Coordinated Atomic Actions:

Formal Model, Case Study and System Implementation

B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, and A.F. Zorzo
University of Newcastle upon Tyne, NE1 7RU, UK

D. Schwier and F. von Henke
University of Ulm, D-89069 Ulm, Germany

ABSTRACT

The Coordinated Atomic Action (or CA action) concept is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components in a distributed object system. It provides a conceptual framework for dealing with different kinds of concurrency and achieving fault tolerance by integrating and extending two complementary concepts — conversations and transactions. Conversations (enhanced with concurrent exception handling) 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.

This paper first presents a formal description of the CA action concept based on a linear-time temporal logic system and then demonstrates the practical utility of CA actions through an industrial safety-critical application — the Fault-Tolerant Production Cell case study. A description of an experimental prototype implementation of CA actions is used to illustrate how support can be provided to the application layer for developing fault-tolerant programs that use CA actions as a structuring tool.

Key Words — Atomicity, concurrency, nested transactions, object-oriented programming, (software) fault tolerance.

1 Introduction

1.1 Background and motivation

Distributed computing often gives rise to complex concurrent and interacting activities. In some cases several concurrent activities may be working together, i.e. *cooperating*, to solve a given problem; in other cases the activities may be completely *independent* or essentially independent but needing to *compete* for shared common system resources. In practice, different kinds of concurrency might co-exist in a complex application which thus will require a general supporting mechanism for controlling and coordinating its various concurrent activities.

Due in no small measure to their complexity, concurrent and distributed systems are very prone to faults and errors. Various fault tolerance techniques for coping with hardware and software faults can provide a practical way of improving the dependability of such systems. Moreover, because certain faults can have an impact on, or arise from, the environment of a computing system [Campbell & Randell 1986], some forms of error recovery may require stepping outside the boundaries of a computer system (i.e. considering the computer system and its environment recursively as an entire distributed system at a higher level of abstraction). In current practice, however, the majority of fault-tolerant computing systems do not attempt to tolerate software faults, or facilitate recovery from errors that affect both the computer system and its environment — rather they concentrate on the problems that arise from operational faults (typically hardware faults). For example, many software systems that use the concept of atomic (trans)actions to construct fault-tolerant distributed applications generally assume that erroneous outputs can be detected before transaction commitment occurs, and that user programs are correct.

Increasingly, software is constructed according to the object-oriented programming methodology which supports clean structuring, simplicity of design and software reuse. If used correctly, object-oriented programming can improve software dependability. However, many of today's computing systems have to deal with extremely complex situations, and given their own resultant complexity, it still seems inevitable that such systems will contain some residual (typically software) design faults or “bugs”. Approaches and mechanisms, such as exception handling and software fault tolerance, are therefore required in order to cope with abnormal run-time events and software bugs. Practical techniques for dealing with exceptions and software faults do exist, especially for sequential systems, and have been proved successful for a variety of

applications [Lyu 1995]. But the great challenge is to extend these techniques to cope with the full potential complexity of a distributed object system.

Existing models and techniques are to some extent ill-suited for many real-world applications where there are specific needs for cooperation and coordination among multiple concurrent activities. On the one hand, it is becoming ever clearer that the traditional (nested) transaction model does not provide satisfactory support for cooperation between concurrent activities [Elmagarmid 1993]. On the other hand, although the *conversation* concept [Randell 1975] provides full support for cooperative concurrent activities in process-oriented systems such as process control or real-time control applications, without special support for object interactions and consistent access to shared objects it has proved difficult to use the conversation concept to control concurrency and facilitate error recovery in an object system [Gregory & Knight 1989][Xu et al 1995].

In this paper we discuss an approach to structuring dependable distributed object systems using a general scheme, called *Coordinated Atomic Actions* or CA actions (first introduced in [Xu et al 1995]), that integrates and extends conversations and transactions in order to support different kinds of (independent, competitive, and cooperative) concurrency and to provide very general forms of fault tolerance.

1.2 Coordinated atomic actions: overview and example

A CA action is a mechanism for coordinating multi-threaded interactions and ensuring consistent access to objects in the presence of competitive concurrency and potential faults. In order to support backward error recovery, a CA action must provide a recovery line which coordinates the recovery points of the objects and execution threads participating in the action so as to avoid the *domino effect* [Randell 1975]. To support forward error recovery, a CA action must provide an effective means of coordinating the use of exception handlers. An *acceptance test* can and ideally should be provided in order to determine whether the outcome of the CA action is successful. The various threads participating in a given CA action enter and leave the action synchronously. If an error is detected inside a CA action, appropriate forward and/or backward recovery measures must be invoked cooperatively in order to reach some mutually consistent conclusion. Error recovery for participating threads of a CA action generally requires the use of explicit error coordination mechanisms within the CA action (e.g. the conversation scheme plus concurrent exception handling [Campbell & Randell 1986]); objects that are external to the CA action and can be shared with other actions and threads must provide their own error coordination mechanisms. These external objects must behave atomically with respect to other CA actions and threads so that they cannot

be used as an implicit means of “smuggling” information [Kim 1982] into or out of a CA action.

Figure 1 shows an example in which two threads enter a CA action synchronously through different entry points. Within the CA action the threads 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 threads. The exception is propagated to the other thread and both threads transfer control to their exception handlers H_1 and H_2 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 pass its acceptance test and exit with an acceptable 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. This would require the use of diversely-designed software alternates if the aim was to tolerate residual design faults.)

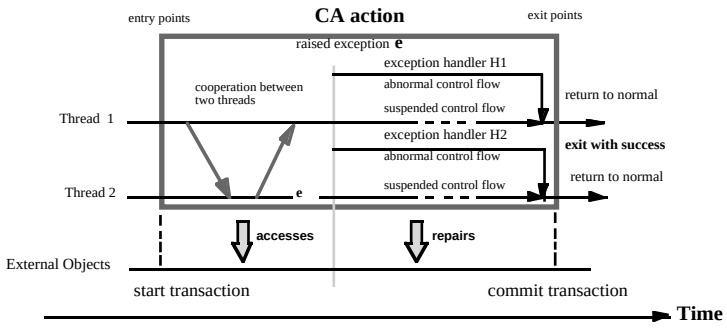

Figure 1 Coordinated error recovery performed by a CA action.

1.3 Related work

The traditional domain for transaction processing is database systems [Gray 1978]. C.T. Davies pioneered the development of the atomic transaction concept [Davies 1973][Davies 1978]. He addressed many concepts concerned with concurrent systems, recovery and integrity within an overall scheme that he called *data processing spheres of control*. However, subsequent early work on transactions, though influenced by Davies, was much less ambitious in its goals.

