Coordinated Exception Handling in Real-Time Distributed Object Systems

Alexander Romanovsky, Jie Xu, Brian Randell

Department of Computing Science, University of Newcastle upon Tyne, NE1 7RU, UK
{alexander.romanovsky, jie.xu, brian.randell}@newcastle.ac.uk

Exception handling in complex concurrent and distributed systems (e.g. ones involving cooperating rather than just competing activities) is often a necessary, but difficult task. No widely accepted models or approaches exist in this area. The object-oriented paradigm, for all its structuring benefits, and real-time requirements each add further difficulties to the design and implementation of exception handling in such systems. In this paper, we develop a general structuring framework based on the coordinated atomic (CA) action concept for handling exceptions in distributed object systems, in which exceptions in both the value and the time domain are taken into account. In particular, we attempt to attack several difficult problems related to real-time system design and error recovery, including action-level timing constraints, time-triggered CA actions, and time-dependent exception handling. The proposed framework is then demonstrated and assessed using an industrial real-time application.

Keywords: atomic actions, cooperative concurrency, distributed systems, exception handling, exception resolution, real-time constraints.

1: Introduction

Exception handling and fault tolerance are often necessary in practical concurrent and distributed systems, not least because such systems are often extremely complex. However, this is not an easy task. First, exception handling in concurrent programs is still a difficult, evolving subject: no widely accepted models or approaches exist. Secondly, although most existing distributed systems are to some extent object-oriented (OO) or object-based, the object-oriented paradigm adds a new complication to system design because several aspects of this paradigm conflict with the principle of structured exception handling [1] Thirdly, real-time requirements cause further difficulties with regard to modelling the real-time behaviour of a system and to handling time-related exceptions properly. Therefore, developing a general exception handling approach that can effectively cope with distribution (and concurrency), object orientation, and real-time aspects is, though most desirable, a great challenge.

© 1999 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.

In this paper, we report our first attempt to attack this problem, and describe a structural framework for handling both value and time-dependent exceptions.

→ We establish a simple system model that captures concepts of objects, execution threads, and coordinated atomic (CA) actions. Inter-thread concurrency is classified into three kinds: independent, competitive (with respect to shared objects), and cooperative concurrency. The real-time behaviour of a system is modelled through action-level timing constraints and time-triggered CA actions together with objects that encapsulate real-time data.

→ We then show how to deal with exceptions in a concurrent and distributed environment. An exception can be raised by an execution thread. Other members of the CA action that the thread belongs to must then be informed of this exception. If and when, for some reason, an exception cannot be properly handled within the action, a further exception must be signalled to the enclosing action. We use two algorithmic mechanisms for coordinating exception propagation and exception signalling. In addition, physical distribution and concurrency introduce the possibility of concurrently raised exceptions. We employ an exception graph approach to resolving multiple exceptions.

→ We study five different types of time-related exceptions at the level of CA actions and suggest methods for handling each type properly. Situations in which real-time exceptions, value-related exceptions or both are raised concurrently are also briefly discussed.

→ Finally, we explain how our framework can be applied to a realistic industrial application — the Production Cell III case study (real-time version) [2] — and report our initial experiences. In particular, the coordinated activity that is carried out in the Production Cell when a crane extracts a processed metal blank from an oven is analysed in detail based on our CA action-based implementation.

2: Related Work

This section surveys and discusses several proposals closely related to our work, especially on how to handle exceptions in OO real-time distributed systems with complex concurrent behaviour.

2.1: Real-Time Atomic Actions

The *exchange* scheme [3] offers a restricted form of atomic actions (conversations) which is suitable for some special real-time systems. Triggered by time, a set of processes enter an exchange, setting appropriate recovery points. Processes cooperate within an exchange. If any process fails, all processes are rolled back to their recovery points; thus the exchange terminates by guaranteeing the “nothing” of “all or nothing” semantics. Nested exchanges are not allowed. This approach essentially simplifies action control and recovery.

Colloquies [4] were proposed as a general framework for describing backward recovery in concurrent systems. Different subsets of participants can take part in the execution of different colloquy alternates (called *dialogs*): after a dialog fails all its participants roll back, some of them may leave the colloquy, the others enter the next dialog together perhaps with some new processes.

If a dialog succeeds the colloquy finishes. Colloquies are an execution-time concept. Each process is supposed to declare its participation in a colloquy by describing a sequence of dialogs which it is going to try in order to achieve its goals. This declaration can have time-outs imposed on the participation of this particular process in both the colloquy and each dialog. If the time-out expires the current dialog is aborted, and all its participants are rolled back (they are free to try another dialog), but the process that has been timed out executes its "last ditch" algorithm and leaves the colloquy.

Another scheme intended for real-time applications is the distributed real-time conversation (DRC) scheme [5]. Diversely designed processes (each with two alternates) execute a DRC by executing two interacting sessions in parallel. The primary session is formed by the primary alternates of all processes, the secondary session is formed by their secondary alternates. The scheme is centralised: each session has a leader (one of the participants) which invites other participants to the session. Two leaders cooperate in the sense that they exchange the results of checking their respective acceptance tests. If the primary session fails the test, then the results of the secondary session are used. This is essentially just a fault masking scheme.

2.2: Real-Time Object Models

From the early days of the object-oriented paradigm, many researchers have considered introducing simple deadline mechanisms into the basic object model. However, such a simple approach fails to demonstrate a significant improvement in the design and development of OO hard-real-time distributed systems. The search for appropriate extensions of the object model in this regard has therefore become an important research issue [6].

The RTO.k model proposed in [6, 7] extends the traditional object model by imposing real-time behaviour on objects. It allows both time-triggered and message-triggered methods and deadlines imposed on the execution of each method. This is a very powerful model within which each method can be implemented using software diversity to provide fault masking. All these features make it possible to guarantee that each RTO.k object meets both timing and fault tolerance requirements. We use this work as the basis of the approach to real-time exception handling that we describe in this paper.

2.3: Systems and Languages

The *Dedos* system [8] is a real-time environment that includes many features for programming highly dependable OO hard real-time distributed systems. The *Deal* language is a part of this environment. It extends C++ by introducing features for concurrent programming and for expressing real-time object behaviour. The entire real-time behaviour (timing annotations) is associated with objects, methods and classes, and is expressed in a very abstract way.

The work in [9] extends the concurrent OO language ABCL/1 by introducing features for specifying real-time constraints and an exception handling mechanism. These extensions essentially rely on the concurrent computational model of ABCL/1 within which method calls are regarded as

message transmissions between concurrent objects, and methods as operations initialised by accepting the corresponding messages. Exceptions are treated as signals that can be transmitted between objects. In the extended ABCL/1 timing constraints can be imposed on the synchronous method calls, on the message accepts and on waiting for the results of asynchronous method calls. If a timing constraint is violated, a pre-defined time-out exception is raised. The language has features for informing another object "complaint destination" of any unexpected occurrences during object execution. The complaints can be of four kinds: unaccepted messages, time-outs, system-defined and user-defined. A complaint destination can be declared in each object. Though this scheme allows handling exceptions in a flexible way, it is not intended for cooperative recovery of multiple communicating objects.

