A Distributed Object-Oriented Framework for Dependable Multiparty Interactions

A. F. Zorzo
Faculdade de Informática
Pontifícia Universidade Católica do RS
90619-900 Porto Alegre - RS - Brazil
zorzo@inf.pucrs.br

R. J. Stroud
Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne - UK - NE1 7RU
r.j.stroud@ncl.ac.uk

Abstract

In programming distributed object-oriented systems, there are several approaches for achieving binary interactions in a multiprocess environment. Usually these approaches take care only of synchronisation or communication. In this paper we describe a way of designing and implementing a more general concept: *multiparty interactions*. In a multiparty interaction, several parties (objects or processes) somehow "come together" to produce an intermediate and temporary combined state, use this state to execute some activity, and then leave this interaction and continue their normal execution. The concept of multiparty interactions has been investigated by several researchers, but to the best of our knowledge none have considered how failures in one or more participants of the multiparty interaction can be dealt with. In this paper, we propose a general scheme for constructing *dependable multiparty interactions* in a distributed object-oriented system, and describe its implementation in Java. In particular, we extend the notion of multiparty interaction to include facilities for handling exceptions. To show how our scheme can be used, we use our framework to build an abstraction mechanism that supports cooperative and competitive concurrency in distributed systems. This mechanism is then applied to program a system in which multiparty interactions are more than simple synchronisations or communications.

Keywords: Distributed Object-Oriented Systems, Multiparty Interactions, Concurrent Exception Handling, Fault Tolerance, Coordinated Atomic Actions

1 Introduction

With the expansion of computer networks, activities involving computer communication are becoming more and more distributed. Such distribution can include processing, control, data, network management, and security [1]. Although distribution can improve the reliability of a system by replicating components, sometimes an increase in distribution can introduce some undesirable faults. To reduce the risks of introducing faults when distributing applications, it is important that distributed systems are implemented in an organized way. One way of organizing distributed object-oriented applications is to model operations that involve more than one object as separate actions that coordinate the necessary interactions between the participant objects. This has the advantage of making the individual objects easy to program and more reusable.

A mechanism that encloses multiple parties (objects or processes) executing a set of activities together is called a *multiparty interaction*. In a multiparty interaction, several parties somehow "come together" to produce an intermediate and temporary combined state, use this state to execute some activity, and then leave the interaction and continue their normal execution.

There has been a lot of work in the past years on multiparty interaction, but most of it has been concerned with synchronisation, or handshaking, between parties rather than the enclosure of several activities executed in parallel by the interaction participants. Specification languages like CSP, LOTOS, or programming languages like Ada only deal with synchronisation between processes. The programmer is responsible for ensuring that the processes involved in a cooperative activity do not interfere with, or suffer interference from, other processes not involved in the activity.

Properties for multiparty interaction have been described in the literature [2] [3]. The properties listed are related to:

- synchronisation upon entry of participants of the interaction;
- using a guard to check the preconditions to execute the interaction, hence the need for having synchronisation upon entry;
- an assertion after the interaction has finished to guarantee that a set of post-conditions has been satisfied by the execution of the interaction; and,
- atomicity of external data to ensure that intermediate results are not passed to other processes before the interaction finishes.

Usually, there is also a discussion about the use of temporary state by the participants of the interaction; the way the body of the interaction is split (usually if the body of the interaction is split in more than one part, each of these parts is called a *role* of the interaction); or the way the number of participants is specified in the interaction, i.e. fixed or variable.

However, to the best of our knowledge none of this work has discussed the provision of features that would facilitate

the design of multiparty interactions that are expected to cope with faults - whether in the environment that the computer system has to deal with, in the operation of the underlying computer hardware or software, or in the design of the processes that are involved in the interaction.

Because faults are expected to occur rarely during the execution of a program, programmers usually refer to them as *exceptions* [4]. To handle the exceptions that may occur during the execution of a program, an exception mechanism is usually provided. This mechanism allows a programmer to describe an exceptional flow that replaces the normal flow of a program whenever an exception is detected in that program.

This paper concerns the concept of a *dependable multiparty interaction* (DMI) - a term that we use for a multiparty interaction that provides facilities for exception handling, in particular including means of:

- *Handling Concurrent Exceptions*: when an exception occurs in one of the bodies of a participant, if it is not dealt with by that participant, the exception must be propagated to all participants [5] [6]. A DMI must provide a way of dealing with exceptions that can be raised by one or more participants. If several different exceptions are raised concurrently, then the dependable multiparty interaction mechanism uses a process of exception resolution to decide upon a common exception that will be raised in all the participants.
- *Synchronisation Upon Exit*: all participants have to wait until the whole interaction finishes, i.e. a participant can only leave the interaction when all of them have finished their roles and the external objects are in a consistent state. This property guarantees that if something goes wrong in the activity executed by one of the participants, then all participants can try to recover from possible faults.

In view of our interest in dependability, and in particular fault tolerance, we adopt the use of pre and post-conditions, which are checked at run-time. Regarding the remaining alternatives listed earlier for multiparty interactions, we have made the following design choices for DMIs:

- The number of processes involved in a given DMI should be fixed, although the particular processes involved should be able to vary.
- The processes should synchronise their entry to and exit from the DMI.
- The DMI mechanism should ensure that as viewed from outside the DMI, its system state should change atomically, though inside the DMI intermediate states will be visible.
- The way the underlying system executes a DMI can be synchronous or asynchronous.

The key idea for handling exceptions is to build DMIs out of unreliable multiparty interactions by chaining them together, where each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. In Figure 1 we show how a basic multiparty interaction and exception handling multiparty interactions are chained together to handle possible exceptions