Basic transaction systems do not allow for subtransactions or support concurrency within a transaction. Nested transactions [Moss 1981] extend the flat transaction model by providing the independent failure property for subtransactions, and supporting competitive concurrency within a containing transaction. A number of generalized transaction models have been developed recently in order to overcome some of the limitations of traditional (flat or nested) transactions, such as lack of support for long-

lived actions, cooperative activities and multidatabase systems. Much of this work is surveyed comprehensively in [Elmagarmid 1993].

Several experimental systems have been developed that successfully combine transaction processing with the object-oriented programming methodology, e.g. Argus [Liskov 1988] and Arjuna [Parrington et al 1995]. These systems offer support for nested transactions and provide a powerful linguistic base for developing reliable distributed programs. Competitive concurrency is well controlled and fully supported in such systems, but no support is provided for cooperative concurrency.

Conversations and their variations [Randell 1975][Campbell & Randell 1986] provide effective support for cooperative activities in process-oriented systems, and moreover pay much attention to the problem of error recovery and fault tolerance. The Arche language [Issarny 1993] extended some ideas behind the conversation concept to provide support for object replication and concurrent exception handling in a distributed object-oriented environment, using a construct called a multi-operation, which is a simplified form of the multi-function construct introduced in [Banatre et al 1986].

Recently, there have been a number of proposals for "multiparty interaction mechanisms" [Jung & Smolka 1996], intended to coordinate interactions among multiple concurrent processes in distributed programming, e.g. Interacting Processes (or IP) [Francez & Forman 1996]. However, to the best of our knowledge, none of the proposals published to date provides a means for fault tolerance or for avoiding interference from unrelated processes and parties.

CA actions were originally introduced in [Xu et al 1995] as a structuring or design concept but their semantics were described only informally and using a very simple example. In this paper we provide a formal description of the CA action concept based on a linear-time temporal logic system [Lamport 1994][Manna & Pnueli 1992]. We then illustrate how CA actions can be applied to a realistic industrial application — the Fault-Tolerant Production Cell [Lewerentz & Lindner 1995][Lotzbeyer & Muhlfeld 1996] — and describe the prototype implementation of the CA action mechanism that was used to develop the demonstration application.

2 Formal description of the CA action concept

In this section we present a formal model for CA actions and use it to demonstrate some of their important characteristics.

2.1 Temporal logic

In this paper, a linear-time temporal logic system is used as a specification language for specifying and proving properties of the CA action concept. Lamport et al [Lamport

1994][Manna & Pnueli 1992] gave a detailed description of how to specify and refine programs and their properties within a temporal logic framework. Due to the limitation of space, we will omit details of their techniques and focus on the part to be used in formalizing the properties of CA actions.

The execution of a CA action *A* generates a sequence of program states, called an execution instance of *A*. When all execution instances of *A* always make a temporal formula φ true, φ will be called a property of *A*. A formula φ is constructed from variables, functions, predicates, the separator $\bullet$, the boolean operators $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\Rightarrow$ (implication), and the temporal operators $\Diamond$ (eventually), $\Box$ (henceforth), and $\underline{\Diamond}$ (sometime in the past).

2.2 Fundamentals

In order to formalize the CA action concept, we need 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 some associated *operations*; these operations determine the externally visible behaviour of the object. A *thread* is an agent of computation and an active entity that is responsible for executing a sequence of operations on objects. A system is said to be *concurrent* if it contains multiple threads that behave as though they are all in progress at one time. In a *distributed* or parallel computing environment, this may literally be true — several threads may execute at once, each on its own processing node.

A CA action provides a mechanism for performing a group of operations on a collection of *local* and *external* objects. Within the CA action these operations are performed cooperatively by one or more threads undertaking *roles* that execute in parallel. A role specifies a sequence of operations on a set of objects. In order 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.

Let *C* be the set of action execution instances. When it causes no confusion, we will use the phrase "action *c*" instead of "action instance *c*". The set *roles*(*c*) contains all roles of action *c*. There are several important sets of objects: *ex-objects*(*c*) is the set of external objects of action *c* which can be shared with other actions and threads, *local-objects*(*c*) is the set of local objects of *c* which can be used by *c* only, and *objects*(*s*) is the set of objects on which the operation sequence *s* is performed, where *s* is a role or a single operation. For any $c \in C$, we consider two basic predicates: *begin*(*c*) is true in a state where action *c* begins, while *end*(*c*) is true in a state where action *c* ends. Finally, *old-value*(*o*, *c*) refers to the value of object *o* in the state where *begin*(*c*) is true, and *new-value*(*o*, *c*) to the value of *o* in the state where *end*(*c*) is true.

2.3 Properties of CA actions

We will formalize four classes of properties of CA actions. Elementary properties characterize relative orders of (nested) action-related execution states. Interface properties specify action entrances, exits and possible outcomes with respect to pre- and post-conditions. The enclosure property enforces the exclusive membership. Finally, fault tolerance properties are relevant to exception handling and error recovery.

2.3.1 Elementary properties

A CA action *c* that ends must have begun earlier.

$\forall\, c \in C \quad \bullet\ \Box\ (end(c) \Rightarrow \underline{\Diamond}\ begin(c))$

A CA action c that begins will end at some time in the future.

$\forall\, c \in C \quad \bullet\ \Box\ (begin(c) \Rightarrow \Diamond\ end(c))$

Let *parent*(*d*) be a function whose return value is the parent or enclosing action of a nested CA action *d*. If a nested action *d* begins, its enclosing action *c* must have begun earlier; if the enclosing action *c* ends, then the nested action *d* must have ended earlier.

$\forall\, c, d \in C \mid c = parent(d) \quad \bullet\ \Box\ (begin(d) \Rightarrow \underline{\Diamond}\ begin(c))$

$\forall\, c, d \in C \mid c = parent(d) \quad \bullet\ \Box\ (end(c) \Rightarrow \underline{\Diamond}\ end(d))$

2.3.2 Interface properties

We need to define two new predicates in order to specify the interface properties: $called(r)$ is true at the point in time when role *r* is called by a thread (typically undertaking a role of an enclosing action), and *return*(*r*) is true at the point in time when *r* returns to its caller.

If a CA action *c* begins, all of its roles must have been called earlier.

$\forall\, c \in C \bullet \Box\ (begin(c) \Rightarrow \forall\, r \in roles(c) \bullet \underline{\Diamond}\ called(r))$

If a CA action *c* ends, all of its roles must return in due course.

$\forall\, c \in C \bullet \Box\ (end(c) \Rightarrow \forall\, r \in roles(c) \bullet \Diamond\ return(r))$

There are four categories of possible outcomes provided by an action *c*, that is, normal outcomes, exceptional outcomes, no outcome (when the action is aborted) and outcomes caused by failure. With respect to different outcome categories the final state *end*(*c*) can be further classified into four mutually exclusive sub-states.