DROL [10] is a concurrent OO language which is intended for real-time programming. It has features for imposing time constraints on method executions and for handing violations of these constraints by both caller and callee objects. Distributed protocols for handling timely exceptions are programmed at the meta-level. These protocols describe the intended cooperation of caller, callee and their meta-objects, deal with situations in which time-out notifications can be lost or some parts of the system can be disconnected during method execution, and guarantee the timely continuation of the execution for both caller and callee.

The *Sina l*anguage, which has no exception handling, is proposed in [11] as a remedy for real-time specification inheritance anomalies. Conventional OO languages mix up real-time specifications and constraints with the application code, which makes inheritance difficult. In Sina real-time class specifications are separated from the application code and have the form of real-time filters that can be easily inherited or redefined.

2.4: Remarks

Our analysis of the existing OO real-time distributed systems and languages, including Ada 95, Java, and many others, shows that, though there are many languages that have exception handling mechanisms, only a few support concurrency, and none of them gives a consistent general model for dealing with exceptions in cooperating concurrent systems. Few concurrent OO languages have both real-time and exception handling features! In those that do the exception handling features are often inconsistent with the concurrency features and do not allow cooperative handling of time-related exceptions. It is not well understood how to deal with exceptions in real-time systems and how to deal with time-dependent exceptions even in systems without complex concurrent activities. Although there are many proposals for OO real-time systems, they are not applicable directly for systems that contain complex cooperative concurrency.

There is considerable research on predicting the timing behaviour of OO real-time distributed systems and, in particular, with the static analysis of the worst case execution time for these systems. The *Dedos* environment provides the users with an off-line scheduling scheme which uses timing annotations to enable schedulability analysis. The work reported in [12], which is of a particular interest for fault tolerance, discusses the static timing analysis of Ada exceptions; it is

intended for calculating the worst case execution time for Ada systems with exceptions. This is a very important issue relative to the use of forward error recovery in such systems. The authors analyse the Ada exception mechanism thoroughly and build a formal model of system execution which includes execution of a prologue, a body, handlers and an epilogue for each block. (The model does not, however, address the problem of exceptions in concurrent systems.) However, within this paper we will not address the problems either of static analysis of system schedulability or of run-time scheduling to meet deadlines. These are separate topics and, we believe, existing well-documented results can be applied directly to the system model that we describe in the next section.

3: An Object-Oriented System Model

3.1: Fundamentals

In order to discuss the basic principles of our proposed framework, we must first introduce a simple model to describe the software systems we are considering. We define a *system* as a set of interacting objects. An *object* is a named entity that combines a data structure (internal state) with its associated operations; these operations determine the externally visible behaviour of the object. A *thread* is an active entity that is responsible for executing a sequence of operations on objects. Threads are the agents of computation. (Threads can exist syntactically, e.g. as in the Java language, or as a purely run-time concept.) A system is said to be *concurrent* if it contains multiple threads that behave as though they are all in progress at the same time. In a *distributed* computing environment, this may literally be true — several threads may execute at once, each on its own processing node.

There are at least three kinds of inter-thread concurrency [13]. *Independent* concurrency means concurrent threads have access to only disjoint object sets, without any form of sharing or interacting. *Competitive* concurrency implies that concurrent threads compete for some common objects, but without explicit cooperation. *Cooperative* concurrency occurs in many actual systems, e.g. real-time control applications, where concurrent threads cooperate and interact with each other in pursuit of some joint goal; each thread is responsible only for a part of the joint goal. Cooperation between concurrent threads may be based on various different forms of communication and interaction. We choose to model *inter-thread cooperation* as information transfer via shared objects. Such an abstraction may cover various actual forms of inter-thread cooperation, including inter-thread communication by updating shared objects that have some synchronisation mechanisms, or by message passing (without requiring shared storage), and of inter-thread synchronisation such as condition synchronisation (usually no data passed) and exclusion synchronisation (usually for shared object schemes).

Note that in control applications in particular it is often appropriate to use the object model not just to describe abstract objects within computers, but also the real objects which the computers are

monitoring and controlling. Issues of synchronisation, and perhaps error recovery, are as important between the computers and these real objects, as they are between their abstract objects.

3.2: Coordinated Atomic Actions

Real-life concurrent and distributed systems are often extremely complex. Faults occur and cause errors. Their consequences cannot always be limited to a system component or even a whole computer, or from affecting the environment of the computer system(s). One way to control such complexity, and hence facilitate recovery after an error has been detected, is to somehow restrict interaction and communication between concurrent threads. *Atomic actions* are the usual tool employed in both research and practice to achieve this goal. An action is an abstraction that allows the application programmer to group a set of operations on objects into a logical execution unit. (An action may be associated with desirable properties, e.g. atomicity, consistency, isolation, and durability, often referred to as ACID [14].) An action can also provide a way of gluing multiple execution threads together and enclosing both their normal and their recovery activities. We model here the dynamic structure of a distributed OO system as a set of interacting *coordinated atomic* (CA) *actions* [15, 16]. In fact we use these as a tool for controlling the entire system complexity.

The CA action concept is a generalised form of the basic atomic action structure. A CA action provides a mechanism for performing a group of operations on a collection of (local or external atomic) objects. These operations are performed cooperatively by one or more *roles* executing in parallel within the CA action. The interface to a CA action specifies the objects that are to be manipulated by the CA action and the roles that are to manipulate these objects. To perform a CA action, a group of execution threads must come together and agree to perform each role in the CA action concurrently, with each thread undertaking its appropriate role.

CA actions present a general technique for achieving *fault tolerance* by integrating the concepts of conversations (that enclose cooperative activities), transactions (that ensure consistent access to shared objects), and exception handling (for error recovery) into a uniform structuring framework. More precisely, CA actions use conversations as a mechanism for controlling concurrency and communication between threads that have been designed to cooperate with each other. Concurrent accesses to shared objects that are external to the CA action are controlled by the associated transaction mechanism that guarantees the ACID properties. In particular, objects that are external to the CA action, and can hence be shared with other actions concurrently, must be atomic and individually responsible for their own integrity. In a sense CA actions can be seen as a disciplined approach to using multi-threaded nested transactions [17] and to providing them with well-structured exception handling.

Figure 1 shows a simple example in which two threads enter a CA action to play their respective roles. Within the CA action, the roles communicate with each other and cooperate in pursuit of some common goal. However, during the execution of the CA action, an exception e is raised by one of the roles. The other role is then informed of the exception and both roles transfer control to their respective handlers H_1 and H_2 for this exception, which attempt to perform forward

error recovery. The effects of erroneous operations on external objects are repaired by putting the objects into new correct states so that the CA action is able to exit with an acceptable overall outcome. (As an alternative to performing forward error recovery, the two participating threads could undo the effects of operations on the external objects, roll back and then try again, possibly using diversely-designed software alternates, if the aim is to provide means of tolerating residual design faults.) Note that threads cannot be created or completed inside CA actions; however we believe that this does not make our model less general.

3.3: Modelling Real-Time System Behaviour

When using CA actions to model the dynamic system behaviour, we have to further model the real-time behaviour of an OO system. There are several ways of characterising real-time system behaviour: using real-time objects, or real-time CA actions or a combination of both. As with Kim and Kopetz's proposal for RTO.k objects [6], timing constraints can be precisely attached to both data and operations of an object, including possible time-triggered mechanisms for invoking operations. On the other hand, time-related requirements could be handled at the level of CA actions as well. We prefer an approach that is mainly based on action-level mechanisms, possibly with a complementary object-level mechanism for handling real-time data. Our choice is motivated by the following considerations:

1) It is desirable to minimise any extension of the conventional (i.e. non-real-time) object models, given that the basic characteristics of abstract data types have been widely accepted. An overly sophisticated extension may cause some difficulties in receiving wide acceptance.