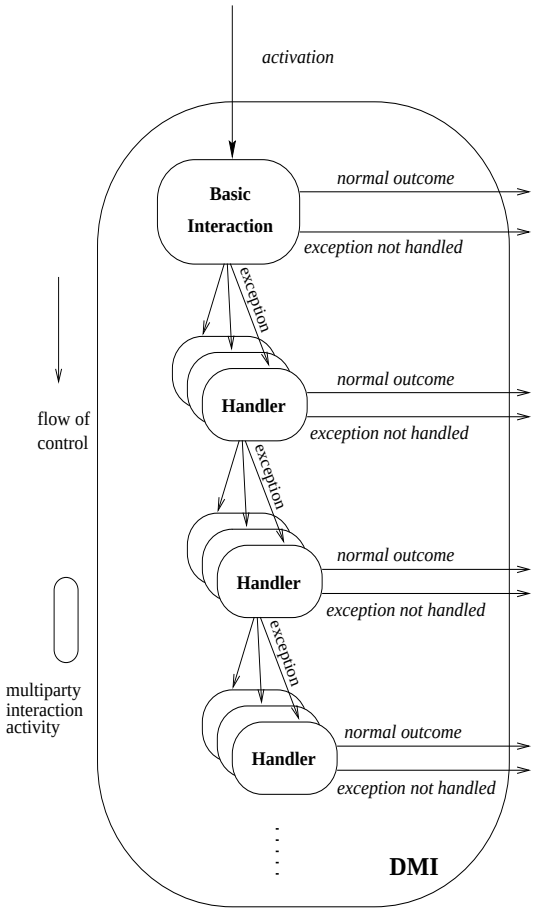

Figure 1: Dependable Multiparty Interaction

that are raised during the execution of a DMI. As shown in the figure, the basic multiparty interaction can terminate normally, raise an exception that is handled by an exception handling multiparty interaction, or raise an exception that is not handled in the DMI. If the basic multiparty interaction terminates normally, the control flow is passed to the callers of the DMI. If an exception is raised, then there are two possible execution paths to be followed: i) if there is an exception handling multiparty interaction to handle this exception, then it is activated by all processes in the DMI; ii) if there is no exception handling multiparty interaction to handle the raised exception, then this exception is passed to the callers. Section 3 contains a further discussion of exception handling in DMIs.

In this paper we show how DMIs can be implemented in distributed object-oriented environments. We have implemented a complete object-oriented API that provides a programmer with the necessary tools for implementing a DMI. Our API can be thought of as a layer implemented on top of an object-oriented transaction system that adds support for coordinated entry and exit from multi-threaded transactions. Furthermore, out API provides a disciplined exception handling mechanism that can deal with exceptions that are raised concurrently in different threads of the transaction. Dealing with these issues correctly in concurrent/distributed systems is sufficiently complex to require

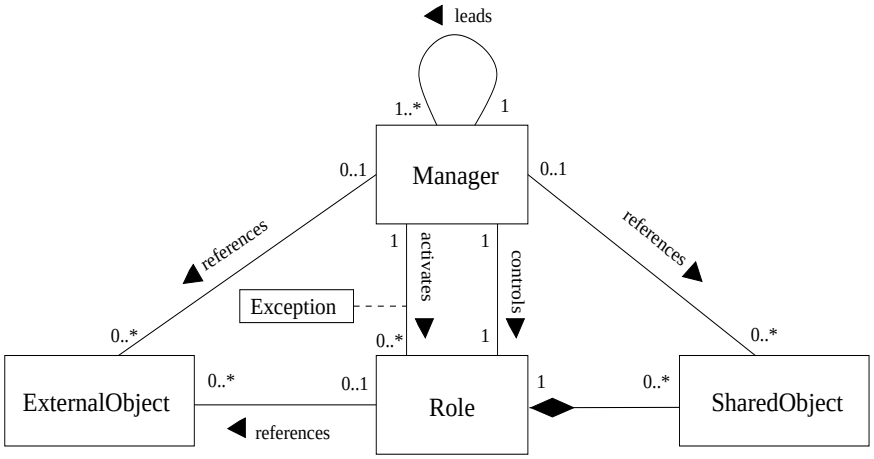

Figure 2: Class Diagram (UML)

the support of an additional API that complements the existing transactional paradigm. We have used our API as a test-bed for exploring language mechanisms for multiparty interactions, and have applied our API in the development of a mechanism for developing cooperative and competitive applications. The implementation of this mechanism has been used for building a controlling system to drive a fault-tolerant production cell.

2 Framework Description

In Section 1 we showed how multiparty interactions could be linked together to implement a DMI. In this section we will describe our generic framework for implementing DMIs. Our framework is composed of four types of distributed objects: *roles* that host the set of operations for the participants of the interaction; *managers* that are responsible for keeping track of the components of the interaction, managing synchronisation of participants, testing the pre and post-condition for the interaction, and deciding upon which exception is to be handled by all the participants of the interaction; *shared objects* that are used for cooperation between the participants; and *external objects* that carry the state of the system in and out of the interaction. Each of the objects described can potentially be distributed in a different host. Each DMI is represented by several sets of these remote objects: one set for the interaction when there is no failures, i.e. *basic interaction*, and several sets for dealing with exceptions that may be raised during the execution of the interaction (either during the basic interaction or during an *exception handling interaction*).

Figure 2 shows the UML class diagram for our API. The figure shows that each role is associated with only one manager, and each manager controls only one role. Managers are connected to a special manager, which is called the leader manager. The leader manager is the only manager that is associated with shared objects. Shared objects are created by roles, which eventually export them to the leader manager and to the other roles. External objects are associated with both managers and roles. Managers will keep track of these objects for possible recovery process, while the roles will use them. Shared objects are created by the roles and are used for communication between the roles. The leader manager has a reference to these objects in order to inform the roles about their existence. As shown in the figure, we do not have a class to represent either a multiparty interaction or a DMI in our framework. A multiparty interaction consists of a set of managers linked together via a leader manager (represented by the **leads** association). Multiparty interactions are connected together in order to create a DMI via the **activates** association. A set of managers, in a multiparty interaction, will activate the appropriate set of roles for dealing with the raised exception. This new set of roles will be controlled by a different set of managers.

To program a new DMI using our framework, the first step is to define a new class that extends the Role class for each party in the interaction. The extended Role class has to redefine at least one method: the body method. This method will contain the set of operations that will be executed by the participant that activates the role. Upon creation each Role has to be informed about the manager that will be managing this role. A manager that 'controls' a Role object is an instance of the Manager class. The Manager class provides a basis for coordinating the participants in a multiparty interaction. We decided to separate the manager from the roles in order to allow the application code of the role to be distributed to the host where the application is created. This strategy will help in avoiding the overloading of one host with the control of the DMI and the application code. Furthermore, in the case of the host of the application role crashing, the manager can still run and recover, together with the other managers, from this crash.

The managers of all roles will compose the controlling body of the interaction. Each manager upon creation is informed of which manager will act as the *leader* in the interaction. The leader is responsible for controlling protocols for synchronisation between managers, for the exception resolution algorithm, and for keeping information about the shared objects. Every manager is a potential leader in our framework, avoiding a possible single failure point, if the host of the leader crashes.

We have implemented the framework described here using the Java language and its RMI ORB to distribute the objects of a DMI.