$end(c) ::= normalend(c) \vee exceptionalend(c) \vee abortedend(c) \vee failedend(c)$

In order to characterize these sub-states, we associate three predicates with the pre- and post-conditions of the action: *pre*(*ex-objects*(*c*)) is true when the pre-condition holds on the values of external objects of *c*, *n-post*(*ex-objects*(*c*)) is true when the post condition for the normal execution of *c* is met, and *e-post*(*ex-objects*(*c*)) is true when one of the post-conditions for exceptional execution of *c* is satisfied. In addition, *accept*(*ex-objects*(*c*)) is true when the values of external objects of *c* pass the acceptance test. To simplify the notation, we replace *ex-objects*(*c*) with *c* in the predicate representations.

If a CA action *c* began with the satisfied pre-condition and ends normally, its normal post-condition should hold.

$\forall\, c \in C \bullet \Box\, ((\underline{\Diamond}\, (begin(c) \wedge pre(c)) \wedge normalend(c)) \Rightarrow n\text{-}post(c))$

If a CA action *c* began with the satisfied pre-condition and ends exceptionally (with a signalled exception), then one of its exceptional post-conditions should hold.

$\forall\, c \in C \bullet \Box\, ((\underline{\Diamond}\, (begin(c) \wedge pre(c)) \wedge exceptionalend(c)) \Rightarrow e\text{-}post(c))$

If a CA action *c* ends with abortion, all its effects on the external objects should have been undone.

$\forall\, c \in C, o \in ex\text{-}objects(c) \bullet \Box\, (abortedend(c) \Rightarrow (new\text{-}value(o, c) = old\text{-}value(o, c)))$

Acceptance tests are usually regarded as an implementation technique, and can be used in practice to examine whether the post-conditions will really hold. Ideally, we can design an acceptance test that provides a necessary and sufficient check, i.e. is such that

$\forall\, c \in C \bullet \Box\, (accept(c) \Leftrightarrow n\text{-}post(c) \vee e\text{-}post(c))$

but in practice one often has to settle for a test that is necessary but not sufficient, i.e.

$\forall\, c \in C \bullet \Box\, (n\text{-}post(c) \vee e\text{-}post(c) \Rightarrow accept(c))$

2.3.3 Enclosure property

The enclosure property $\boldsymbol{P}_e$ of a CA action enforces its exclusive membership; that is, 1) information can be only transferred at the entrance and exit of an action, and 2) during the execution of an action a role inside the action cannot interact in any way with a role or thread that is not in the action.

Note that there are only two means of passing information in an action, i.e. local objects and external objects. Objects local to an action *c* can be used only by *c* and no information can be transferred via such objects from or to other actions or threads outside *c*. However, objects external to an action *c* can be shared with other actions and/or with the threads outside *c*, and could therefore cause information to be smuggled

out of or into action *c* during its execution, depending upon how they are shared. Such smuggling must be prevented.

If an action *c* is executed as a whole with respect to its external objects, i.e. no other actions or outside threads can use those objects during the execution of *c*, no information smuggling is possible. In fact, this is too restrictive and in practice, by carefully interleaving the operations of other actions and/or threads on the external objects of *c*, it is possible to increase concurrency and improve performance without causing information smuggling. We will derive two conditions that the concurrent (interleaved) execution of two actions or an action and an outside thread must satisfy in order to ensure $\boldsymbol{P}_e$ with respect to their shared objects.

Let C_l be the set of action instances at the same level of nesting within a given enclosing action, i.e. $\forall\ c, d \in C_l$, *parent*(*c*) = *parent*(*d*). We will first define three relationships between actions and roles at a given level of nesting.

Two actions at the same level of nesting are said to be *strictly serialized* if their executions do not overlap at any point in time, that is

$\forall\ c, d \in C_l \bullet \Box\ ((begin(d) \wedge \underline{\Diamond}\ begin(c)) \Rightarrow \underline{\Diamond}\ end(c))$

We use $c \gg d$ to represent execution sequences that satisfy:

1) $\forall\ c, d \in C_l \bullet \Box\ (begin(d) \Rightarrow \underline{\Diamond}\ end(c))$ and

2) $\forall\ o \in$ *ex-objects*(*c*) $\cap$ *ex-objects*(*d*) $\bullet$ (*new-value*(*o*, *c*) = *old-value*(*o*, *d*))

Two actions at the same level of nesting are said to be *concurrent* if there is a point in time at which both actions are performing their own operations, that is

$\forall\ c, d \in C_l \bullet \Box\ ((begin(d) \wedge \underline{\Diamond}\ begin(c)) \Rightarrow \neg\ \underline{\Diamond}\ end(c))$

We use $c \parallel d$ to represent execution sequences where *c* and *d* are concurrent.

Let us now consider a more complex relationship at a given level of nesting between an action *c* and the roles of *parent*(*c*) that do not participate in *c*, i.e. that bypass *c* (see Figure 2). Let *p* = *parent*(*c*), and we define *bypass*(*p*, *c*) as the set of such roles of *p*. For any $i \in$ *bypass*(*p*, *c*), if *ex-objects*(*c*) $\cap$ *objects*(*i*) $\neq \varnothing$, i.e. if role *i* has access to some of *c*'s external objects, then we have to ensure that the operations of role *i* on these external objects do not interfere with *c*. Here we define *ex-op*(*i*, *c*) as the set of such operations. (For a given operation *op*, predicate *called*(*op*) is true at the point in time when *op* is called by a thread and *return*(*op*) is true when *op* returns to its caller.)

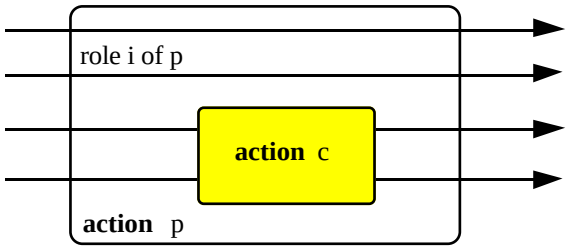

Figure 2 Action c and the bypass role i.

The interleaved execution of *c* and role *i* is said to be *proper* if action *c* is executed as a whole (i.e. no operation is interleaved into the execution of *c*) between the operations in *ex-op*(*i*, *c*), that is

$\forall\ c \in C, i \in bypass(parent(c), c), op \in ex\text{-}op(i, c)$

- $\Box\ ((begin(c) \Rightarrow \underline{\Diamond}\ return(op)) \vee (end(c) \Rightarrow \Diamond\ called(op)))$

We use $c\ |^p\ i$ to represent properly interleaved execution sequences, and use $c \parallel i$ to represent interleaved execution sequences of *c* and role *i*.