2) Rigorous treatment of the temporal behaviour of a distributed system requires a global view of the system. A CA action in our system model is often designed for a joint goal at the application-level, and so modelling the real-time system behaviour at the action-level is of logical importance, especially for time-dependent cooperative activities.

3) Due to the complex cooperation that can be enclosed in a CA action, error recovery local to a single role is often unfeasible; instead global recovery at the action-level will be required. Action-level timing constraints will thus facilitate the design, analysis, and testing of recovery activities. Moreover, proven hardware fault tolerance techniques and typical software fault tolerance schemes often use atomic actions as units of fault confinement and fault tolerance. Action-level timing constraints will give unique treatment to time-dependent faults within the atomic action structure.

4) Time-triggered concurrent activities may contain many different, sequential or concurrent, operations on objects, but time-triggered mechanisms at the operation level do not provide appropriate support for triggering a group of operations jointly. Because a CA action can naturally enclose these activities, a time-triggered mechanism at the action level will offer a much simpler way of precisely triggering cooperative activities at their intended times.

We choose to model the temporal behaviour of a system at the level of CA actions by means of both deadline and time-triggered mechanisms. (Note that our model is more general than will often be needed, but is used here for expository proposes, and because it suits the requirements of the case study presented in Section 6 below.) A CA action declaration with timing constraints might take the form below:

```
CA action action_name (formal parameters)
start[t0, t1] finish[t2, t3] within T;
```

Informally, the above indicates an action that must be started by its participating threads between times `t0` and `t1`, and finished between times `t2` and `t3` with the total execution time being within a relative period of time `T` (see Figure 2). Any violation of these constraints will lead to the raising of a time-related exception in the system.

An action might also be time-triggered. Such an action would be started by the supporting system (e.g. by a scheduler or a real-time clock) when the time comes. Because the action would not be called by the participating threads, the internal threads that play roles of the action would either be forked or resumed automatically (depending on an actual implementation). When the action finished the roles would be joined or suspended. In the time-triggered case, `t0` and `t1` would be used by the scheduler to start the action, and would not be timing constraints for any caller at the application level. However, the execution of an action would still be constrained by `t2`, `t3` and `T`. (Other syntax forms are possible; for example, "**every** `P` **between** `t0` **and** `t1`" might be incorporated into the action declaration, meaning that the action must be executed every `P` time units between the period of time [`t0`, `t1`].)

Without loss of generality, we assume in our model that a given action might be triggered either by messages (i.e. multiple participating threads start the action jointly) or by a real-time clock that causes the automatic creation of multiple internal threads to play their respective roles; a unique *execution instance* of this action will be created in response to time- or message-triggering. A concurrency control mechanism must be used to give higher priorities to the time-triggered execution instances.

Note that, while an operation-level deadline mechanism may be optional in our model, an extension of the conventional object model is unavoidable if real-time data are to be taken into account. Attaching timing constraints to some internal data of an object becomes a natural decision, because we usually assume that a CA action may not retain data. However, real-time object models have been discussed extensively in [6, 7], so we will not address any further details here. Rather, we would like to focus on various new issues at the CA action level, especially on how to handle real-time exceptions in an OO distributed system.

4: Exception Handling in a Concurrent and Distributed Environment

In this section, we discuss how to handle exceptions in a concurrent and distributed environment, based on the system model described in the last section. (Namely, the dynamic structure of a system is regarded as a set of interacting nested CA actions.) *Exception handling* is

viewed here as a general mechanism for coping with exceptional system conditions or *errors* caused by either *hardware faults* or *software faults*.

In many cases, in the sorts of environments we are considering, traditional mechanisms for handling exceptions in sequential programs are no longer appropriate. One major difficulty is that the process of handling an exception may need to involve multiple concurrent components at a time when they are trying to cooperate in order to solve a global problem. Another complication is that several exceptions may be raised concurrently in different nodes of a distributed environment. Existing proposals and actual concurrent languages either ignore these difficulties or only cope with a limited form of them. We introduce in this section a general approach to attacking these problems; the next section will focus on handling real-time exceptions in response to various time violations.

An exception (handling) mechanism is a programming language control structure that allows programmers to describe the replacement of the normal program execution by an exceptional execution when occurrence of an exception (i.e. inconsistency with the program specification and hence an interruption to the normal flow of control) is detected [18]. For any given exception mechanism, *exception contexts* are defined as regions in which the same exceptions are treated in the same way; often these contexts are blocks or procedure bodies. Each context should have a set of associated *exception handlers*, one of which will be called when a corresponding exception is raised. There are different models for changing the control flow, but the *termination* model is most popular and is adopted here. This model assumes that when an exception is raised, the corresponding handler copes with the exception and completes the program execution. If a handler for this exception does not exist in the context or is not able to recover the program, then the exception will be propagated. Such *exception propagation* often goes through a chain of procedure calls or nested blocks where the handler is sought in the exception context containing the context which raised or propagated the exception.

We now describe our approach with respect to the following five aspects.

Exception Declaration: For a given CA action, there are two types of exceptions: those that are totally internal to the CA action and that when raised are to be handled by its own handlers, and those that are known in and are to be signalled to its environment (e.g. its caller or the enclosing action).

All exceptions, $\boldsymbol{e} = \{e_1, e_2, e_3, ...\}$, that are raised within a CA action must be declared within the action definition. The corresponding exception handlers are associated with respective roles that the participating threads are to perform. The exceptions, $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \varepsilon_3,...\}$, that are signalled from a CA action to its environment should be specified in the interface to the CA action. These exceptions are signalled in order to indicate that, although internal exception handling might have been (unsuccessfully) attempted, an unrecoverable exceptional condition has occurred within the action, and/or only incomplete results can be delivered by the action.

There are two special exceptions μ and ∂ in $\boldsymbol{\varepsilon}$. An `undo` exception, μ, implies that the action has been aborted and all of its effect shave been undone. Since `undo` is not always possible, a `failure` exception, ∂, will indicate that the action has been aborted but that its effect may not

have been undone completely. For a nested CA action and its direct-enclosing action, the definitions of $\boldsymbol{e}$ and $\boldsymbol{\varepsilon}$ are fully recursive, namely,

$$\boldsymbol{\varepsilon}_{nested} \prod \boldsymbol{e}_{enclosing}$$

Exception Handling and Propagation: When a thread enters the action to play a specified role, it enters the related exception context. Some or all of the participating threads may later enter nested CA actions. Since the nesting of CA actions results in the nesting of exception contexts, each participating thread of the nested action must be associated with an appropriate set of handlers. Exceptions can be propagated along nested exception contexts, namely the chain of nested CA actions. Three terms are used here to clarify the route of exception propagation: an exception e_i in $\boldsymbol{e}$ is *raised* by a role within a CA action, other roles of the same action are then *informed* of the exception e_i and, if handling the exception within the CA action is not fully successful, a further exception ε_j in $\boldsymbol{\varepsilon}$ will be *signalled* from a nested action to its enclosing action (see Figure 3).