2.1 Manager

The Manager class is the major class in our framework. Each role has to be managed by a different manager object. When instantiating a Manager object, the manager has to be informed of its name, the name of the interaction, the leader of the interaction (a manager without a leader is its own leader), and a list of exceptions that will be treated by the manager. Each exception in the list is associated with a role from an exception handling interaction. This new role, which is controlled by a different manager, will be called to treat that exception when it is raised in all roles of the interaction. The list of exceptions attached to the managers is the link between the multiparty interactions of a DMI. Exceptions that are raised in a multiparty interactions whose managers do not treat exceptions are propagated to the enclosing context. The following Java commands show how to instantiate a set of managers for an interaction:

```
Manager mgr1=new Manager("mgr1","DMIname",eh1,null);
Manager mgr2=new Manager("mgr2","DMIname",eh2,mgr1);
```

There are four ways of creating a manager object: creating a leader manager object that handles exceptions; creating a leader manager object that does not handle exceptions; creating a manager object that handles exceptions and is led by a leader; or, creating a manager object that does not handle exceptions and is led by a leader. In the above example, two managers were created for the same DMI with mgr1 acting as the leader of the DMI and mgr2 being led by mgr1). The eh1 hashtable contains the list of exceptions that are treated by mgr1 and the roles that are activated in the case of one of the exceptions that are in the list being raised. All hashtables of managers from the same DMI must contain the same list of exceptions to be handled. If the hashtables do not contain the same list of exceptions, then an exception is raised at creation time.

Upon activation, each manager executes the sequence of operations in the Java code show below as its main execution body. The first activity a manager executes is to synchronise itself with all the other managers in the interaction by calling the synchroniseBegin method. This method blocks until the leader determines that all the managers have synchronised and the multiparty interaction is ready to begin. Once the synchroniseBegin returns, the manager checks if the role precondition is valid. The preCondition method receives all the objects that will be passed to the role managed by this manager as parameters. If the precondition is not satisfied, then a PreConditionException is thrown. Otherwise, the manager executes the body of the role is is controlling by calling the bodyExecute method of the Role object. After the role has finished its execution, the manager synchronises with all the other managers before testing its post-condition. If the post-condition is passed (accepted), then the manager synchronises with all the other managers and the interaction is finished. This sequence of steps is executed if all the activities of all the participants of an interaction finish without raising any exception. If an exception is raised during the execution of a role, the manager will catch this exception in the catch(Exception e) block. In this situation, the manager calls an exception resolution algorithm, and after receiving back the exception it has to handle, the manager activates the role in the exception handling interaction that deals with this exception. If there is no exception handling interaction for the exception it has to handle, then this same exception is thrown by all managers in the DMI to the callers of the roles.

```
try {
    synchroniseBegin(); // synchronise upon entry
    if (!roleManaged.preCondition(listOfParameters))
        throw new PreConditionException();
    // execute the role
    roleManaged.bodyExecute(this,listOfParameters);
    // wait everyone before checking post-conditions
    synchroniseEnd();
    if (!roleManaged.postCondition(listOfParameters))
        throw new PostConditionException();
    synchroniseEnd(); // really exit synchronously
} catch (Exception e) {
   // exception resolution part, call exception
   // resolution algorithm, and activate role
   // associated with the exception (if there is one)
}
```

By default, the Manager class provides a built-in exception resolution mechanism based on [6]. This mechanism works as follows. When a role raises an exception, its corresponding manager is notified of that exception. The manager then informs the leader which interrupts all roles that have not raised an exception. After all roles have been interrupted or have notified the leader manager of an exception (exceptions can be raised concurrently), an exception resolution algorithm is executed by the leader. This algorithm tries to find a common ancestor[1] exception from all raised exceptions. When such an exception is found, the leader informs all managers about that exception and an exception handling interaction is activated (in the same way a complete new set of managers and roles would be activated) using the exception handlers list which the manager was initialized with. If there is no interaction handler for that exception, a handler for the highest level exception (Exception class) is tried. If there is no handler even for Exception, then the exception is passed to the enclosing context.

In the event of one of the managers or one of the roles crashing, the managers communicate with each other and decide to raise a CrashedManagerException or a CrashedRoleException exception. If the manager that has crashed was the leader, then a new leader may be chosen by the managers that are still running. If a CrashedManagerException or a CrashedRoleException is raised, then these exceptions are propagated to the callers of the DMI.

If the user of the framework wants to provide its own algorithm for deciding which exception is to be handled by all the roles, then the Manager class can be extended and a method called exceptionResolution must be provided. This method must return an exception that is derived from the Exception class. A list containing the exceptions that were

[1] A common exception of which all raised exceptions are subtypes. In the worst case scenario, the common exception is Exception.

raised by the roles is passed to the new exception resolution method.

2.2 Role

After a new Manager object has been created, the programmer of the multiparty interaction has to create a role object that will be controlled by that manager. This role object has to be an instance of a new role class derived from the Role class provided by the framework. Each new class derived from Role contains the main code for one of the roles that compose the multiparty interaction. Only objects whose type is derived from Role can belong to a multiparty interaction. When deriving a new class from the Role class, the programmer has to implement at least one method: the private body method that will contain the main code of that role. This method does not return any value. It receives a list of external objects as parameter (see Java code below). If an exception is raised during the execution the the body method, then that exception can be handled internally, if it does not affect other roles, or it can be thrown to the manager of that role that will resolve which exception handling interaction will deal with that exception (or some common exception if more than one role raises an exception simultaneously).

```
public class RoleName extends Role {
  SharedObject so; // shared object
  public RoleName(Manager mgr, String roleName) {
    super (mgr, roleName); // set role with name & manager
    so = new SharedObject(); // creates a shared object
    mgr.sharedObject("soName", so); // export shared object
  }
  protected void body(Transactional list[]) throws Exception
  {
    // code for the body of the RoleName
  }
}
```

Extensions of the Role class are also responsible for declaring the shared local objects used for coordinating the roles within a particular interaction, and for checking part of the pre and post-conditions of the interaction. After the shared objects have been created, the role must inform its manager about those objects using the sharedObject method. This will export the shared objects, making it possible for other roles in this interaction to use them later. The pre and post-conditions of an interaction can be checked in a distributed way; each role checks part of the conditions, or one role could be delegated to check the whole pre and post-condition of the interaction. The delegation could be achieved by using shared objects between the roles. The methods used to test pre and post-conditions are called preCondition and postCondition. These methods may be redefined in the new role class. The Java code above is an example which shows how the Role class can be extended, and how shared objects can be created and exported by this new role.

Roles are distributed objects in our framework and provide a user with the following public methods: execute and bodyExecute. When the execute method is called, the role passes control to its manager which will execute the bodyExecute method of this role. The bodyExecute method takes the manager descriptor as its parameter. This descriptor is checked against the manager descriptor that the role was created with. This guarantees that only the manager of this role can execute its main body (body private method of the new role class).

