Journal of Universal Computer Science, vol. 10, no. 7 (2004), 843-858
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © *J.UCS*

Implementing Coordinated Error Recovery for Distributed Object-Oriented Systems with AspectJ

Fernando Castor Filho
(State University of Campinas, Brazil
fernando@ic.unicamp.br)

Cecília Mary F. Rubira
(State University of Campinas, Brazil
cmrubira@ic.unicamp.br)

Abstract: Exception handling is a very popular technique for incorporating fault tolerance into software systems. However, its use for structuring concurrent, distributed systems is hindered by the fact that the exception handling models of many mainstream object-oriented programming languages are sequential. In this paper we present an aspect-based framework for incorporating concurrent exception handling in Java programs. The framework has been implemented in AspectJ, a general purpose aspect-oriented extension to Java. Our main contribution is to show that AspectJ is useful for implementing the concerns related to concurrent exception handling and to provide a useful tool to developers of distributed, concurrent fault-tolerant applications.

Keywords: aspect-oriented programming, exception handling, coordinated atomic actions, separation of concerns, distributed programming
Categories: D.3.3, D.2, D.3

1 Introduction

Exception handling [8] is a well-known technique for incorporating fault tolerance [2] into software systems. An exception handling system (EHS) offers control structures that allow developers to define and raise *exceptions*, indicating the occurrence of an error, and *exception handlers*, responsible for putting the system back into a coherent state. *Handling contexts* are regions where the same exception types are treated in the same way. When an exception is raised, the underlying EHS interrupts the normal processing and transfers control to an appropriate exception handler. If no appropriate handler is available, the exception is signaled, or propagated, to an outer context, usually the caller of the operation where it was raised.

Various modern object-oriented programming languages include EHS's. Although some of these languages natively provide constructs for concurrent programming, in all of them exception handling is purely sequential. Some authors [3] argue that special features for involving many concurrent objects in exception handling are so difficult to develop and use that only sequential exception handling should be employed. In spite of this, concurrent (or coordinated) exception handling [4] is a powerful tool for structuring large, distributed, and concurrent software systems [18,21] and means for mitigating its inherent complexity are required.

Aspect-oriented programming (AOP) [14] has appeared recently as a means for modularizing systems that present *crosscutting concerns*. A crosscutting concern can affect several units of a software system and usually cannot be modularized by traditional object-oriented design techniques. It has been argued elsewhere [16] that exception detection and handling are crosscutting concerns that can be better modularized by the use of aspect-oriented techniques. However, works on the subject which employ AOP have focused solely on the sequential EHS's available in programming languages such as Java [5], C++ [11], and C# [10].

Although the model of Java for exception handling is representative of many object-oriented programming languages, it is not well-suited for exception handling in concurrent systems. Java does not prescribe adequate rules for propagation of exceptions signaled by a participant of a group of threads cooperating in order to achieve a common goal. For instance, in Java, if a participant is unable to handle an exception, its thread is simply killed. This may produce incorrect behavior, such as inconsistent results and deadlocks. Furthermore, it is not possible to associate handlers to elements that are meaningful to the concurrent execution of the group of objects. Romanovsky and Kienzle [17] argue that problems such as these are due to the fact that exception handling issues are being considered separately from those of system structuring. The authors suggest that exception handling should have a natural integration with constructs for concurrent execution.

In this paper, we describe an approach to implementing an aspect-based framework that complements the EHS of Java with coordinated exception handling. This framework, which we call ACE (**A**spect-based **C**oordinated **E**xception handling), was implemented in AspectJ [13], a general purpose aspect-oriented extension to Java.

Our goal is twofold: First, we want to analyze the benefits and disadvantages of using aspects to build a framework for coordinated error recovery, instead of relying on an exclusively object-oriented implementation [23]. Second, we want to provide support for the construction of reliable object-oriented systems with requirements such as concurrency and distribution.

This paper is organized as follows. Section 2 gives a brief overview of AspectJ. Section 3 describes the approach we employ for coordinated exception handling. Some background knowledge on sequential exception handling is assumed. Section 4 presents the design and implementation of ACE. Section 5 rounds the paper and presents some ideas for future work.