There are at least two ways of signalling an exception from a nested action to its enclosing action. One possibility is that a "leading" role has been pre-defined by the designer, or is determined dynamically, which has the responsibility for signalling an agreed exception to the enclosing action. Another approach, however, adopts a more distributed strategy: each role of the nested action is responsible for signalling its own exception. These exceptions should be the same but in fact might be different. Because an action in our model is required to have the ability of handling concurrent exceptions, the exceptions concurrently signalled from the nested action will be handled simply as if they are concurrently raised in the enclosing action.

Control Flow: The termination model of control flow is used here — in any exceptional situations, handlers take over the duties of participating threads in a CA action and complete the action either successfully or by signalling an exception ε to the enclosing action.

External Objects: Since the effect that a CA action in our system model can be observed only through the committed state of some external objects, once an exception is raised within the CA action and hence error recovery is requested, the related external objects must be treated explicitly and in a coordinated fashion, the aim being to leave them in a consistent state, if at all possible. The standard way of doing this in transaction systems is by restoring the objects to their prior states. However, an exception does not necessarily cause restoration of all the external objects. (Indeed, external objects, particularly real ones in the computers' environment, might not be capable of state restoration.) Appropriate exception handlers may well be able to lead such objects to new valid states. But when it is detected that one or more external shared objects have failed to reach a correct state, a `failure` exception ∂ must be signalled to the enclosing CA action in the hope that it may be able to handle the situation.

Exception Resolution: If several exceptions are raised at the same time, a simple method for resolving the exceptions is to prioritise them. The disadvantage of this scheme is that it does not allow representation of situations where the concurrently raised exceptions are merely manifestations of a different, more complicated, exception. To provide a more general method, an *exception graph* representing an exception hierarchy can be utilised. If several exceptions are raised

concurrently, then the multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions [19]. In principle, each CA action should have its own exception graph.

Figure 4 shows an example of an exception graph containing three primitive exceptions e_1, e_2, e_3 at the level 0. The resolving exception $e_1 \leftrightarrow e_2$ at level one will be raised when e_1 and e_2 are raised concurrently. Similarly, the exception $e_1 \leftrightarrow e_2 \leftrightarrow e_3$ at level two will be raised in order to cover all the three primitive exceptions. This resolving exception may still be handled by the current action, or otherwise the universal exception at level three will be further raised.

It is very important to notice that exception resolution and exception signalling require a set of efficient algorithms for implementing the above principles, that is, controlling information passing between roles and actions, ensuring a consistently resolved exception, and signalling a further exception when the need arises. Due to the limitation of space, the details of the algorithms we have developed and related experimental studies are omitted here; their descriptions and discussions can be found in [20, 21].

5: Dealing with Time-Dependent Exceptions

Time violations are very often signs either of design faults or of the system becoming overloaded (e.g. message traffic jams). In particular, real-time exceptions in atomic actions are often caused by *deserter* processes, i.e. processes that fail to arrive when expected [22] (the time-out mechanism is the most practical way of detecting process desertion). In this respect we distinguish entry and exit desertion for a given CA action since, we believe, they should be treated separately.

The basic idea of coordinating roles' recovery activities within an action [19] can be naturally extended to the treatment of real-time exceptions. Either forward recovery, backward recovery, or a combination of both can be used. The only requirement is to make error recovery fast and cheap and to involve it early enough to be effective. This can be achieved using application-specific knowledge: either provide simple rollback and re-try or design appropriate handlers which move the system into a consistent acceptable state within a short time interval. Moreover, all of the action-related activities such as joint recovery, abortion of nested actions, exception resolution and exception signalling, must be fast as well. We have obtained some experience with the design and implementation of fast recovery and supporting mechanisms; some of our experimental studies are described in [20, 21].

Let us now consider in detail how to handle time violations within our system framework. In Section 3 the syntax example of a CA action implies five possible types of timing constraints: `t0`, `t1`, `t2`, `t3` and `T`. In the interests of simplicity and brevity, we will denote the corresponding real-time exceptions that may be raised at the execution time: e_{t0}, e_{t1}, e_{t2}, e_{t3} and e_T. Note that only one of them can be raised for a given action at a given time.

The first two exceptions are interface exceptions related to action initialisation. Exception e_{t0} is raised when all participating threads have entered the action too early, i.e. before `t0`. This type of

exception must be signalled to, and handled at, the level of the enclosing action. The idea is that, although all of the participating threads are ready to enter the action, the data they have produced or have used before t0 might be old and not from the correct time domain. The containing action is responsible for delaying the action and supplying timely, correct, and fresh data. Exception e_{t1} is raised when some participating threads have not entered the action before t1, which is a clear sign of entry desertion. The containing action should be informed and should presumably try to involve all the threads including the deserter in the recovery activity.

Exception e_{t2} is raised when all roles have completed their execution too early, i.e. before t2. All participating threads should be involved in error recovery, which must produce the intended results within the correct time domain. (A re-try can be used if there is enough time left.) Exception e_{t3} is raised when some roles have not completed their tasks before t3. This is a clear sign of exit desertion. All threads including the deserter must be involved in the recovery activity. Exception e_T can be caused by similar reasons: it is raised when the action does not terminate within time interval T.

There are at least two approaches to trying to guarantee the timely behaviour of the handlers related to the exceptions discussed above. One approach is to design handlers in such a way that their execution time is negligible and so that it can be effectively guaranteed that they will not fail. The other way is to impose timing constraints (e.g. of the type "**within** T") on these handlers as well, and to incorporate such considerations into system timing analyses.

To speed up exception handling, the enclosing action often has to abort all nested actions. Although many languages (e.g. Ada 95 and Java) have features for interrupting separate processes an action-based scheme, should such a facility be needed, would have to provide more sophisticated features: aborting any number of nested actions, executing the abortion handlers, signalling exceptions from the nested actions, and involving all participants of the nested actions in the abortion activity, etc. Pre-emptive CA action schemes [23] provide these features, and can be implemented by the decentralised protocol described in [24].

The CA action framework allows us to express various time-related outcomes of a CA action. If error recovery after a real-time exception has been successful within the action, then its containing action should know nothing about it. If only degraded results can be produced, then an exception associated with the results must be signalled to the containing action. In particular, when, perhaps as a result of deliberately conservative scheduling, enough time remains to undo the effect of the action, the abort exception will be signalled, or otherwise the time-out failure exception will be signalled, meaning that the action has been interrupted but there has not been enough time left to move the system into a consistent state.

The last important problem we will discuss in this section is how to deal with concurrent value and/or timing exceptions. First of all, the general approach discussed in Section 4 is directly applicable if several value exceptions have been raised concurrently within an action with timing constraints. When considering time-related exceptions, we may need a somewhat different policy for exception resolution. For example, although exception e_{t2} (i.e. the action terminates too early)

could be treated in the same way as any value exception, conceptually it cannot be raised concurrently with any other exceptions. Thus, in this case there is no need for exception resolution.

Two other exceptions e_{t3} and e_T are related to exit desertion, and it is possible that one of them is raised concurrently with some value exceptions. We believe that exception resolution should be used here because, generally speaking, recovery with respect to these real-time exceptions must be different if one or more value exceptions have also been raised at the same time. The latter means that the action state has been damaged and that the erroneous information may have spread within the action borders. In this case, returning the system to a correct consistent state becomes more difficult.