2.3 External, Shared and Local Objects

The third class in our framework is the ExternalObject class. This class implements an interface called Transactional. The Manager class expects objects from a class that implements the Transactional interface. The Transactional interface defines the following public methods: begin, commit, and abort. The ExternalObject class provides a basic implementation for the Transactional interface. Any new class extended from ExternalObject class is provided with this basic implementation of the Transactional interface but must provide its own definitions of commitState and abortState methods. We decided to implement our own simple transactional system, but this could easily be replaced by an existing one like the Arjuna system[7]. External objects are passed to the multiparty interaction via input parameters when activating a role.

The fourth class in our framework is the SharedLocalObject class. Shared local objects are the objects used by roles in order to exchange information with each other. These objects are used only inside the interaction and their values are discarded after the interaction has finished, either when the interaction terminates normally, or when an exception is raised by one of the roles. The following Java code shows how a role can get a reference to a shared object that was exported by another role (see Java code in Section 2.2 for the exporting of a shared object). In the body method of the role, the manager of that role is asked about an object called "soName" using the getSharedObject method. If the object exists then a reference to that object is returned, otherwise a null is returned.

```
SharedObject so = (SharedObject)
                  mgr.getSharedObject("soName");
```

Shared local objects are remote objects in our framework, so there is a chance that these objects can be accessed by adjacent interactions (e.g. parent or sibling interactions). However, even though shared local objects can be seen by other interactions, only threads that are executing the interaction (or roles belonging to that interaction) should be able to access them. To ensure these semantics, every time a thread starts to execute an interaction, the managers inform the shared local objects about the threads that are authorized to access them. It is possible to perform an internal check because shared local objects are bound to particular instances of interactions at object creation time.

The roles in a multiparty interaction may also use private local objects. These objects are not used concurrently, so it is the responsibility of the role to take care of them. If any kind of recovery is necessary they have to be recovered by their owner. If a role that created local objects cannot recover them, then an exception should be raised.

3 Building Multiparty Interaction Abstractions

The framework we have described in Section 2 makes it possible to construct DMIs from multiparty interactions in a very flexible way by gluing them together as exception handlers. DMIs built in this way can be applied directly to implement reliable systems (see [8]). However, sometimes it is necessary to adopt a more restricted form of exception handling in which case our framework can be used to implement a more constrained programming abstraction with stronger exception handling semantics. CA actions [9, 10] are an example of such an abstraction. In this section, we describe the exception handling semantics of CA actions and explain how a CA action mechanism can be implemented using our framework. However, it would equally be possible to use our framework to implement other abstractions such as multiway rendezvous [11] or DisCo actions [12].

The Coordinated Atomic (CA) action is a mechanism for coordinating multi-threaded interactions and ensuring consistent access to objects in the presence of concurrency and potential faults. It can be regarded as providing a programming discipline for nested multi-threaded transactions that in addition provides very general exception handling facilities. The scheme is directly suitable for handling situations in which hardware and software faults have not been masked by the underlying transaction mechanism but have instead been reported to the application level to deal with, and/or in which there are application-level failure situations that have to be responded to.

A CA action involves multiple roles and the roles participating in an action must agree about its outcome. There are four possible kinds of outcome: normal, exceptional, abort and failure. A CA action terminates normally if it is able to satisfy its post-conditions. If a CA action does not terminate normally, then each role must *signal* an exception to indicate the outcome. The roles should agree about the outcome so each role should signal the same exception. If an exception is *raised*[2] during the execution of a CA action, this triggers a process of exception handling. Depending on how successfully the CA action can recover from the exception, it may still terminate normally or else exceptionally. If error recovery is not possible, the CA action may attempt to roll-back the state of external objects and signal abort. If the roll-back is unsuccessful, then the CA action must signal failure.

If a CA action terminates exceptionally, the corresponding exception is raised in the enclosing context. CA actions can be nested and this means that an action which terminates by signalling an exception is effectively passing on the responsibility for exception handling to the enclosing CA action.

Figure 3 shows how the CA action mechanism is built using multiparty interactions. In the figure, rounded boxes represent multiparty interactions, arrows represent the flow of control, and the big rounded box represents the CA action mechanism. As we can see in the figure, CA actions are implemented with three types of multiparty interactions: one interaction for the normal execution of the CA action; a set of multiparty interaction handlers for dealing with the exceptions that may be raised during the normal execution of the CA action; and a third interaction that deals with the rolling back process of the CA action.

[2] We use the term *raise* for exceptions that are dealt with inside the CA action, and *signal* for exceptions that are raised in the enclosing CA action.

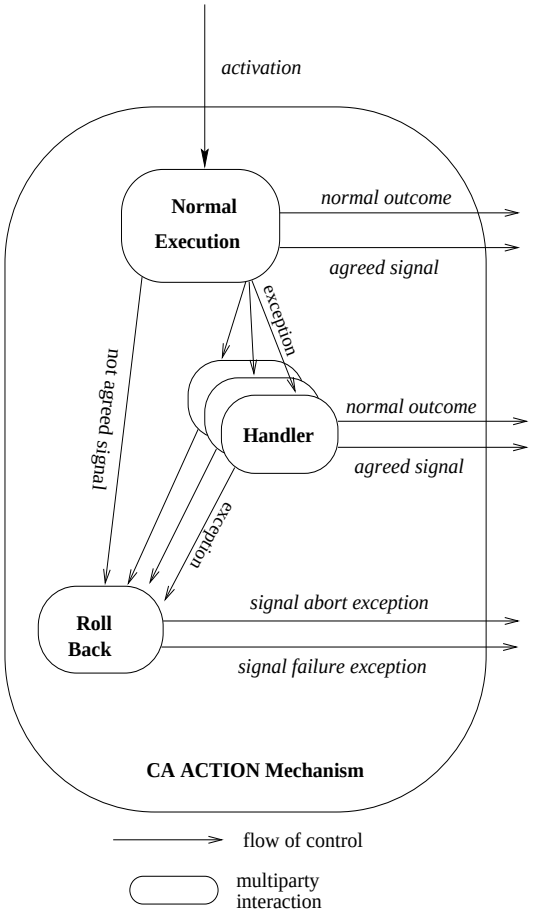

Figure 3: CA Action Mechanism

The way the exceptions shown in Figure 3 are raised by DMIs and handled by the CA action mechanism are driven by the following rules:

1. Each role in a CA action may terminate normally or exceptionally by signalling an exception to the enclosing CA action. The roles should agree about the outcome of the action. However, a role may also signal an abort exception or a failure exception to indicate that the action should abort or fail (see 7). In Figure 3 this rule is represented by the *normal outcome* and *agreed signal* lines.

2. If the roles do not agree about the outcome, then the underlying CA action support mechanism attempts to abort the action by undoing its effects on external objects. This rule is represented in the figure by the *not agreed signal* line from the normal execution interaction to the roll back interaction.