2 AspectJ Overview

AspectJ [13] is a general purpose aspect-oriented extension to Java. It extends Java with constructs for picking specific points in the program flow, called *join points*, and executing pieces of code, called *advice*, when these points are reached. Join points are used to capture *crosscutting concerns*, that is, concerns that affect several different program units and can not be modularized by traditional object-oriented techniques. A typical example of crosscutting concern is logging. The implementation of this concern must be scattered across all the modules in a system, because some contextual information must be gathered in order for the recorded information to be useful. Other common examples of crosscutting concerns include profiling and

authentication.

AspectJ adds a few new constructs to Java, in order to support the selection of join points and the execution of advice in these points. A *pointcut* picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call, method execution, field access, and class instantiation. A pointcut may be formed by the combination of various different join points selected only under specific conditions.

Advice are pieces of code that are executed when a join point is reached. These may be executed *before*, *after*, or *around* the selected join point. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point.

The language also allows programmers to modify the static structure of a program by means of *inter-type declarations*. Inter-type declarations can introduce new members in a class or interface, such as methods and fields, or modify the relationships between types.

Aspects are units of modularity for crosscutting concerns. They are similar to classes, but may also include pointcuts, advice, and inter-type declarations. The following code snippet presents a simple aspect.

```
01: public aspect SimpleAspect {
02:   public void Participant.exampleMethod() { ... }
03:   pointcut methodCallsFromParticipants(Participant p1):
04:       call(* Participant.exampleMethod(..)) && this(p1);
05:   before(Participant p1): methodCallsFromParticipants(p1){
06:        System.out.println("method called");}
07: }
```

In the aspect above, line 2 presents an inter-type declaration that adds the method **`exampleMethod()`** to the type **`Participant`**. If the latter is an interface, the new method is added to both interface and implementing classes. Lines 3 and 4 present a pointcut that selects calls to **`exampleMethod()`**. This pointcut has one argument of type **`Participant`**, corresponding to the caller of the method (**`this(p1)`**). It selects calls to the **`exampleMethod()`** method, defined by the type **`Participant`** (**`Participant.exampleMethod(..)`**). The method may have any return type (as indicated by the "**`*`**" symbol) and take any parameters. Line 5 defines an advice that is executed before the join points selected by the pointcut in line 3. This advice simply prints the message "method called" on the screen (line 6).

Aspects are associated to pure Java code by means of a process called *weaving*. Therefore, the tool responsible for performing weaving is called *weaver*.

3 Action-Oriented Exception Handling

In most programming languages, exception handling is inherently associated with system structuring concepts [17]. For instance, Java exception handling contexts are blocks of statements within method declarations and exceptions are objects defined by classes extending the class **`Exception`**. The exceptions that a method may signal define exit points for its execution (similarly to its result) that are only used when an error occurs. If an exception is raised within a method and no handlers for that

exception are available, the exception is propagated to the caller method.

In this paper, we use the concept of *action* to structure the concurrent execution of software systems. An action consists of a set of participants, units of computation such as threads or processes, that cooperate in order to achieve a common goal. The concept of Coordinated Atomic Actions (CA actions) [22] is employed to structure fault-tolerant concurrent systems in an action-oriented manner. CA actions provide a conceptual framework for dealing with different kinds of concurrency and achieving fault-tolerance by extending and integrating two complementary concepts: atomic actions [4] and ACID transactions [9]. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery. ACID transactions guarantee consistent access to shared objects.

A CA action consists of a set of participants cooperating inside it and a set of objects accessed by them (Figure 1). In a CA action, participants access shared objects that have the ACID properties. A CA action may terminate normally, in which case it produces a normal outcome and commits transactions on shared objects. If one or more exceptions are raised by action participants during action execution, all the participants are involved in coordinated exception handling. If two or more exceptions are concurrently signaled, an exception resolution scheme is used to combine these exceptions into a single one that represents all the exceptions signaled. If handling is successful, the action completes normally. Otherwise, an exception is propagated and responsibility for recovery is passed to the caller of the action. In this case, transactions on all shared objects are aborted. Actions may be nested in order to define different exception handling contexts. In the example of Figure 1, a nested action *Action2* is created within the action *Action1*.