The exception graph may have a very special form when one or more value exceptions have been raised concurrently with either e_{t3} or e_T. With our approach, handlers which are designed just for these time-related exceptions are not suitable for concurrent value and timing exceptions and thus such timing exceptions are not included in the resolution graph. Instead, the worst case execution times of all value exception handlers are attached to the nodes of the graph and the resolution procedure will try to find the covering exception which can be handled within the time interval permitted for error recovery. Clearly, the requirement is to design the value exception handlers in such a way that error recovery is fast enough and that it is possible to find a fast covering handler for any concurrently raised exceptions. A modification to this approach could be to use two handlers for each value exception: one for fast recovery when either e_{t3} or e_T has been raised concurrently with this value exception, and another for normal execution when no timing constraint has been violated.

The above discussion has attempted to deal with all the possible, in many cases rather complex, types of situations that might arise. Evidently, in many cases a designer might choose to ignore many of these possibilities and to use a much simplified exception handling system. The issue then will be to justify such a decision, and the use of such a simplified scheme.

6: Case Study: A Real-Time Production Cell

Many practical systems that interact with their environments are incapable of simple backward recovery. Exception handling and forward error recovery are the major means of improving the reliability of such systems. In 1993 an industrial production cell model, taken from a metal-processing plant in Karlsruhe, Germany, was specified (and a controllable graphical simulator provided) as a challenging case study by the Forschungszentrum Informatik (FZI) [25], within the German Korso Project. This case study has attracted wide attention and has been investigated by over 35 different research groups and universities. At Newcastle, we used CA actions as a structuring tool to design a control program for the model and implemented it in the Java language [26]. The program we developed used CA actions to separate the code for the various safety-related requirements from code which merely concerned functional requirements. This program was used to control the FZI simulator, to which it added an animated graphical overlay showing the dynamic creation, progress and completion of the various CA actions. Similar work has

now been done on Production Cell III [2] — a real-time version of the original cell model. (The original Production Cell I model is a rather more complex machine shop than Cell III, but does not consider timing issues and constraints.) Based on our framework and the approaches described in the previous sections, our latest system design and implementation mainly focuses on the guarantee of timing requirements.

Production Cell III consists of eight devices: two conveyor belts (the feed belt and the deposit belt), four processing units (they may be different, e.g. presses and ovens), two portals (each equipped with a travelling crane). Figure 5 shows a top view of the production cell — a screen shot of the simulator provided by FZI (Figure 6 provides an annotated diagram which helps to interpret this screen shot). The task of the cell is to get a metal blank (or plate) from its "environment" via the feed belt and the portal, process the blank by using the specified processing unit(s), and return it to the environment via the other portal and the deposit belt. The blanks are carried in just one direction, i.e. from left to right. Each of the devices is associated with a set of sensors that provide useful information to a control program and a set of actuators through which the control program can have control over the whole system. In particular, a bar-code reader is located at the end of the feed belt to collect information about the processing procedures and timing constraints to be applied to each blank.

More precisely, the production cycle for each blank is as follows: 1) if the barrier light for insertion shows green, a blank may be added, e.g. by the blank supplier, to the feed belt from the environment, 2) the feed belt positions the blank right in front of the bar code reader, 3) following the instruction from the bar-code, portal_1 moves the blank to the specified processing unit(s) in the required order, 4) the processing units process the blank in turn, 5) portal_2 places the blank on the deposit belt, and 6) if the barrier light for deposit is green, the plate may be carried to the environment where a container may be used, e.g. by the blank consumer, to store the processed pieces.

The control program to implement the above functions must satisfy a number of requirements regarding safety (e.g. no machine or blank collisions), liveness and correctness. The correctness requirements contain several time-related constraints, including:

1) a blank must not stay longer than t_G seconds in the cell; and
2) when a blank is being processed by a processing unit i, it must not stay in the unit i longer than max_i and may not leave before min_i seconds.

CA actions, as a design and implementation concept, can provide appropriate support for damage confinement, complexity control, fault tolerance, critical condition validation, and coordination of concurrent activities of various devices. In our current implementation the entire control program was organised as 16 top-level CA actions; each action usually includes one step of the blank processing and typically involves passing a blank between two devices (see Figure 6, in which related CA actions are portrayed as overlays on the FZI simulator diagram). These actions are constructed in such a way that neither blanks nor devices can collide because the mutual

exclusion feature of an action can guarantee that a blank or a device cannot be involved in more than one action at a time. This further ensures a safe deadlock if the actions the devices participate in are wrongly ordered due to design mistakes or certain faults. In addition, each hardware device is associated with an execution thread which is responsible for specifying a sequence of actions the device will participate in. Certain device sensors are also regarded as the participating threads of some actions. All metal blanks are designed as external objects and can be shared by the CA actions.

Again, consider the production cycle for a given blank. Action `loadFeedBelt` first interacts with the Cell's environment, i.e. to receive a blank from the blank producer. The `unloadFeedBelt` action then unloads the feed belt using portal_1. The portal_1 device can move the blank to processing unit 1, 2, 3, or 4 by one of `loadUnit` actions, according to the specified application requirements (only the action that loads unit 1 is shown in the limited space of Figure 6). Action `process_1, 2, 3,` or `4` can then control the processing of the blank within unit 1, 2, 3, or 4. After that, the portal_2 device can unload a specified processing unit through an `UnloadUnit` action (again only the action that unloads unit 4 is shown in Figure 6). The `loadDepositBelt` action then allows portal_2 to drop the blank onto the deposit belt, and finally the `unloadDepositBelt` action delivers the blank to the blank consumer.

We consider here only take the typical `unloadUnit` action that removes a blank from the oven (i.e. unit 3) as an example to explain the action-level timing constraints and how real-time exceptions can be correctly handled. For convenience we term this action `Oven_Portal_Removal`. Figure 7 shows further details of the nested CA actions within the `Oven Portal_Removal` action that enclose cooperative activities of the oven (processing unit 3) and portal_2 in order to remove a blank from the oven at the right time. The nested action `take_out_blank` (from the oven) is a time-critical action, and it must be triggered and finished at the right time. Specifically, this action cannot be started before the time $t + min3$, but must be finished before the time $t + max3$, where t is the time when the blank was put into the oven. Due to its special time-criticality, this action may be triggered by time with a guaranteed priority. The containing action `Oven_Portal_Removal` is less time-critical and can be triggered by the participating threads. However, since there is the system-level deadline t_G for a given blank, all the CA actions that control the activities within the cell (excluding the feed belt and the deposit belt) must be assigned appropriate deadlines derived from the global t_G. For example, the enclosing action in Figure 7 has a deadline T_{OPR} and the nested action has a deadline T_{TOB}.