3. If the abort is successful, then an abort exception is signalled to the enclosing CA action; otherwise, a failure exception is signalled. See *signal abort exception* and *signal failure exception* lines in the figure.

```
// set of managers for the roll-back interaction
Manager mgrRB1 =new Manager("mgr1-RB", "CA DMI", null, null);
Manager mgrRB2 =new Manager("mgr2-RB", "CA DMI", null, mgrRB1);
Manager mgrRB3 =new Manager("mgr3-RB", "CA DMI", null, mgrRB1);
// declaration of roles for the roll-back interaction
// and the rb1, rb2, rb3 hashtables
:
// set of managers to deal with E1 exception
Manager mgrE11 = new Manager("mgr1-E1", "CA DMI", rb1, null);
Manager mgrE12 = new Manager("mgr2-E1", "CA DMI", rb2, mgrE11);
Manager mgrE13 = new Manager("mgr3-E1", "CA DMI", rb3, mgrE11);
// declaration of roles for the handler interaction
// for E1 and the e11, e12, e13 hashtables
:
// set of managers for the normal execution
// of the DMI
Manager mgr1 = new Manager("mgr1", "CA DMI", e11, null);
Manager mgr2 = new Manager("mgr2", "CA DMI", e12, mgr1);
Manager mgr3 = new Manager("mgr3", "CA DMI", e13, mgr1);
```

4. If an exception is raised during the normal execution of a CA action, then control is passed to the corresponding exception handler for each role. If two or more exceptions are raised concurrently, then a process of exception resolution must take place first.

5. If an exception is raised during the execution of an exception handler, then the underlying CA action support mechanism will attempt to abort the action (see 3) or else signal a failure exception to the enclosing action.

6. Once exception handling begins within a CA action, it is not possible to resume normal execution of the CA action but it is possible for the exception handlers to terminate normally or exceptionally, depending on the extent to which error recovery is successful. However, all roles must still agree about the outcome (see 2).

7. A role may signal abort or failure at any time to indicate that error recovery is not possible and the action must abort or fail. For the purposes of determining the outcome, failure takes precedence over abort which takes precedence over every other exception that can be raised internally.

8. For a given action, exception handlers can only be provided for exceptions that are raised internally within that action. Exceptions that are signalled by an action are handled at the level of the enclosing action. Thus, an action cannot provide an exception handler for its own abort or failure exceptions.

9. If an action terminates by signalling an exception to its enclosing action, then this triggers the process of exception handling in that action (see 5).

The above semantics for handling exceptions has been applied to our API in order to build CA actions (no change to the API has been made). In the Java code on the top of this page we show how three sets of managers would be created and linked together to respect the CA action semantics. The roles controlled by the three first managers (mgr1-RB, mgr2-RB and mgr3-RB) are responsible for bringing the external objects to the state they had before the CA action had started. Notice that there is no exception handling list for roles of these managers. Any exception that is raised during the rolling back interaction will be mapped to a *failure* exception and passed to the enclosing context. The second set of managers (mgr1-E1, mgr2-E1 and mgr3-E1) control the roles that deal with an exception E1 that may be raised during the *normal execution* interaction of a CA action. If any exception is raised during the interaction that deals with exception E1, then the roll-back interaction will be activated (rb1, rb2 and rb3 lists are passed to the managers of the handling interaction for E1, and these lists contain links to the roles of the roll-back interaction). The third set of managers (mgr1, mgr2 and mgr3) control the set of roles responsible for the normal execution of the CA action. If exception E1 is raised during the normal execution interaction, then the exception handling interaction for E1 is activated. If any other exception is raised during the normal execution interaction, then the roll-back interaction is activated (the e11, e12 and e13 lists contain links to the roles of the handling interaction for E1, and the roles of the roll-back interaction).

A set of CA actions was built using the above approach and then used to program a controlling software for the fault-tolerant production cell presented in the next section.

4 Building Safety-Critical Systems

Industrial installations that have several pieces of equipment controlled by software systems require interactions between this equipment to be executed in a safe and fault-tolerant way. In this section we show how to apply the CA action mechanism implemented using our API to a fault-tolerant production cell case study. The Fault-Tolerant (FT) Pro-

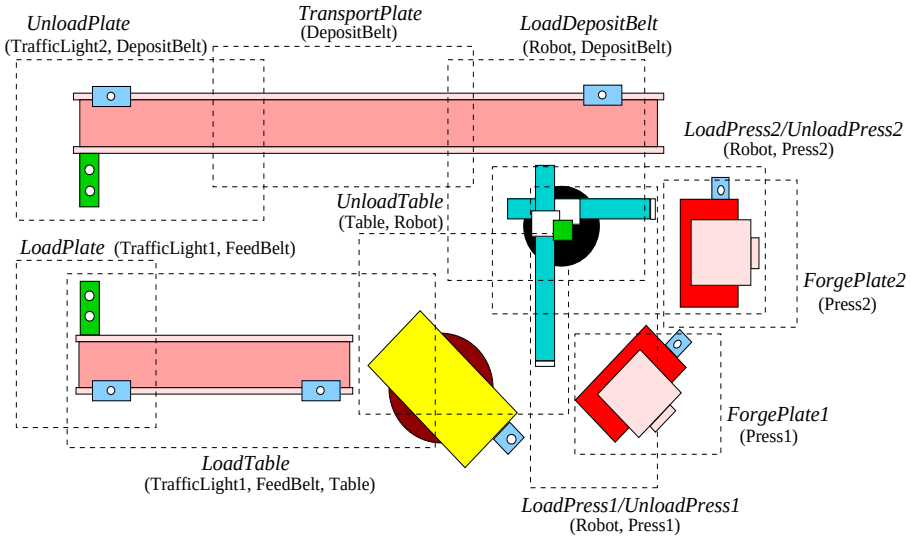

Figure 4: Fault-Tolerant Production Cell

duction Cell [13] we use is an extension of a production cell case study described in [14], which is a model based on an actual industrial installation in a metal-processing plant in Karlsruhe, Germany. It was developed in the Forschungszentrum Informatik (FZI). The FT Production Cell is composed 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 arms. The state of devices is reflected by sensors that provide information about their position. Each device has a set of actuators that are used by a control program to change the state of the device. Sensors also return information about failed devices.

Figure 4 shows the way in which CA actions enclose the control of a sequence of operations between devices. Each CA action encloses a set of devices that must interact in a coordinated fashion to satisfy the safety and fault tolerance requirements of the case study. If two CA actions overlap, they cannot be performed in parallel because they both involve the same device. For example, the *UnloadTable* CA action cannot be executed in parallel with the *LoadPress1* CA action because both CA actions involve the robot, and the robot can participate in only one of them at a time.