Let *end-values*(*o*, *s*) be the set of end values of *o* derived from possible execution sequences *s*. We can now state the serializability of CA actions formally:

1) The concurrent (interleaved) execution of two actions at the same level of nesting is *serializable* if their effects on the intersection of their external objects are always equivalent to the effects of the strictly serial execution of the two actions, that is

$\forall\ c, d \in C_l, o \in ex\text{-}objects(c) \cap ex\text{-}objects(d)$

- $\Box\ (end\text{-}values(o, c \parallel d) \in end\text{-}values(o, c \gg d \cup d \gg c))$

2) The concurrent (interleaved) execution of a CA action and a bypass role of its parent is serializable if the execution is equivalent to one of their proper executions, that is

$\forall\ c \in C, i \in bypass(parent(c), c), o \in ex\text{-}objects(c) \cap objects(i)$

- $(end\text{-}values(o, c \parallel i) \in end\text{-}values(o, c\ |^p\ i))$

where *end-values*(*o*, *c* || *i*) and *end-values*(*o*, $c\ |^p\ i$) refer to the values of *o* in the state where *end*(*c*) is true.

It follows that the following theorem gives sufficient conditions for the enclosure property $\boldsymbol{P}_e$ of a CA action to hold despite the sharing of its external objects.

Theorem : No information can be transferred across the side-firewalls of a CA action c $\in C$ *if with respect to any o* $\in$ *ex-objects*(*c*),

1) For any d $\in C$, *only serializable interleaved executions of c and d are allowed, and*

2) For any $i \in bypass(parent(c), c)$, only serializable interleaved executions of c and role i are allowed.

2.3.4 Fault tolerance properties

This subsection further characterizes the interface properties of an action from a fault tolerance point of view. A CA action $c \in C$ should have a simple and deterministic behaviour, that is, either end normally or signal an appropriate exception to its enclosing action (or its environment). The termination model for exception handling is used here, i.e. when an exception is raised within an action, the corresponding exception handler copes with the exception and completes the action execution.

Let $e(c)$ be the set of (internal) exceptions that can occur during the execution of action c, and $\varepsilon(c)$ be the set of exceptions that c can signal externally. We define two new state predicates: *raise*(e, c) becomes true in the state where an exception e is raised within c, and *signal*(e, c) becomes true in the state where action c signals an exception e to its enclosing action or its user environment. Whenever an exception e is raised in c or signalled to c, the predicate *exception*(e, c) will become true. The effects of error recovery are indicated by two additional state predicates: *full-recovery*(c) is true if all error recovery performed by c is fully successful, and *undone*(c) is true if the undo operation with respect to the effects of c is fully successful.

The normal end of an action can be reached if no exception occurs or error recovery is fully successful, that is

$$\forall\, c \in C \bullet \Box\ (normalend(c) \Rightarrow (\neg\, \exists\, e \in e(c) \bullet \underline{\Diamond} exception(e, c)) \vee \underline{\Diamond} full\text{-}recovery(c)))$$

The exceptional end of an action implies the signalling of an appropriate exception e that specifies an exceptional outcome, that is

$$\forall\, c \in C \bullet \Box\ (exceptionalend(c) \Rightarrow \exists\, e \in \varepsilon(c) \bullet signal(e, c))$$

The aborted end of an action removes any effect that the action may have had on its external objects and signals a special abortion exception `abort`, that is

$$\forall\, c \in C \bullet \Box\ (abortedend(c) \Rightarrow signal(\texttt{abort}, c) \wedge \underline{\Diamond}\ undone(c))$$

The failed end of an action is reached if error recovery is not possible. In this worst case, a special failure exception `fail` is signalled, that is

$$\forall\, c \in C \bullet \Box\ (failedend(c) \Rightarrow signal(\texttt{fail}, c))$$

Within a CA action it is also important to study the relative ordering of the execution states related to exceptions and exception handling in the interests of fault tolerance. We

define *handling*(*e, r, c*) as a state predicate that is true in the state where role *r* of action *c* starts handling the exception *e*.

If an exception is raised within an action and cannot be handled locally by a role, then all the roles of the action must handle the exception cooperatively:

$$\forall\, c \in C, e \in e(c) \bullet \Box\ (raise(e, c) \Rightarrow \Diamond\ (\forall\, r \in roles(c) \bullet handling(e, r, c)))$$

The signalling of an exception from a nested action will cause all the roles of the enclosing action handle the exception cooperatively:

$$\forall\, c \in C, e \in \varepsilon(c) \bullet \Box\ (signal(e, c) \Rightarrow \Diamond\ (\forall\, r \in roles(parent(c)) \bullet handling(e, r, parent(c))))$$

Note that all the roles within a given action are supposed to be able to handle the same exception. However, in a complex concurrent system, there is a possible complication that several exceptions can occur at the same time and thus require a process of exception resolution [Campbell & Randell 1986][Romanovsky et al 1996]. We now define a new predicate *resolving*(*c*) that is true in the state action *c* starts resolving multiple exceptions raised concurrently, and define $R(\{e_1, ..., e_k\})$ as a set function that returns an exception which covers all the exceptions $e_1, ..., e_k$ that occurred concurrently. Two or more concurrent exceptions, raised in *c* or signalled to *c*, will lead to the state that *c* starts resolving these exceptions using function $R(\{e_1, ..., e_k\})$.

$$\forall\, c \in C \bullet \forall\, ce(c) \subseteq e(c)\ \mid\ |ce(c)| \geq 2 \bullet \Box\ (\bigwedge_{e \in ce(c)} exception(e, c) \Rightarrow \Diamond\ resolving(c))$$

$$\forall\, c \in C \bullet \forall\, ce(c) \subseteq e(c)\ \mid\ |ce(c)| \geq 2 \bullet \Box\ (resolving(c) \Rightarrow \Diamond\ (\forall\, r \in roles(c) \bullet handling(R(ce(c)), r, c)))$$

Remarks: This formalization of CA actions provides what we hope readers will agree is a description of the properties of CA actions that is precise but readily-understandable. We have used the formalization as the basis for a theorem about CA actions in section 2.3.3, which states two useful conditions for ensuring the absence of information smuggling. For reasons of space we have not included any other theorems.

3 Case study: Fault-Tolerant Production Cell

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 FZI in 1993 [Lewerentz & Lindner 1995], 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 [Zorzo et al 1997]. The control program we developed was then linked to the FZI simulator, demonstrating a good guarantee of functional and safety-related requirements.

Similar work has now been done on Production Cell II, an extended version of the original cell model, which contains an extra press together with additional sensors and traffic systems for fault detection and tolerance [Lotzbeyer & Muhlfeld 1996]. The specification of Production Cell II recognizes that devices, sensors and actuators can fail, so the required control program becomes much more complex than the program for the non-fault-tolerant production cell as it has to cope with both normal and many exceptional situations. The system design and implementation for Production Cell II described in this section demonstrate how CA actions can aid the structuring of systems that have to adhere to functional and safety-related requirements even in the presence of hardware and software faults.