In our control program, various possible exceptions are defined and the corresponding handlers are provided. Consider the action `take_out_blank` again. There may be an `early_start` exception for this action. Conceptually, this should be handled by the enclosing action (e.g. by delaying the action execution). However, we use a real-time clock to trigger the action precisely and to avoid this kind of `early_start` exception. A `time_out` exception reports the violation of either $t + max3$ or T_{TOB}. The simple recovery measure used in the handler is to take the blank out from the oven immediately and to signal a further exception to the containing action, indicating that

the blank may have been damaged or just partially processed. At the level of the enclosing action `Oven_Portal_Removal`, a concurrent value exception, e.g. `sensor_value_error`, may occur concurrently with the `time_out` exception from the nested action. In this case, the pre-defined resolution graph is searched and a resolving exception that covers both exceptions is identified. When the action `Oven_Portal_Removal` raises a `time_out` exception with respect to T_{OPR}, the forward recovery measure aborts any nested action in progress and removes the blank from the oven immediately (although the blank may be only partially processed). However, non-real-time exceptions can be handled using the backward recovery technique. For example, a temporal portal motor fault can prevent the portal from being ready in time for the oven. The exception handler in our implementation will simply move the portal back to the initial position and re-move it towards the oven again.

Our design and experiments have provided promising evidence that the action-level real-time model offers a practical and rigorous way of structuring and developing safety-critical real-time applications. This is because CA actions as high-level abstract execution units provide an additional abstract layer between the scheduler and concrete operations on objects. This abstract layer can guarantee the system safety requirements, e.g. prevent the building of a system in which blanks and devices can collide, and therefore ease and simplify the task of designing and implementing a scheduler that controls the dynamic execution of CA actions.

The proposed structuring framework used in this industry-oriented case study not only helps to precisely express real-time requirements, but also facilitates the tasks of handling time violation and recovering real-time exceptions. The resulted implementation displays clear program structuring and good system extendibility, especially for a complex system that contains concurrent cooperative activities. Our experience indicates that it would be a much more difficult task to design the system at the object level without the use of the CA action abstraction.

7: Conclusions

This paper has focused on the topic of exception handling in OO real-time distributed systems. Our solutions are intended to be applicable to a wide set of practical real-time systems that interact with their environments (e.g., the production cell application); such systems typically are incapable of the simple backward recovery that is characteristic of transaction-based systems. The OO exception model developed in this paper extends and improves the models which may be found in sequential OO languages, and the non-concurrent models used in some concurrent OO languages. Our model also includes an approach to specifying real-time behaviour of a system at the level of CA actions. Methods for handling various time-related exceptions are identified.

The principal merits of our approach are that it deals with the typical complexity of many industrial real-time systems, in which multiple activities, some competing for shared resources, others cooperating, are going on concurrently, and in which faults can occur, and cannot necessarily be simply masked. It thus emphasises the provision of error recovery, without unduly restricting such error recovery to being backward error recovery — since such a restriction can be quite

unrealistic when one is considering computer systems interactions with the environment. It allows for time constraints to be attached to various activities, and allows for the possibility that such time constraints might in practice be violated — and provides means of exception handling for such occurrences. It does thus in a well-structured fashion which facilitates modular design. The case study described here has encouraged us regarding the benefits that such a strategy provides to the system designer. It is our belief that the task of rigorously validating both the temporal and functional aspects of the design will also benefit from the use of such structuring [27], though this has yet to be demonstrated.

Acknowledgements

This work was supported by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa), and has benefited greatly from discussions with a number of colleagues within the project, in particular R.J. Stroud, I. Welch and A.F. Zorzo at the University of Newcastle upon Tyne, and A. Burns, S. Mitchell, and A.J. Wellings of the University of York. Our thanks go to the anonymous referees for their helpful comments.

References

1 **Miller, R, Tripathi, A** 'Issues with Exception Handling in Object-Oriented Systems', *Proc. ECOOP'97. LNCS-1241,* Finland (1997) pp 85-103

2 **Lotzbeyer, A, Muhlfeld, R** 'Task Description of a Flexible Production Cell with Real Time Properties'. Internal Technical Report. Forschungszentrum Informatik. Karlsruhe. Germany. (ftp.fzi.de/pub/prost/projects/korsys/task_descr_flex_2_2.ps.gz) (1996)

3 **Anderson, T, Knight, J C** 'A Framework for Software Fault-Tolerance in Real-Time Systems', *IEEE Trans. Softw. Engng.* Vol SE-9 No 3 (1983) pp 355-364

4 **Gregory, S T, Knight, J C** 'A new linguistic approach to backward error recovery', *Proc. 15th Int. Symp. on Fault-Tolerant Computing,* Michigan, (1985) pp 404-409

5 **Kim, K H, Bacellar, L** 'Time-Bounded Cooperative Recovery with Distributed Real-Time Conversation Scheme', *Proc. IEEE WORDS'97,* (1997)

6 **Kim, K H, Kopetz, H** 'A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials', *Proc. IEEE COMPSAC'94,* (1994) pp 392-402

7 **Kim, K H, Subbaraman, C** 'Fault-Tolerant Real-Time Objects', *CACM* Vol 40 No 1 (1997) pp 75-82

8 **Hammer, D K, Luit, E J, Roosmalen, O S, van der Stok, P D V, and Verhoosel, J P C** 'Dedos: A Distributed Real-Time Environment', *IEEE Paral. and Distrib. Techn.* Winter (1994) pp 32-47

9 **Ishisugi, Y, Yonezawa, A** 'Exception Handling and Real-time Features in Object-Oriented Concurrent Language', in *Concurrency: Theory, Language, and Architecture* **Yonezawa, A, (Ed.)** LNCS 491 (1991)

10 **Takashio, K, Takoro, M** 'DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems', *Proc. OOPSLA'92,* (1992)

11 **Aksit, M, Bosch, J, van der Sterren, W, and Bergmans, L** 'Real-Time Specification Inheritance Anomalies and Real-Time Filters', *Proc. ECOOP'94, LNCS 821,* (1994) pp 386-407

12 **Chapman, R, Burns, A, and Wellings, A** 'Worst-Case Timing Analysis of Exception Handling in Ada', *Proc. Ada UK Conference,* London, UK (1993)

13 **Hoare, C A R** 'Communicating Sequential Processes', *CACM* Vol 21 No 8 (1978) pp 666-677

14 **Lynch, N A, Merrit, M, Weihl, W E, and Fekete, A** *Atomic Transactions*, Morgan Kaufmann, (1993)

15 **Xu, J, Randell, B, Romanovsky, A, Rubira, C, Stroud, R, and Wu, Z** 'Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery', *Proc. 25th Int. Symp. on Fault-Tolerant Computing,* Pasadena, (1995) pp 499-508

16 **Randell, B, Romanovsky, A, Stroud, R, Xu, J, and Zorzo, A F** 'Coordinated Atomic Actions: from Concept to Implementation'. Department of Computing Science, University of Newcastle upon Tyne, TR No 595 (1997)

17 **Shrivastava, S K, Dixon, G N, and Parrington, G D** 'An Overview of the Arjuna Distributed Programming System', *IEEE Software* Vol 8 No 1 (1991) pp 66-73

18 **Cristian, F** 'Exception Handling and Tolerance of Software Faults', in *Software Fault Tolerance* **Liu, M (Ed.)** J. Wiley (1994)

19 **Campbell, R H, Randell, B** 'Error recovery in asynchronous systems', *IEEE Trans. on Soft. Eng.* Vol SE-12 No 8 (1986) pp 811-826

20 **Xu, J, Romanovsky, A, and Randell, B** 'Exception Handling and Resolution in Distributed Object-Oriented Systems', *Proc. 18th Int. Conference on Distributed Computing Systems*, The Netherlands (1998) pp 12-21

21 **Xu, J, Romanovsky, A, and Randell, B** 'Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation'. Department of Computing Science, University of Newcastle upon Tyne, TR No 612 (1997)