Each device in the FT Production Cell is controlled by a corresponding thread that is responsible for specifying the sequence of CA actions in which the device participates. For example, for simple cyclic execution, a device controller has a straightforward structure with an endless loop, which means that the thread of the controller executes a fixed sequence of CA actions. For example, the controller thread for the table in the FT Production Cell would repeatedly execute first the loading of the table and then the unloading of the table with metal plates.

Table controller **thread**:
loop
  execute table role in the *LoadTable* CA action
  execute table role in the *UnloadTable* CA action
end loop;

The loading of the table is an example of a CA action that is executed by three parties in our design: the controller thread for the table; the controller thread for the feed belt; and the controller thread for the traffic light at the beginning of the feed belt. Notice that any fault that may happen while loading the table will not affect the rest of the production cell directly because this fault will be enclosed by a CA action.

This same approach for dealing with devices has been adopted in [8] and [15]. However, in [15] the approach was applied for the development of the controlling software for a production cell where fault tolerance was not a major concern. Also, in [8] the semantics for dealing with faults is based on DMI semantics rather than CA action semantics presented in this paper.

4.1 Applying the API to the FT Production Cell

We have designed one object for each of the devices in the production cell. These device objects are composed of other devices and sensors, e.g. the robot device is composed of two arms and an angle sensor. The device objects are the ExternalObjects in our API. In Figure 5 we show three device objects for the FT Production Cell case study, e.g. a Table object that is composed of one ValueSensor, which stores the values of the angle sensor of the table, and three boolean sensors: one to indicate if the table is in its upper position; one to indicate if the table is in its lower position; and, one to indicate if there is a plate over the table. We have used this same approach to design all the device objects in our system.

Figure 5 shows how the objects of CA action are organized in the API for the LoadTable CA action. The LoadTable CA action encloses all the operations that are needed for the process of loading the table by the feed belt. The LoadTable CA action is composed of three role objects: FeedBeltRole, TableRole, and TLight1Role. These three roles act upon the devices in a coordinated way to load a metal plate from the feed belt to the table. The execution of these ac-

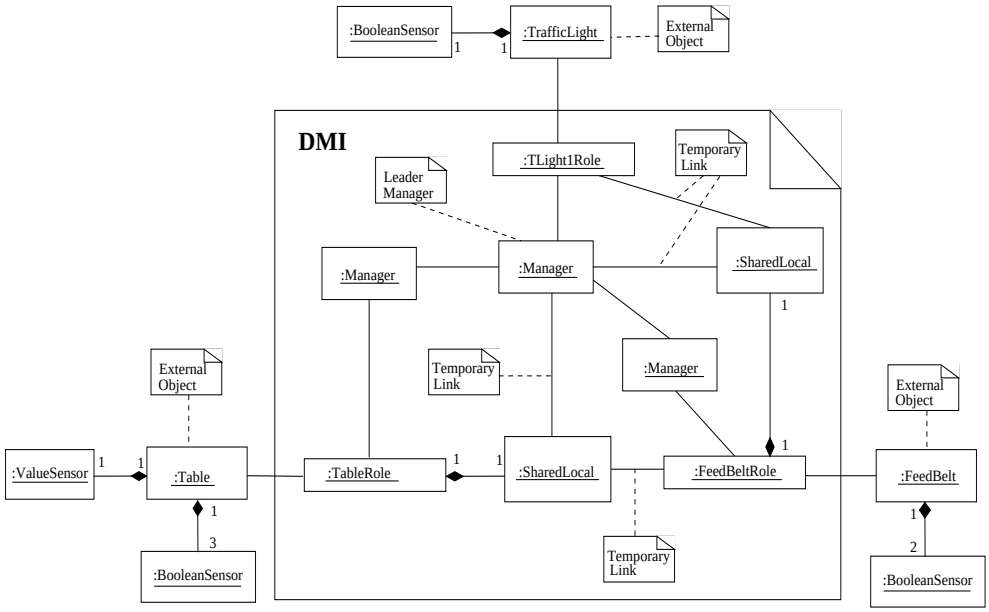

Figure 5: Links between objects for the LoadTable CA action (in UML)

tivities must use two SharedLocalObjects: one is used by the TableRole to inform the FeedBeltRole that the table is ready to be loaded; and the other is used by the FeedBeltRole to inform the TLight1Role that the traffic light can be turned to green.

The following code shows how the body of the TableRole is implemented for the LoadTable CA action.

```
protected void body(Transactional list[])
                throws Exception {
  Table table = (Table) list[0]; // External Object
  try {
    // Rotate to the left to the loading angle.
    table.left();
    table.angle().waitValue(POS_FEEDBELT);
    table.stop_h();
    // Move table down to the loading high.
    table.downward();
    table.down().waitValue(Boolean.TRUE);
    table.stop_v();
    // Inform feed belt that table is ready.
    waitTable.synchronize();
  } catch (Exception e) {
    // OOPS! Problems. Stop everything!
    table.stop_h();
    table.stop_v();
    throw e;   // Pass the exception to manager.
  }
}
```

Every time an exception that may affect the whole CA action is raised in a role, that exception has to be thrown by that role (see **catch** block above). The manager of the role will catch the exception and will start the exception handling process as explained in Section 2.1. In Figure 6, we show a possible scenario where two exceptions are raised during the execution of the LoadTable CA action. Two roles, FeedBeltRole and TableRole concurrently raise exceptions FeedBeltStuckException and TableAngleException respectively (step **1** in the figure). These exceptions are caught by the role managers which inform the leader about these exceptions (step **2**). The leader then detects that TLight1Role is still executing and interrupts the thread executing that role (step **3**). An InterruptedException is therefore raised by the manager of TLight1Role informing the leader that the role has been interrupted successfully (in this case the manager of TLight1Role and the leader are the same) (step **4**). The leader then decides upon which exception has to be handled: exception FeedBeltTableException in our example. Exception FeedBeltTableException is sent to all managers of the CA action (step **5**) which will activate the roles in an exception handling interaction to deal with exception FeedBeltTableException (step **6**). A new set of managers and roles in the handler will then begin execution as if they belonged to a normal interaction.

We have based our implementation of exception handling for the FT Production Cell on the failure analysis and definition presented in [16].

