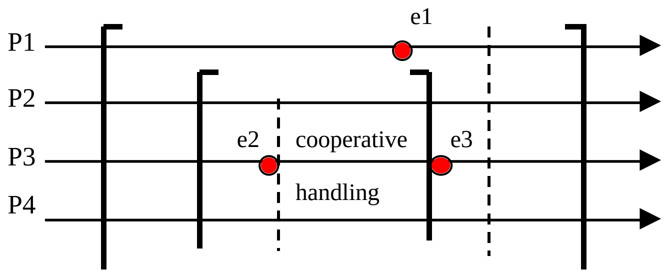

Fig. 9. Exception handling in cooperative systems: internal exception e2 is in the nested action (in participant P3); after an attempt to handle it cooperatively (by P3 and P4) interface exception e3 is propagated to the containing context. Another exception, e1, is raised concurrently in this context, and these two exceptions have to be resolved before cooperative handling starts at the level of this action

5.3 Exception Handling in Systems with Cooperative and Competitive Concurrency

Conceptually, exception handling in CA actions [25] (Fig. 8) is very similar to that in atomic actions [4]: all action participants are involved in cooperative handling of any internal exception, internal exceptions raised concurrently are resolved, external exceptions are explicitly propagated by action participants, etc. (Fig. 9). The main extension is an explicit dealing with local and transactional objects [35].

The CA action interface can contain one or more abort exceptions, a predefined failure exception and a number of exceptions corresponding to partial (committed and consistent) results which the action can provide. In the latter case it uses external exceptions to inform the containing action of the fact that it has not been able to produce a complete required result and, indirectly, of the state in which objects have been left and of the partial results produced. When an abort interface exception is signalled, the CA action is aborted: all local objects are destroyed (although, to improve performance, they can be simply re-initialised if software diversity or retry are used for recovery) and all modifications of transactional objects are cancelled. A failure interface exception is signalled by the support when some serious problems are encountered; for example, the support cannot abort or commit the states of transactional objects. When an interface exception corresponding to a partial result is signalled outside an action, the state of all transactional objects is committed before raising this exception in the containing context. In all these cases signalling an interface exception means that the responsibility for dealing with such abnormal event is passed to a higher level in the system structure. The identity of the exception (with, possibly, some output parameters) and the associated post-conditions provide this level with all information it might need about the reasons for the exception and the current system state.

There has been considerable experimental research on developing object-oriented CA action schemes in Java and Ada and on applying CA actions to developing realistic case studies: a series of Production Cell case studies, including a fault tolerant one [34] and a real time one; a distributed internet Gamma computation; an auction system and a subsystem of a railway control system which deals with train control and coordination in the vicinity of a station. This research has produced first ever field results on applying exception resolution: elaborate resolution graphs have been built for a system controlling a complex industrial application with high reliability and safety requirements [34].

6 Conclusions

The purpose of this survey is to analyse the exception handling models used in concurrent (mainly object-oriented) languages and systems. Development of exception handling features is tremendously complicated by the fact that exception handling is a crosscutting issue which affects all other techniques and mechanisms used in system development as any of them can encounter abnormalities of different types and has to deal with them properly. Poorly developed models can undermine the basic purpose of exception handling, which is concerned with dealing with abnormalities in a disciplined and uniform way throughout the whole system (design and execution). In this survey we wanted to demonstrate the main trends in developing exception handling models for complex concurrent systems and to compare the existing models using some fundamental ideas we believe in as the criteria:

- exception handling should reflect the way systems and their execution are structured
- exception handling is the most general mechanism for achieving system fault tolerance
- concurrent systems should be structured in a way which is different from that of structuring sequential systems
- (nested) actions containing the execution of several objects should serve as (nested) exception contexts
- atomicity of structuring unit execution is crucial for both fighting system complexity and providing system fault tolerance

In conclusion, we would like to emphasise that advanced exception handling models to be employed in concurrent object-oriented systems should relate to

- the development paradigm adhered to (e.g. object-orientation)
- the main implementation features (e.g. information and behaviour encapsulation, typing, inheritance, concurrency, distribution)
- the type of concurrency (competitive, cooperative, disjoint)
- the system development (design) techniques used
- the way systems are structured (objects, classes, actions, modules)
- the application-specific characteristics of the system to be designed

Acknowledgments: Alexander Romanovsky has been supported by the EC IST RTD Project on Dependable Systems of Systems (DSoS). Jörg Kienzle has been partially supported by the Swiss National Science Foundation project FN 2000- 057187.99/1.