22 **Kim, K H** 'Approaches to mechanization of the conversation scheme based on monitors', *IEEE Trans. Softw. Engng.* Vol SE-8 No 3 (1982) pp 189-197

23 **Romanovsky, A, Randell, B, Stroud, R, Xu, J, and Zorzo, A** 'Implementation of Blocking Coordinated Atomic Actions Based on Forward Error Recovery', *Journal of Syst. Architecture* Vol 43 No 10 (1997) pp 687-699

24 **Romanovsky, A, Xu, J, and Randell, B** 'Exception Handling and Resolution in Distributed Object-Oriented Systems', *Proc. 16th Int. Conference on Distributed Computing Systems,* Hong Kong (1996) pp 545-553

25 **Lewerentz, C, Lindner, T** *Formal Development of Reactive Systems: Case Study "Production Cell", LNCS-891,* Springer-Verlag (1995)

26 **Zorzo, A F, Romanovsky, A, Xu, J, Randell, B, Stroud, R J, and Welch, I S** 'Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study'. 3rd Year Report, ESPRIT LTR Project 20072 on Design for Validation (1998) (to appear in *Software — Practice & Experience*)

27 **Kopetz, H, Nossal, R** 'Temporal Firewalls in Large Distributed Real-Time Systems', *Proc. IEEE Workshop on Future Trends in Distributed Computing,* Tunisia (1997) pp 310-315

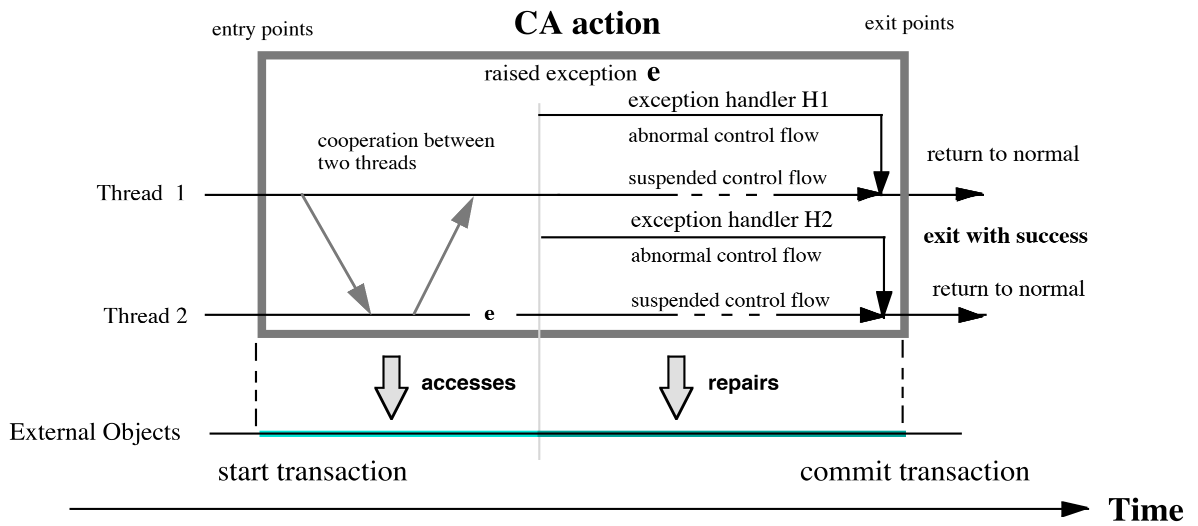

Figure 1 Coordinated error recovery performed by a CA action.

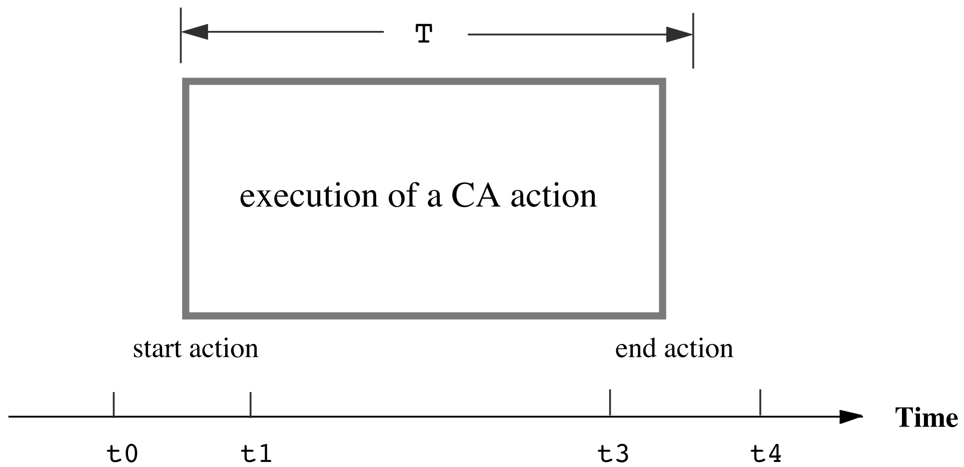

Figure 2 Typical timing constraints on the execution of a CA action.

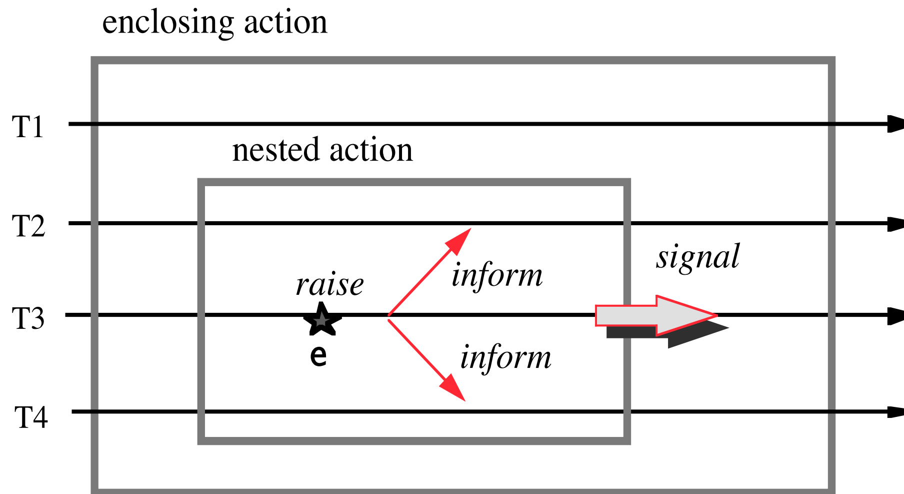

Figure 3 Exception propagation over nesting levels.

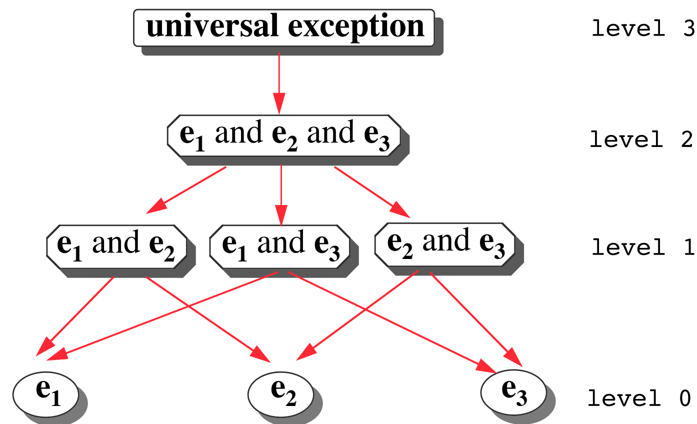

Figure 4 Example of a three-level exception graph.