4.2 Discussion

The use of the framework presented in this paper has helped us to model interactions between objects as a separate activity. This facilitated the design and implementation of objects for the FT Production Cell case study, in the sense that objects that represent devices are only concerned with the basic operations of the devices. For example, in designing and implementing the robot object, we had to consider only the operations that the robot could perform, e.g. operations to rotate the robot: *left*, *right*, and *stop*. Operations that are related to the environment the robot is inserted

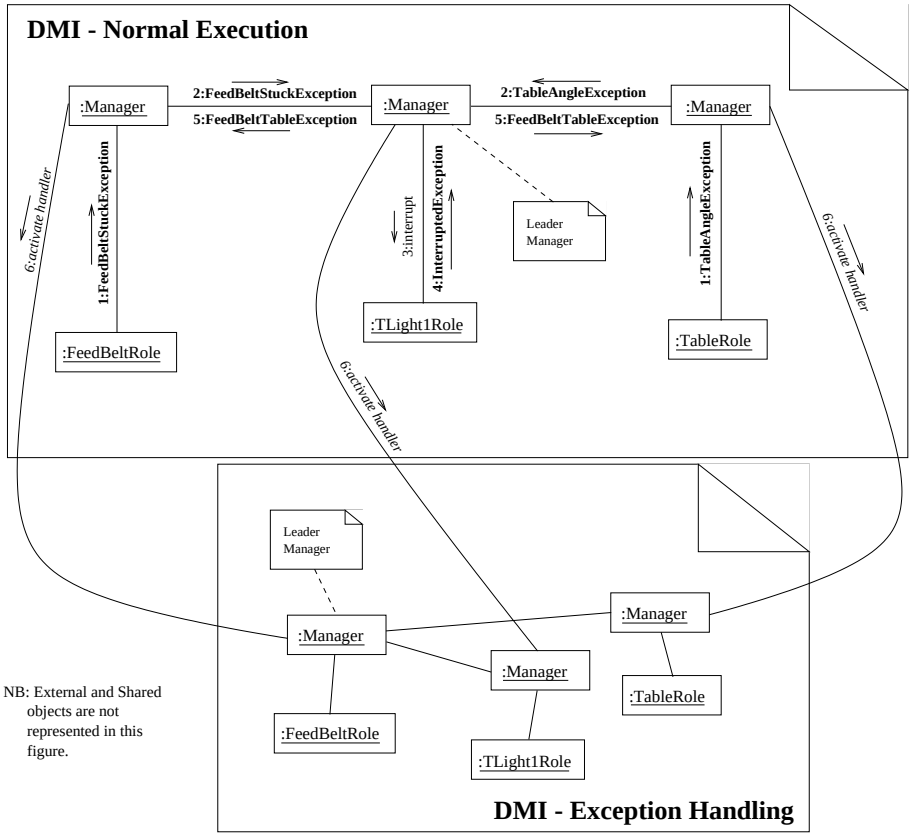

Figure 6: Dealing with Concurrent Exceptions

in, are not designed/implemented in the robot object, e.g. the unloading of the table, or the loading of the press by the robot. Because these operations can vary depending on where the robot is installed, they are left to be implemented in a separate place, making the reuse of the robot object possible without modifications. DMIs are well suited to this sort of strategy, and have the additional benefit of enclosing and recovering possible failures that may happen during this kind of interaction.

Figure 7 shows the costs of using our dependable multiparty interactions in the implementation of a system. We have measured the cost of an empty DMI and of a DMI where exceptions are raised by all roles of the interaction. In the graph we compare these two executions of our framework with a simple multiway rendezvous[3] mechanism. Even though our mechanism adds an overhead to the application interaction, we benefit from the inclusion of features that help the programmer to enclose failures and abstract the interactions from the objects. Furthermore, the overheads associated with using our framework are measured in microseconds whereas network and device overheads are measured in milliseconds, so the cost of using our framework is negligible for the kind of distributed applications we have described. We also notice that an increase in the number of participants does not cause the implosion of our framework, which scales in the same way as a simple multiway rendezvous mechanism.

The times in Figure 7 were measured on a 200Mhz Pentium PC running Linux and Java version 1.1.6. All participants were running in the same Java Virtual Machine.

[3] We created a new class that provides a Java synchronized method which blocks all its callers until it is called by the last caller (second caller for a 2-party rendezvous, third caller for a 3-party rendezvous, ...).

5 Related work

As mentioned in Section 1, there has been a lot of work on multiparty interaction, but most of it has been concerned with synchronisation, or handshaking, between parties rather than coordination of several activities executed in parallel by the interaction participants. Specification languages like CSP and LOTOS, programming languages like Ada, or mechanisms like Multiway Rendezvous, take into consideration only the synchronisation between processes. There is no mechanism that helps a programmer to enclose a set of activities that have to be executed in a coordinated fashion. Usually the programmer is left with all the work of coordinating interactions in those languages.

In Interacting Process (IP) [17], a high level notation for the expression of distributed systems, it is possible to describe multiparty interactions for basic synchronization and for interprocess communication. IP also provides an abstraction for defining a system composed of teams and roles. Each team is composed by roles that can be executed by different processes. Although IP is a good notation to describe multiparty interaction, it assumes a fault-free scenario when the interactions are being executed.

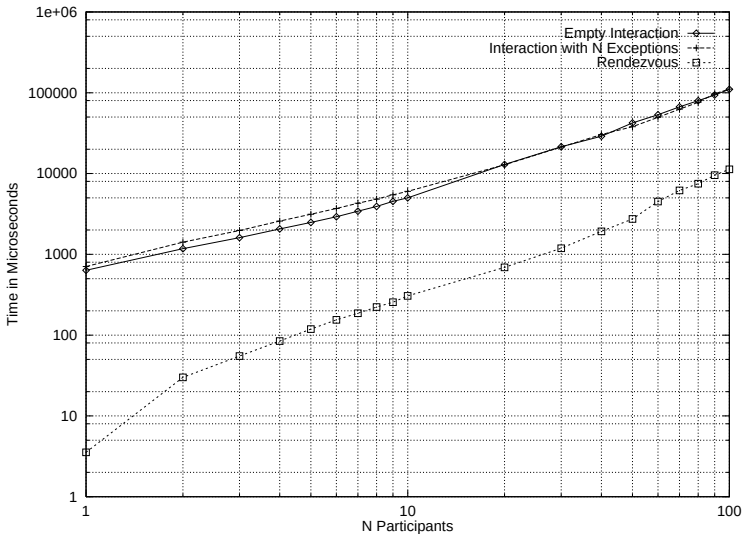

Figure 7: DMIs vs. Rendezvous

Distributed Cooperation (DisCo) [12] is a specification language based on objects and actions for reactive systems. In DisCo, objects are considered participants in an action. These participants can participate in only one action at a time. Actions in DisCo describe the transformation of the system state represented by the participants. DisCo does not allow actions to be nested and does not consider the possibility of failures during the execution of actions.

Recently, the Coordinated Atomic (CA) action concept [9, 10] has been introduced as a unified approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object-oriented system. Although the CA action mechanism is a very good approach for providing multiparty interactions in a dependable way, it may sometimes be rather restrictive due to its strong semantics of how to handle exceptions that occur during an interaction (see Section 3). A Coordinated Atomic action Language (COALA) [18] has been proposed to describe applications implemented using the CA action concept. Implementations of CA actions have been presented in [15] and [19]. However, these implementations use a different approach for distributing roles and managers.