References

[1] Banatre, J.P., Issarny, V.: Exception Handling in Communication Sequential Processes. Technical Report 660, INRIA-Rennes, IRISA (1992)
[2] Best, E.: Semantics of Sequential and Parallel Programs. Prentice Hall. London New York (1996)
[3] Buhr, P.A., Mok, W.Y.R.: Advanced Exception Handling Mechanisms. IEEE Transactions on Software Engineering, **SE-26**, 9 (2000)
[4] Campbell, R.H., Randell, B.: Error Recovery in Asynchronous Systems. IEEE Transactions on Software Engineering, **SE-12**, 8 (1986) 811-826
[5] Cristian, F.: Exception Handling and Tolerance of Software Faults. In Lyu, M.R. (ed.): Software Fault Tolerance. Wiley (1994) 81-108
[6] Enterprise JavaBeans. Specification, v.1.1, Sun Microsystems, Inc. (1999)
[7] Goodenough, J.B.: Exception Handling, Issues and a Proposed Notation. Communications of ACM, **18**, 12 (1975) 683-696
[8] Gray, J.N., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, California (1993)
[9] Haines, N., Kindred, D., Morrisett, J.G., Nettles, A.M., Wing, J.M.: Composing First-Class Transactions. ACM Transactions on Programming Languages and Systems, **16**, 6 (1994) 1719-1736
[10] Hoare, C.A.R.: Parallel Programming: an Axiomatic Approach. In Goos, G., Harmanis, J. (eds.): Language Hierarchies and Interfaces. Lecture Notes in Computer Science, Vol. 46. Springer-Verlag, Berlin Heidelberg New York (1976) 11-42
[11] Horning, J.J., Randell, B.: Process Structuring. Computing Surveys, **5** (1974) 69-74
[12] Ichisugi, Y., Yonezawa. A.: Exception Handling and Real Time Features in Object-Oriented Concurrent Language. In Yonezawa, A., Ito, T. (eds.): Concurrency: Theory, Language, and Architecture. Lecture Notes in Computer Science, Vol. 491. Springer-Verlag, Berlin Heidelberg New York (1991) 92-109
[13] Issarny, V.: An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software. Journal of Object-Oriented Programming, **6**, 6 (1993) 29-40
[14] Jimenez-Peris, R., Patino-Martinez, M., Arevalo, S.: TransLib: An Ada 95 Object Oriented Framework for Building Transactional Applications. Computer Systems: Science & Engineering Journal, **15**, 1 (2000) 113-125
[15] Kienzle, J., Romanovsky, A.: Combining Tasking and Transactions: Open Multithreaded Transactions. Presented at the 10th Int. Real-Time Ada Workshop, Avila, Spain (2000) (to be published in AdaLetters, 2000)
[16] Kim, K.H.: Approaches to Mehcanization of the Conversation Sccheme Based on Monitors. IEEE Transactions on Software Engineering, **SE-8**, 3 (1982) 189-197

[17] Kurki-Suonio, R., Mikkonen, T.: Liberating object-oriented modeling from programming-level abstractions. In Bosch, J., Mitchell, S. (eds): Object-Oriented Technology: ECOOP'97 Workshop Reader, Lecture Notes in Computer Science, Vol. 1357. Springer-Verlag, Berlin Heidelberg New York (1998) 195-199

[18] Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice (1990)

[19] Liskov, B.: Distributed Programming in Argus. Communications of the ACM, **31**, 3 (1988) 300-312

[20] Object Management Group Object Transaction Service. Draft 4. OMG. OMG Document (1996)

[21] Parrington, G.D., Shrivastava, S.K., Wheater, S.M., Little, M.C.: The Design and Implementation of Arjuna. USENIX Computing Systems Journal, **8**, 3 (1995) 255-308

[22] Perry, D.E.: The Inscape Environment. In Proc. of the 11th International Conf. On Software Engineering. Pennsylvania (1989) 2-11

[23] Randell, B.: System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, **SE-1**, 6 (1975) 220-232

[24] Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z., Xu, J.: From Recovery Blocks to Coordinated Atomic Actions. In Randell, B., Laprie, J.-C., Kopetz H., Littlewood, B. (eds.): Predictably Dependable Computer Systems. Springer-Verlag, Berlin Heidelberg New York (1995) 87-101

[25] Randell, B., Romanovsky, A., Stroud, R.J., Xu, J., Zorzo, A.F.: Coordinated Atomic Actions: from Concept to Implementation. Computing Dept., University of Newcastle upon Tyne. Technical Report 595 (1997)

[26] Romanovsky, A.: Conversations of Objects. Computer Languages, 21, 3/4 (1995) 147-163

[27] Romanovsky, A., Xu, J., Randell, B.: Exception Handling and Resolution in Distributed Object-Oriented Systems, in Proc. of the 16th International Conference on Distributed Computing Systems, Hong Kong (1996) 545-552

[28] Romanovsky, A.: On Structuring Cooperative and Competitive Concurrent Systems. Computer Journal, **42**, 8 (1999) 627-637

[29] Thomsen, B., Leth, L., Prasad, S., Kuo, T.-S., Kramer, A., Knabe, F., Giacalone, A.: Facile Antigua Release - Programming Guide. TR ECRC-93-20, ECRC GmbH, Germany (1993) http://www.ecrc.de/research/projects/facile/report/report.html

[30] Tripathi, A., Van Oosten, J., Miller, R.: Object-Oriented Concurrent Programming Languages and Systems. Journal of Object-Oriented Programming, **12**, 7 (1999) 22-29

[31] Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., Scheidhauer, R.: Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems, **19**, 5 (1997) 804-851

[32] Wellings, A.J., Burns, A.: Implementing Atomic Actions in Ada 95. IEEE Transactions on Software Engineering, **SE-23**, 2 (1997) 107-123

[33] Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery, in Proc. of the 25th International Symp. on Fault-Tolerant Computing. Pasadena, California (1995) 499-509

[34] Xu, J., Randell, B., Romanovsky, A., Stroud, R. J., Zorzo, A. F., Canver, E., von Henke, F.: Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions, in Proc. of the 29th International Symp. on Fault-Tolerant Computing, Madison, (1999) 68-75

[35] Xu, J., Romanovsky, A., Randell, B.: Concurrent Exception Handling and Resolution in Distributed Object Systems. IEEE Transactions on Parallel and Distributed Systems. **TPDS-11**, 11 (2000) 1019-1032