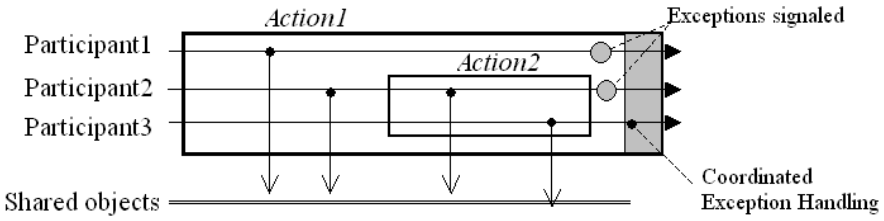

Figure 1: A CA action with three participants and a nested CA action.

4 Framework Design

ACE was built with the goal of complementing the EHS of Java with action-oriented exception handling, more specifically, with the concept of CA actions. In order to achieve this goal and to produce a reusable implementation, framework development has been guided from the beginning by the following design directives.

The concept of action should be explicit to application programmers. Application code based on ACE should employ the concept of action explicitly. This approach is adopted by the exception handling mechanism proposed by Garcia et al [7]. As mentioned in Section 3, exception handling is closely related to system structure and, in the case of concurrent exception handling, with constructs related to concurrent execution. Trying to make action-oriented exception handling transparent to

application code may have a negative impact on system maintenance and understandability.

Framework hotspots should be object-oriented. Framework users should only have to deal with object-oriented concepts. Although the framework is implemented as a combination of aspects, classes, and interfaces, application programmers should not need to know AspectJ or anything related to aspect-oriented programming, in order to use it. This directive makes framework usage easier for Java programmers and users of other object-oriented reusable implementations of coordinated exception handling.

The implementation of ACE consists of five main concerns related to the basic requirements of action-oriented exception handling and to some important non-functional requirements. The "core" framework comprises two concerns: action structuring (Section 4.1) and coordinated exception handling (Section 4.2). The remaining three concerns implement non-functional requirements: distribution (Section 4.3), preemptive abortion (Section 4.4), and transaction interface. The following subsections describe these points in detail, except for transaction interface, which is addressed elsewhere [15, 19]. In order to better assess our approach, throughout this section, we compare ACE to the purely object-oriented framework devised by Zorzo and Stroud [23] for building dependable multiparty interactions. The authors have used this framework, which we call DMI in the rest of this section, to implement the concept of CA actions.

4.1 Action Structuring

The action structuring concern makes it possible to organize concurrent systems as actions where participants cooperate in order to achieve a certain goal. In our approach for action structuring, participants are built by implementing the **`Participant`** interface. This interface defines only one method, **`execute()`**, that implements the basic functioning of the participant. It takes as argument an object of type **`Transactional`** which is shared by all action participants. The **`Transactional`** interface is defined by ACE and used for accessing objects in a transactional manner. Implementations of this interface should be provided by framework users.

Actions are represented by the **`Action`** interface, which extends **`Participant`**. This interface and a ready-to-use implementation are provided by ACE and implement the basic mechanisms for action structuring.

Implementation of the action structuring concern is complemented by an aspect, **`ActionStructuring`**. By means of inter-type declarations, this aspect makes **`Participant`** a subtype of **`Runnable`**, the interface provided by Java for defining objects that run in their own threads. Furthermore, it adds a new method to **`Participant`**, **`run()`**, that is called when a thread is started and is responsible for invoking **`execute()`** in each participant.

`ActionStructuring` is also responsible for managing references between actions and their participants. The following pointcut is used for this means. It selects all calls to the **`addParticipant()`** method, defined by **`Action`**.

```
01: pointcut participantAddition(Action cp, String id,
02:    Participant part): target(cp) && args(id, part)  &&
03:    execution(* *..Action.addParticipant(
04:      String,Participant));
```

The **addParticipant()** method includes a new participant in an action. The pointcut above captures the moment in which this method is executed, as well as some contextual information. This information consists of (i) the **Action** object on which the method was invoked (**target(cp)**), and (ii) the identifier for the participant being added, as well as the participant itself (**args(id, part)**) (lines 1 and 2). When this pointcut is reached, an advice is executed after it, setting **cp** as the action **part** belongs to. This information is stored by **Participant** in a new field introduced by an inter-type declaration.