We have chosen to implement the CA action mechanism using our API because it provides strong exception handling semantics for dealing with possible failures that happen during the execution of an interaction (see Section 3). Mechanisms like *actions* in DisCo [12] or *teams* in IP [17] do not consider exceptions in the execution of an interaction, let alone any description of the semantics for dealing with exception that can happen during the execution of an interaction. Therefore, using our API to implement such mechanisms would have been a trivial task.

A commonly used mechanism for providing consistency on shared objects is the transaction model. Languages like Argus [20] or Avalon [21] or systems like Arjuna [7] implement the transaction model, but do not provide any disciplined way of handling concurrent exceptions. The framework for DMIs presented in this paper provides a disciplined way of enclosing interacting processes that uses an underlying multi-threaded transaction systems to guarantee consistency on shared objects that may be accessed concurrently by different groups of interacting processes. In addition, our framework provides very general exception handling features, thus augmenting the fault tolerance provided by the underlying transaction system.

6 Conclusion

In this paper we have introduced a way of designing dependable multiparty interactions in distributed object-oriented systems. The API that provides DMIs has been used to implement a mechanism that brings together the concept of conversation and transaction: the Coordinated Atomic (CA) action concept. Using the implemented CA action mechanism, control software for a fault-tolerant production cell was produced.

DMIs are a very important way of describing cooperation between several participants even in the event of faults during the cooperation. We showed in this paper how to organize these kinds of activities in an object-oriented fashion. Interactions between objects were modeled as DMIs. This resulted in a very neat way of implementing basic objects to represent real devices in an industrial installation.

The way DMIs are activated by the underlying system is not taken into account in this paper. The properties used here suffice, even if different scheduling mechanism are used or if the DMIs are activated in a synchronous or asynchronous way.

A full language that includes all the properties for DMIs is being developed at the University of Newcastle upon Tyne. This language is called Dependable Interacting Processes and will be published shortly as a technical report. A compiler for this new language will then generate code that uses the API presented in this paper.

Acknowledgments

The ideas presented in this paper have benefitted from several discussions with Professor Brian Randell. We would like to thank our colleagues from the Department of Computing Science at the University of Newcastle, Jie Xu, Alexander

Romanovsky and Ian Welch for their contributions to discussions on the development of a framework for the CA action mechanism and for the design of the fault-tolerant production cell case study. We also thank several members of the ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa). A. F. Zorzo is being supported by CNPq/Brazil under grant number 200531/95.6.

References

[1] P. G. Neumann. "Distributed Systems Have Distributed Risks". In *Communications of the ACM*, 39(11), pp. 130, 1996.

[2] M. Evangelist, N. Francez, and S. Katz. "Multiparty Interactions for Interprocess Communication and Synchronization". In *IEEE Transactions on Software Engineering*, 15(11), pp. 1417-1426, November 1989.

[3] Y.-J. Joung and S. A. Smolka. "A Comprehensive study of the Complexity of Multiparty Interaction". In *Journal of ACM*, 43(1), pp. 75-115, January 1996.

[4] F. Cristian, "Exception Handling and Software Fault Tolerance". In *IEEE Transactions on Computers*, C-31(6), pp. 531-540, 1982.

[5] R. H. Campbell and B. Randell. "Error Recovery in Asynchronous Systems". In *IEEE Transactions on Software Engineering*, SE-12(8), pp. 811-826, 1986.

[6] A. Romanovsky, J. Xu and B. Randell. "Exception Handling and Resolution in Distributed Object-Oriented Systems". In *Proceedings of 16th IEEE International Conference on Distributed Computing Systems*, Hong Kong, pp.545-552, May 1996.

[7] S. Shrivastava, G. N. Dixon, and G. D. Parrington. "An Overview of the Arjuna Distributed Programming System". In *IEEE Software*, 8(1), pp. 66-73, 1991.

[8] A. F. Zorzo. "Dependable Multiparty Interactions: A Case Study". In *Proceedings of TOOLS-29 Europe 99*, IEEE CS Press, Nancy, France, pp. 319-328, June 1999.

[9] 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 *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, IEEE CS Press, Pasadena, USA, pp. 450-457, 1995.

[10] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. "Coordinated Atomic Actions: from Concept to Implementation". Department of Computing Science, Technical Report TR595, University of Newcastle upon Tyne, 1997.

[11] A. Charlesworth. "The Multiway Rendezvous". In *ACM Transactions of Programming Languages and Systems*, 9(2), pp. 350-366, 1987.

[12] H.-M. Järvinen and R. Kurki-Suonio. "DisCo Specification Language: Marriage of Actions and Objects". In *Proceedings of the 11th International Conference on Distributed Computing Systems*, IEEE CS Press, pp. 142-151, 1991.

[13] A. Lötzbeyer and R. Muhlfeld. "Task Description of a Fault-Tolerant Production Cell". FZI Technical Report, Karsruhe, Germany, 1996.
http://www.fzi.de/divisions/prost/projects/korsys

[14] C. Lewerentz and T. Lindner. "Formal Development of Reactive Systems: Case Study Production Cell". In *Lectures Notes in Computer Science 891*, Springer-Verlag, January 1995.

[15] A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. J. Stroud, and I. S. Welch. "Using Coordinated Atomic Actions to Design Safety-Critical Systems: A Production Cell Case Study". In *Software, Practice and Experience*, 29(8), pp. 677-697, August 1999.

[16] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. "Rigorous Developemnt of a Safety-Critical System Based on Coordinated Atomic Actions". In *Proceedings of the 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, IEEE CS Press, Madison, WI, USA, June 1999.

[17] I. Forman and F. Nissen. *Interacting Processes.* ACM Publishers. 1996.

[18] J. Vachon, D. Buchs, M. Buffo, G. D. M. Serugendo, B. Randell, A. Romanovsky, R. J. Stroud, and J. Xu. "COALA - A Formal Language for Coordinated Atomic Actions". In *DeVa - Design for Validation - Third Year Report*, ESPRIT Long Term Research Project 20072, December 1998.

[19] A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. J. Stroud, and I. S. Welch. "Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems". In *OOPSLA'97 - Workshop on Dependable Distributed Object Systems*, Atlanta, USA, 1997. (Extended version in *DeVa - Design for Validation - Second Year Report*, ESPRIT Long Term Research Project 20072, pp. 241-260, December 1997)

[20] B. Liskov. "Distributed Programming in Argus". In *Communcations of the ACM*, 31(3), pp. 300-312, March 1988.

[21] J. L. Eppinger, L. B. Mummert, and A. Z. Spector (editors). *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publ., 1991.