Production Cell II consists of six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extensible arms each equipped with an electromagnet (see Figure 3, which is taken from the FZI supplied simulator). These devices are associated with a set of sensors that provide information to a control program and a set of actuators via which the control program can exercise control over the whole system. The task of the cell is to get a metal blank from its “environment” via the feed belt, transform it into a forged plate by using a press, and return it to the environment via the deposit belt.

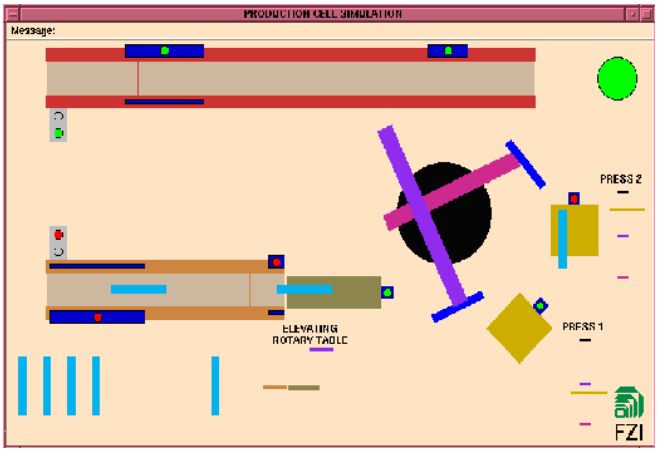

Figure 3 Fault Tolerant Production Cell (top view).

More precisely, the production cycle for each blank is as follows: 1) if the traffic light regulating insertion shows green, a blank may be added, e.g. by the blank supplier, to

the feed belt (bottom of Figure 3) from the environment, 2) the feed belt conveys the blank to the table, 3) the table rotates and lifts the blank to the position where the robot can use its magnet to pick the blank up, 4) arm1 of the robot takes the blank and places it into an unoccupied press, 5) the press forges the blank, 6) arm2 places the forged plate on the deposit belt (top of Figure 3), and 7) if the traffic light controlling the deposit belt is green, the plate may be forwarded further and carried to the environment where a container may be used, e.g. by the blank consumer, to store the forged pieces. Two presses are fed by the robot alternately. If one of the presses is non-operational, the robot must use the other press exclusively.

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. These requirements must be met even when one of the two presses has failed. Our design for the control program separates the safety, functionality, and efficiency requirements between a set of CA actions that occur during the system execution and a set of device/sensor controllers that determine the order in which the CA actions are executed. The safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor controllers. We have designed 12 top-level CA actions; each action includes one step of the blank processing and typically involves passing a blank between two devices.

A blank is designed as an external object. One role of a CA action takes the blank as an input argument, and another takes it as an output argument. The device corresponding to the role that has the blank as an input argument passes the blank to the role that has the blank as an output argument. Some actions also have sensor roles that check whether or not the devices are in the right position. Figure 4 portrays the 12 related CA actions as overlays on the FZI simulator diagram.

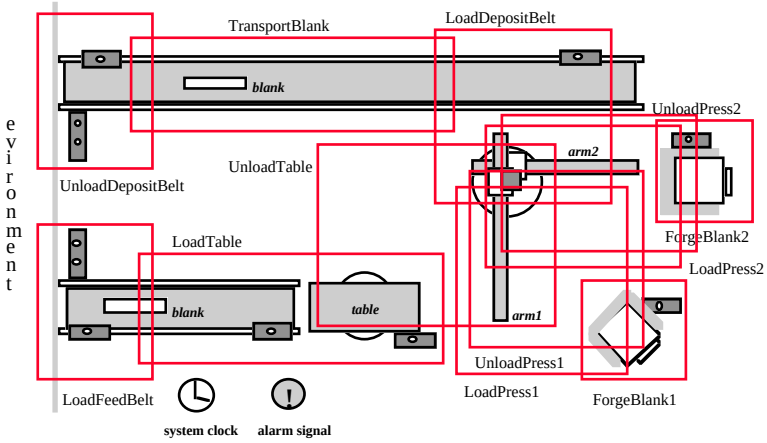

Figure 4 CA actions that control the Production Cell II.

Note that an intersection between CA actions in Figure 4 means that those CA actions cannot be executed in parallel. The mutual exclusion feature of an action together with the proper execution order of the actions guarantees that a blank or a device cannot be involved in more than one action at a time so that neither blanks nor devices can collide. Furthermore, if the actions that the devices participate in are wrongly ordered due to design mistakes or certain faults, then the result will be a safe deadlock.

Each hardware device is associated with a device controller (i.e. an execution thread) which is responsible for dynamically specifying the sequence of actions that the device will participate in. For example, without compromising safety and functionality requirements, the `robot` thread can skip all the CA actions related to one press if the press has failed, and fault tolerance is therefore achieved. Figure 5 shows the interactions (themselves involving nested CA actions) between the participating threads of the `LoadPress1` action. The action has four roles: `Robot`, `Press1`, `RobotSensor`, and `Press1Sensor` and represents the cooperation that arranges for the arm1 of the robot to drop a blank into press1.

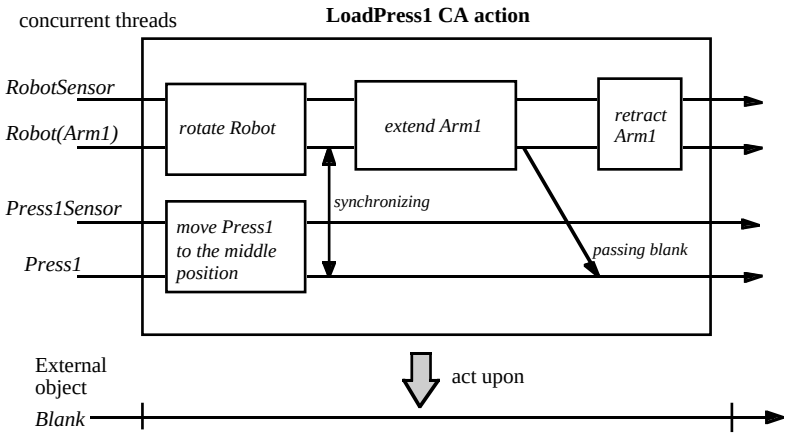

Figure 5 The LoadPress1 CA action.