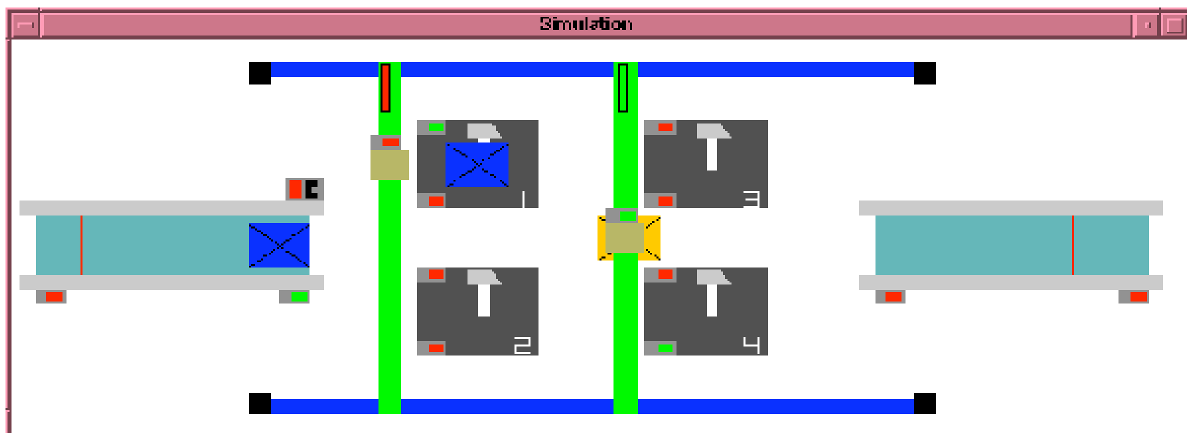

Figure 5 Production Cell III (screen shot).

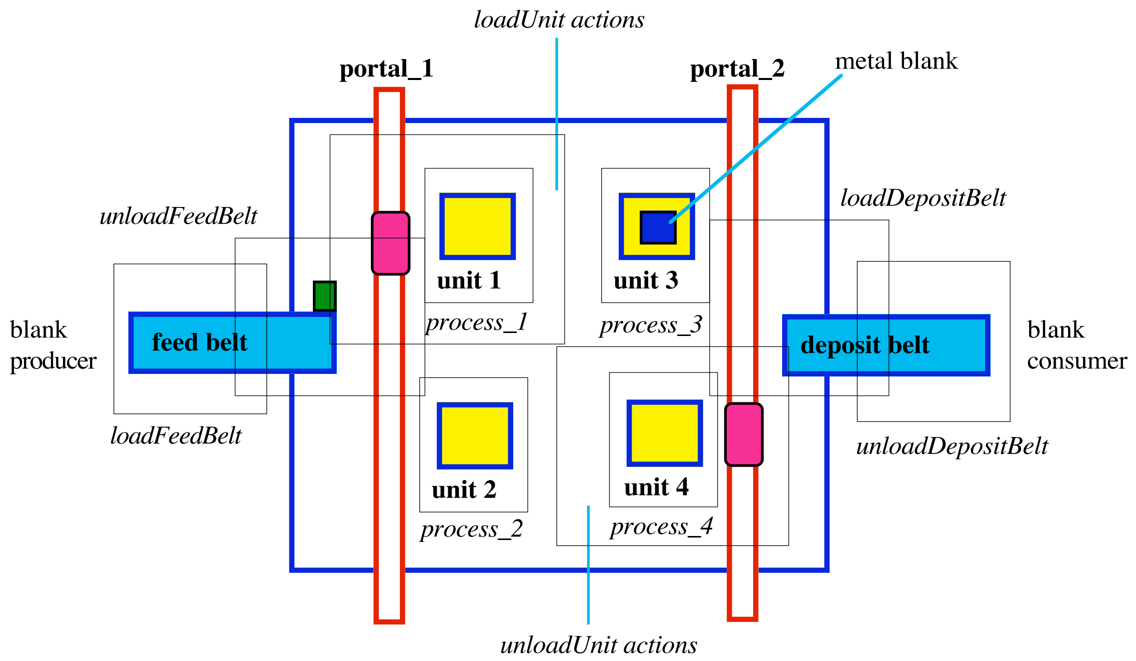

Figure 6 CA actions that control the cell.

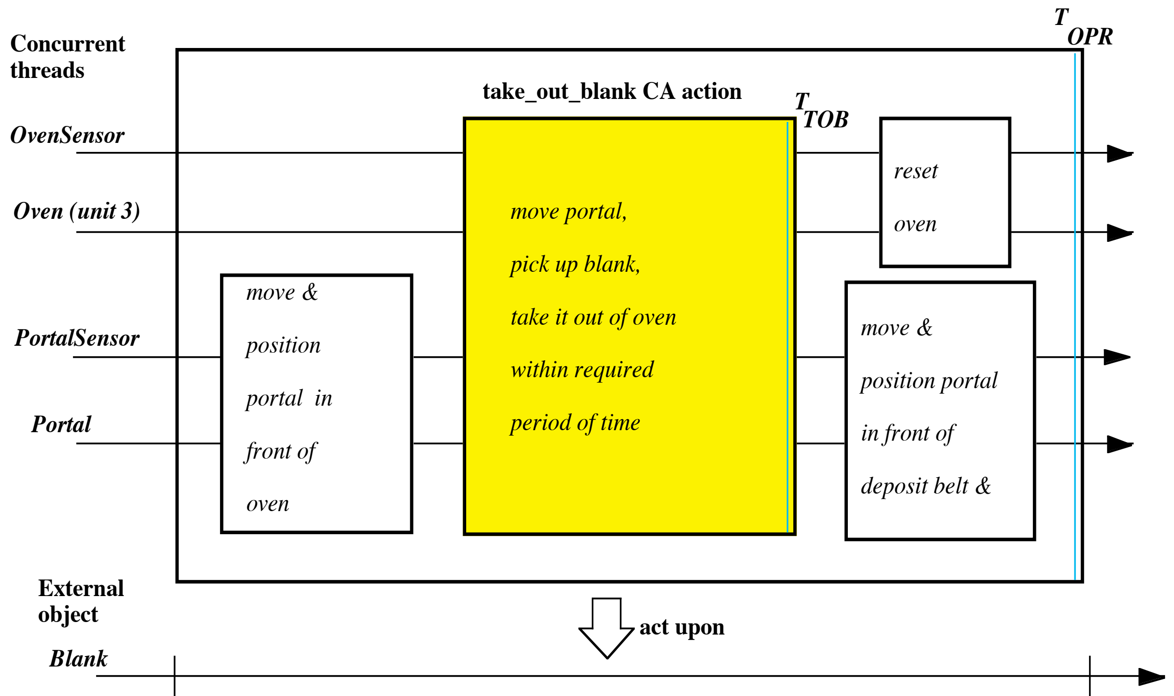

Figure 7 The `Oven_Portal_Removal` action.