Creation and execution of an action is very simple. The following code snippet creates an action **a1**, comprising two participants, **p1 and p2**, and executes it.

```
01: Participant p1 = new ParticipantImpl();
02: Participant p2 = new ParticipantImpl();
03: ActionFactory fac = ActionFactory.getInstance();
04: Action a1 = fac.createAction("a1", 2000);
05: Transactional shared = new TransactionalImpl("An action");
06: a1.addParticipant("Participant1", p1);
07: a1.addParticipant("Participant2", p2);
08: a1.execute(shared);
```

Lines 1 and 2 create two participants, **p1** and **p2**. Line 4 creates a new action with identifier **a1** and which will wait at most 2000ms for participants to complete their execution. Actions are started by calling the **execute()** method on the action object (line 8). This method starts each participant in a new thread, passing the shared objects (line 5) as arguments to each of them.

Discussion

Part of the **ActionStructuring** aspect could be implemented as a class from which all participants would inherit. We have avoided this approach, however, because Java only allows single inheritance and user applications might require participants to extend a certain class. Moreover, it is argued by some authors [6, 12] that code inheritance promotes very strong coupling between classes, and that interfaces, together with aggregation, should be used whenever possible. In our approach, all the extra functionalities required by concurrent exception handling that would otherwise be implemented in a superclass are provided by aspects.

The use of aspects for implementing this concern helped making ACE easier to use, since participant classes defined by user applications need only to implement an interface. Furthermore, our aspect-based implementation completely encapsulates thread management-related issues. For instance, in the DMI framework, the threads on which each participant will be executed must be explicitly created and started by framework users. In spite of this, the aforementioned benefits could also be achieved by employing wrappers [6] for participants, in a purely object-oriented solution. These wrappers would be responsible for executing participants in new threads and

maintaining references to the object representing the action. Hence, the aspect-based implementation did not present advantages over existing object-oriented solutions.

4.2 Coordinated Exception Handling

This concern enriches action structuring with concurrent exception handling. It is implemented by the **`CoordinatedExceptionHandling`** aspect. This aspect requires that an exception handler class be created for each class defining a participant, in a given application. Exception handler classes implement the **`ExceptionHandler`** interface, which defines a single method, **`handleExceptions()`**, responsible for handling one or more exceptions.

The **`CoordinatedExceptionHandling`** aspect defines a pointcut, **`participantExecution`**, that intercepts all calls to the **`execute()`** method of objects of type **`Participant`** that are not of type **`Action`**. Two advice are associated with this pointcut. The first one creates, for each participant in an action, an instance of the corresponding exception handler class and associates this object with the participant. If no exception handler class is available or instantiation fails, an empty exception handler that simply re-throws any exceptions received is associated to the participant. Participant and exception handler classes are matched by adopting a naming convention to their definition.

The other advice associated to the **`participantExecution`** pointcut catches and records any exceptions signaled by participants. The following code snippet presents part of this advice.

```
01: void around(Participant p) : participantExecution(p) {
02:   p.setExceptionalResult(null);
03:   try {
04:     proceed(p);
05:   } catch (Exception e) {
06:     p.setExceptionalResult(e); }
07: }
```

The **`proceed()`** (line 4) statement is defined by AspectJ and can only be used in **`around`** advice. It resumes the execution of the intercepted code (in this example, the **`execute()`** method of **`Participant`**) and, after it finishes, returns to the advice, in the line following the **`proceed()`** statement. In the example above, if no exceptions are raised, execution is finished with no additional behavior being introduced. Otherwise, the exception is caught and stored in the participant itself, by calling the **`setExceptionalResult()`** (line 6). This method is introduced in **`Participant`** by an inter-type declaration.

`CoordinatedExceptionHandling` also defines a pointcut, **`actionExecution`**, that intercepts calls to the **`execute()`** method of objects of type **`Action`**. The following code snippet presents an advice associated to this pointcut.

```
01: void around(Action cp)
02:     throws Exception : actionExecution(cp) {
03:   try {
04:     proceed(cp);
05:     this.performExceptionHandlingIfNecessary(cp);
06:   } catch (Exception e) {
07:     cp.setActionExceptionalResult(e);
08:       throw e;
09:    }
10: }
```

The **performHandlingIfNecessary()** method (line 5) called after the **proceed()** statement (line 4) checks if any of the participants signaled an exception during its execution. If none did, the method returns and the action terminates. Otherwise, it collects all the exceptions signaled by action participants and initiates exception resolution. Many different schemes are possible for exception resolution. The one adopted by ACE, in conformance to Java´s EHS, assumes that exceptions are defined by classes and chooses the most specific exception class that is a supertype of the types of all exceptions signaled by participants.