Our design ensures an action will begin only if its pre-conditions are valid. It is further assumed that if no exception has been raised then its post-conditions should hold (though this could be checked using an acceptance test). For example, the `LoadPress1` action is associated with the following pre- and post-conditions.

pre-conditions	*post-conditions*
arm1 & arm2 extension 0.0	arm1 & arm2 extension 0.0
blank on arm1	no blank on arm1
no blank in the press1	blank in the press1
	robot angle –45°
press1 on the bottom position	press1 on the middle position

If an error occurs, exception handling and error recovery may be only partially successful. Exceptional post-conditions with respect to various given faults are used to specify the exceptional outcomes of an action. In our implementation, the signalling of an exception from any top-level action is actually the process of triggering the cell's alarm signal and sending a message to the user (e.g. [alarm "*blank on arm1*"]). For the `LoadPress1` action the following table shows some of its exceptional post-conditions.

exception to signal	*exceptional post-conditions*
press1 failure & blank on arm1	arm1 & arm2 extension 0.0
	blank on arm1
	no blank in the press1
	robot angle –45°
press1 failure & blank in press1	arm1 & arm2 extension 0.0
	no blank on arm1
	blank in the press1
	robot angle –45°
arm1 magnet failure & blank lost	arm1 & arm 2 extension 0.0
	no blank on arm1
	no blank in the press1
	robot angle –45°
	press1 on the bottom position
other exceptions ...	other conditions ...

For each hardware device, various possible failures are classified and their detection is performed by a combined use of sensor values and the (time-out) stopwatch. The following table shows the failure types of press1 and the means of detecting those failures. (Sensor `S7` indicates whether a blank is in press1, `S8`, `S9`, and `S10` indicate the lower, middle, and upper positions of press1 respectively, actuator `A4` is to move the lower part of press1, and the stopwatch is used to monitor the time while press1 moves up or down.)

failure type	*failure detection*
sensor failure	
S7, S8, S9, S10 (stuck at 0)	values of other sensors
actuator failure	
A4 (motor fails)	values of S7, S8, S10, stopwatch
blank	
stuck or lost	value of S7

It is assumed that any hardware failure and some software-related faults (e.g. those that cause the acceptance test to fail) in the control program could happen at any time during the execution of an action. When a fault is detected within an action, an internal exception is thus raised by a related role and the other roles in the same action will be

informed of the exception and that their execution has to be interrupted. In the meantime, if necessary a diagnostic program is executed to identify the fault type, e.g. deciding whether it is a sensor failure or a press motor failure. When the fault type has been identified and all the roles are ready, they start to handle the fault by using forward or backward error recovery. For example, in the situation that press1 fails and an unforged blank is still on arm1, the `LoadPress1` action uses forward error recovery to produce an exceptional outcome by moving the robot to an appropriate position so that it will be able to put the unforged blank into press2 once it is available, and by switching on the alarm signal to inform the user of the failure of press1. Conversely, if action `LoadPress1` fails to rotate the robot to the intended position, the action will simply use backward error recovery to move the robot back to its initial position and rotate it again.

For each (enclosing or nested) action, various exceptions are defined and an exception graph for resolving concurrent exceptions is declared. For example, the `LoadPress1` action may contain internal exceptions such as `pm_stop` (press motor stops unexpectedly), `rm_stop` (robot rotation motor stops), `pm_nmove` (press motor cannot move), `rm_nmove` (robot rotation motor cannot move), `s_stuck` (sensor(s) stuck at 0), `l_plate` (lost plate), `cs_fault` (control software fault(s)), `l_mes` (lost or corrupted message) and `rt_exc` (run time exceptions such as underflow or overflow). An exception graph for this action is shown in Figure 6, assuming that no more than two exceptions are raised concurrently. For example, when both press and robot rotation motors fail, the exception graph will be searched and the resolving exception `dual_motor_failures` will be raised. Three or more concurrent exceptions as well as other undefined exceptions will not be resolved and will simply lead to the raising of the `universal` exception. (The handler for the `universal` exception is responsible for stopping and leaving the production cell in a pre-defined safe-state and for switching on the cell's alarm signal.)

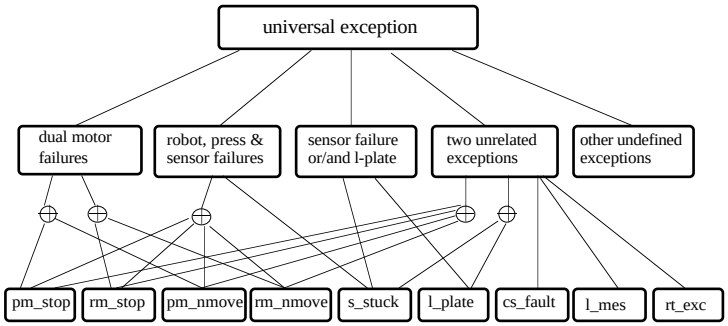

Figure 6 Exception graph for the `LoadPress1` action.

Remarks: Using CA actions in this case study has the very important benefit of encapsulating the safety-critical code and provides a disciplined framework for exception handling.

4 Distributed implementation of CA actions

In the two years since the original publication of the CA action proposal [Xu et al 1995], we have explored various design and implementation issues for CA actions in a wide range of experiments. However, rather than devising a new programming construct for CA actions, we have realised CA actions using a combination of programming conventions backed up by sets of reusable components (e.g. action controller, resolution procedure, etc.). Among our experimental implementations we have implemented a distributed CA action scheme using a centralized manager in Java — this is the one that was used for the case study described in section 3.

CA action support mechanisms provide some or all of the following services: 1) *coordinating service*: activating roles and synchronising role entry and exit; 2) *exception service*: interrupting roles if one of them raises an exception, resolving concurrent exceptions, and propagating exceptions; 3) *information exchange service*: dealing with local and external objects and providing other mechanisms for inter-role cooperation; 4) *fault tolerance service*: providing recovery points, executing the acceptance test, and controlling the adjudicator and diverse sets of roles. In general, such support retains references to external objects, role components and the adjudicator, and saves information about participating threads and nested actions.

In our distributed Java implementation of CA actions each action is represented by a set of components: a centralized CA action manager object, a set of role objects, a set of external objects, and a set of internal objects. The CA action support system is responsible for managing synchronous action entry and exit, global exception handling, error recovery, consistency and atomicity of external and local objects. Figure 7 shows how the components of a CA action are distributed.

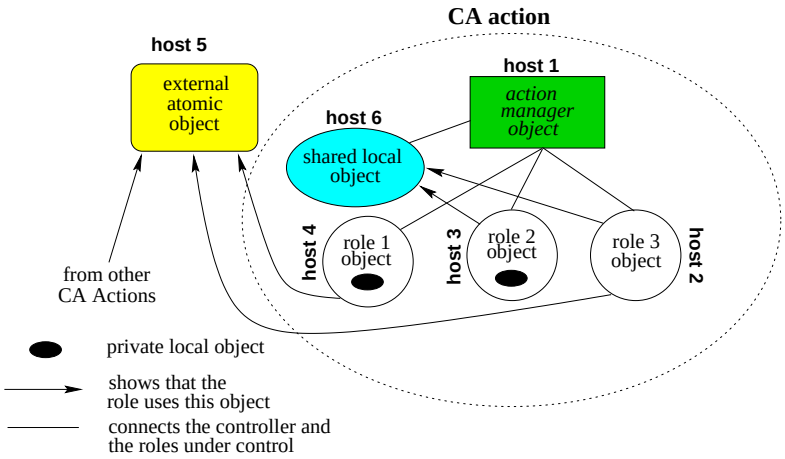

Figure 7 Distribution of action components in the Java implementation.

This Java implementation provides the application programmer with a design framework consisting of four classes that are used to program CA actions: `CAActionManager`, `CAActionRole`, `ExternalObject`, and `SharedLocalObject`.

To program a new CA action, the first step is to define a class that extends the `CAActionManager` class. The `CAActionManager` class contains the operations to deal with entry and exit synchronisation, action exception handling, access to external objects, and access to the roles that compose the action. The `CAActionManager` class therefore provides a basic framework for coordinating the participants in a CA action and copes with the application-independent aspects of error handling. Extensions of `CAActionManager`, such as action `LoadPress1` discussed in section 3, are responsible for declaring the shared local objects used for coordinating the roles within a given CA action and also for specifying the application-dependent aspects of exception handling.

The `CAActionManager` class provides a simple default exception handling mechanism. The `exceptionHandling(Exception e)` method is intended to deal with an exception that cannot be handled locally by the roles; it should be redefined by the application programmer. This is the only method that can be redefined by the programmer who extends the `CAActionManager` class. Taking the `LoadPress1` class as an example, its definition is as follows:

```
class LoadPress1 extends CAActionManager
       Channel robotPressChannel;   // Shared local object used by the roles
       public LoadPress1() {
              robotPressChannel = new Channel(this, “robotPressChannel”);
              SharedLocalObject list[] = {robotPressChannel};
              super(list);
       }
       private void exceptionHandling(Exception e) throws ... {
              // implementation of LoadPress1 exception handling
       }
}
```

This new class has one `Channel` object, `SharedLocalObject` used for internal role communication. The `CAActionManager` is informed of this object so that the roles belonging to this action can gain access to it later via the `getSharedObject()` method.

After a new `CAActionManager` class has been defined, the programmer of the CA action must define the roles that will compose the CA action. This is done by extending the second most important class in the framework: the `CAActionRole` class. Each new class derived from `CAActionRole` contains the main code for one of the roles. For example, action `LoadPress1` is composed of four roles: `Press1`, `Robot`, `Press1Sensor` and `RobotSensor`, so the programmer must create four new classes by extending the `CAActionRole` class. Only objects whose type is derived from `CAActionRole` can participate in a CA action. When deriving a new class from the

`CAActionRole` class, the programmer has to implement at least one method: the private `execute()` method that will contain the main code of that role.

In order to bind role objects to particular instances of CA actions, it is necessary to specify a reference to the corresponding `CAActionManager` object whenever an instance of a role object is created. The `CAActionRole` constructor attaches the role to the `CAActionManager` object using the `attachRole()` method of the `CAActionManager` class.

We have chosen in this implementation to treat individual roles within an action as units of distribution because this allows them to be executed in the locations at which information is produced or consumed. Although several other approaches (e.g. [Wellings & Burns 1996]) view atomic actions as packages (modules, objects etc.), such packages cannot be split into parts and distributed. The general idea of attaching exception handling to roles suits role distribution very well, in spite of the fact that these handlers are controlled by the application-independent action controller (global exception resolution and coordinated action exit are not local decisions). Exception handling local to a role is application-specific and should be performed in its context. Moreover, because of this, roles will deal with external and local object recovery.

The third class in this framework is the `ExternalObject` class. Every new class extended from this class is provided with transactional semantics, i.e. `Begin`, `Commit`, and `Abort` methods, but must provide its own definitions of `CommitState` and `AbortState` methods. In this framework, the transactional semantics of CA actions are achieved by implementing a multi-threaded CA action interface on top of the JavaArjuna system. The Arjuna system is an object-oriented programming system that supports nested atomic transactions for structuring fault-tolerant applications [Parrington et al 1995]. It was originally implemented in C++ but has since been improved and re-written in Java to form JavaArjuna. This system employs nested transactions to control the state changes of objects and to ensure that only consistent state transformations occur on objects despite concurrent access and failures.

Our implementation issues a corresponding transaction on the external objects (e.g. the `metal plate` objects) whenever a new CA action starts, and the transaction ends when the CA action ends. The multi-threaded mechanism in JavaArjuna is used as follows: the thread that first arrives at the CA action interface starts a transaction, and other subsequent threads will then join the same transaction. At the exit, the transaction is ended by removing all the participating threads from the transaction synchronously.

External objects can be recovered following a failure in the CA action by either forward or backward error recovery. Backward recovery is provided in an application-independent fashion, while forward error recovery must be performed by the action roles because such recovery is application-specific and cannot be provided by the underlying CA action support mechanism. External objects are passed to CA actions via input parameters when activating a role (class `CAActionRole`).

The fourth class in this framework is the `SharedLocalObject` class. Local objects are the objects used by the roles in order to exchange internal information. They can be recovered by participant's handlers in an application-specific way. Each shared object is attached (logically) to an action role which has if necessary to recover it as part of action recovery. The object designer can take advantage of any application-specific knowledge to make recovery of shared local objects fast and simple; if, however, such simple recovery is not possible, then these objects can be treated as external ones.

Shared local objects are designed as remote objects in the framework, so there is a chance that attempts will be made to access these objects by adjacent CA actions (e.g. parent, sibling or child actions). To prevent illegal access, we require that the current action identifier be passed as an extra argument to each method invoked on a shared local object from within a CA action (class `CAActionRole`). Then it is possible to perform an internal check because shared local objects are bound to particular instances of CA actions at object creation time (class `CAActionManager`).

*Remarks:*The application of this Java framework to a variety of case studies involving CA actions has demonstrated that the framework is indeed reusable and can achieve a useful separation between application-independent and application-dependent aspects of exception handling and error recovery.

5 Conclusions and further work

Some two years on from the original publication of the CA action concept [Xu et al 1995], we feel that we now have a much fuller understanding of CA actions and the design issues involved in their implementation. The formalization of the CA action concept has been particularly helpful in clarifying a number of confusing aspects and identifying many important properties of the concept. Our Java-based implementation has greatly facilitated fast prototyping of experimental systems and provided a well-structured way of dealing with a variety of possibly very complicated fault situations. The implementation framework described here has been used for a variety of case studies [Zorzo et al 1997] and has proved its general utility. Other implementations in Ada (e.g. [Xu et al 1997]) have enabled us to investigate more distributed

implementations of CA actions — these have included the development of a much improved version of the exception resolution algorithm. The convenient way in which CA actions could be used to embody all the safety-related aspects of our various Production Cell case studies was an advantage that we had not anticipated. This type of design was one that we arrived at through consideration of a particular application (or rather group of related applications) — we now realize that a more methodical means of arriving at such designs is possible, as well as being highly desirable. Work on this topic is described in [DeLemos & Romanovsky 1997]. Ideally, of course, such an approach would be allied to formal means of system validation — something that could we believe take advantage of the temporal logic model of CA actions that we have presented here.

Clearly, much remains to be done and we therefore plan to continue our research into CA actions as a major activity in the ESPRIT Long Term Research Project on "Design for Validation" (DeVa) together with a number of our partners on that project. For example, it is necessary to develop a more detailed linguistic formulation for CA actions even though, for practical reasons, our implementation experiments are likely to remain based on the use of existing languages buttressed by library routines and programming conventions. (We also hope to make use of recent work in DeVa on object-oriented reflection [Stroud & Wu 1995][Fabre & Perennou 1996], so as to achieve a greater logical separation of code involved in various aspects of the CA mechanism from application code.)

Last but not least, we would also like to extend the CA action concept so as to make it suitable for use in real-time systems and have already started our investigation [Romanovsky et al 1997]. This describes an extended mechanism for handling and resolving real-time exceptions that deals with issues such as an early or late start of and an early or late end of an action, arranging that enough time remains for performing appropriate error recovery, and resolving multiple ordinary and real-time exceptions. In fact, it is possible to draw an interesting parallel between the use of the present CA action scheme for structuring the behaviour of a system in the value domain, and the use of temporal firewalls [Kopetz & Nossal 1997] for structuring system behaviour in the time domain. We therefore intend to investigate the ways in which these two concepts might complement each other in cooperation with our colleagues at the Technical University of Vienna and the University of York.

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 our colleagues, especially John Fitzgerald.

References

[Banatre et al 1986] J.P. Banatre, M. Banatre and F. Ployette, "The Concept of Multi-Functions: A General Structuring Tool for Distributed Operating Systems," in *Proc. 6th Distributed Computing Systems Conf.*, pp.478-485, 1986.

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

[Davies 1973] C.T. Davies, "Recovery Semantics for a DB/DC system," in *Proc. ACM Nat. Conf*, pp.136-141, 1973.

[Davies 1978] C.T. Davies, "Data Processing Spheres of Control," *IBM System Journal*, vol.17, no.2, pp.179-198, 1978.

[DeLemos & Romanovsky 1997] R. DeLomos and A. Romanovsky, "Coordinated Atomic Actions in Modelling Objects Cooperation," Technical Report, Dept. of Computer Science, Univ. of Newcastle upon Tyne, no.620, 1997.

[Elmagarmid 1993] A.K. Elmagarmid (ed.). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publ., 1993.

[Fabre & Perennou 1996] J.C. Fabre and T. Perennou, "FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications," in *Proc. 2nd European Dependable Computing Conference* (EDCC2), Taormina, 1996.

[Francez & Forman 1996] N. Francez and I.R. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*, Addison-Wesley, 1996.

[Gray 1978] J.N. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer *et al* eds., pp.393-481, Springer-Verlag, 1978.

[Gregory & Knight 1989] S.T. Gregory and J.C. Knight, "On the Provision of Backward Error Recovery in Production Programming Languages," in *Proc. 19th Int. Symp. on Fault-Tolerant Computing*, pp.506-511, Chicago, 1989.

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

[Jung & Smolka 1996] Y-J. Joung and S. A. Smolka, "A Comprehensive Study of the Complexity of Multiparty Interaction," *Journal of the ACM*, vol.43, no.1, pp.75-115, 1996.

[Kim 1982] K.H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Trans. Soft. Eng.*, vol.SE-8, no.3, pp.189-197, 1982.

[Kopetz & Nossal 1997] H. Kopetz and R. Nossal, "Temporal Firewalls in Large Distributed Real-Time Systems," Technical Report, Institut fur Technische Informatik, Technische Universitat Wien, no.7/97, 1997.

[Lamport 1994] L. Lamport, "The Temporal Logic of Actions," *TOPLAS*, vol.16, no.3, pp.872-923, 1994.

[Lewerentz & Lindner 1995] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell"*, LNCS-891, Springer, Jan. 1995.

[Liskov 1988] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no.3, pp.300-312, 1988.

[Lotzbeyer & Muhlfeld 1996] A. Lotzbeyer and R. Muhlfeld, "Task Description of a Fault-Tolerant Production Cell," Internal FZI Technical Report, Karlsruhe, (ftp://ftp.fzi.de/pub/PROST/projects/korsys/task_descr_flex_2_2.ps.gz), 1996.

[Lyu 1995] M. Lyu (ed.).*Software Fault Tolerance*, (Trends in Software series), John Wiley & Sons, 1995.

[Manna & Pnueli 1991] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, Springer, 1991.

[Moss 1981] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," *Ph.D. Thesis (Tech. Report 260)*, MIT Lab. for Computer Science, Cambridge, MA., 1981.

[Parrington et al 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems Journal*, vol.8, no.3, 1995.

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

[Romanovsky et al 1996] A. Romanovsky, J. Xu and B. Randell, "Exception Handling and Resolution in Distributed Object-Oriented Systems," in *Proc. 16th IEEE Intl. Conf. Distributed Computing Systems*, pp.545-552, Hong Kong, 1996.

[Romanovsky et al 1997] A. Romanovsky, J. Xu and B. Randell, "Exception Handling in Object-Oriented Real-Time Distributed Systems," Technical Report, Dept. of Computer Science, Univ. of Newcastle upon Tyne, no.618, 1997.

[Stroud & Wu 1995] R.J. Stroud and Z. Wu, "Using Metaobject Protocols to Implement Atomic Data Types," in *Proc. ECOOP'95*, LCNS 952, pp. 165-189, 1995.

[Wellings & Burns 1996] A.J. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95", Tech. Report, Dept. of Computer Science, Univ. of York, no.YCS-263, 1996.

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

[Xu et al 1997] J. Xu, A. Romanovsky and B. Randell, "Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation," Technical Report, Dept. of Computer Science, Univ. of Newcastle upon Tyne, no.612, 1997.

[Zorzo et al 1997] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud and I. Welch, "Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems," *OOPSLA'97 Workshop on Dependable Distributed Object Systems*, 1997.