Figure 2 presents a simple exception class hierarchy used to illustrate how exception resolution works in ACE. If, during the execution of an action, two participants simultaneously signal exceptions of types **E3** and **E4**, exception resolution will create a new exception of type **E5**, since it is the most immediate superclass of both **E3** and **E4**. If, on the other hand, exceptions of types **E**, **E3**, and **E4** are signaled at the same time, the resolved exception will be of type **E**, because it is a superclass of both **E3** and **E4**, and because we assume that the superclass relation is reflexive.

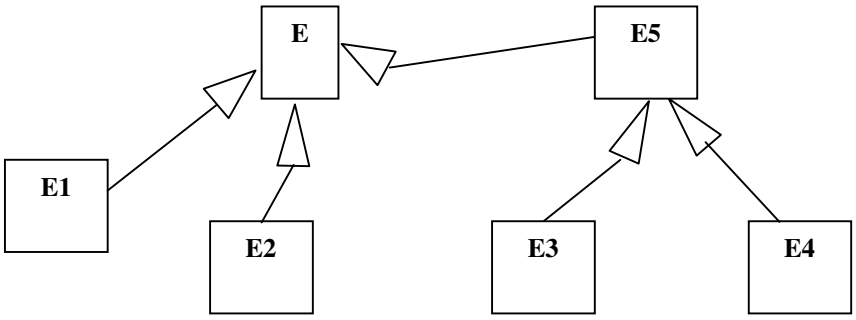

Figure 2: A simple exception class hierarchy.

After exception resolution finishes, the resolved exception is delivered to all participants and exception handling is initiated. For each participant, a new thread is created and the **handleExceptions()** method is called on the associated exception handler. This method takes the resolved exception as argument. Furthermore, the exceptions signaled by the participants during normal execution are also supplied as additional contextual information. Exception handling terminates normally if the participants do not signal any exceptions, while handling the resolved exception.

Otherwise, a **FailureException** is signaled, indicating that the action as a whole has failed and system state may be inconsistent.

Discussion

The implementation of coordinated exception handling by an aspect did not require any modifications to the code of the action structuring concern. Implementing similar features in a purely object-oriented language would either require the use of code inheritance, the solution adopted by the DMI framework, which we reject, or the construction of additional wrappers. Both cases require either the application code or the implementation of the action structuring concern to be modified. Therefore, the aspect-based solution promotes better separation of concerns.

When an exception is raised, it is useful to collect some information regarding the context in which it was raised. This contextual information is then made available to exception handlers, in order to increase the effectiveness of exception handling. Relevant information usually includes the values of local variables declared within the raising method. Unfortunately, it was not possible to collect this contextual information with the current version of AspectJ (1.1), since it does not allow the definition of pointcuts related to local variables.

4.3 Distribution

ACE has been devised with the goal of supporting development of large scale fault-tolerant object-oriented distributed systems. However, action structuring and coordinated exception handling concerns implicitly assume that all the participants of an action share the same address space and processing unit. This conflict between goal and assumptions is resolved by the distribution concern. We have used Java RMI [5] as the underlying distribution technology.

First Attempt: Aspects

In order to introduce the distribution concern, we began by implementing distribution for server-side objects. We first tried the approach described by Soares et al [19] for adding distribution to object-oriented systems by means of aspects. First, a new interface, **`DistributedParticipant`**, was created. This interface defines all the methods implemented by participants that may be accessed remotely. The methods in this interface should specify the **`RemoteException`** exception in their **`throws`** clauses because Java requires it from all methods that might be called remotely. **`DistributedParticipant`** extends **`Remote`**, the interface used by RMI to indicate that an object is remote, and includes both the methods defined by the **`Participant`** interface and the methods added to **`Participant`** by inter-type declarations. We then created a new aspect, **`Distribution`**, and made **`Participant`** a subtype of **`DistributedParticipant`** by means of an inter-type declaration.

This approach did not work, however, because classes defining remote objects must directly implement their remote interfaces. The solution described above works in the example application presented by Soares et al [19] because the classes defining remote objects implement the remote interface directly.

In the implementation of the distribution concern, the implementation classes are not known beforehand (because these are application-specific participants). In order to bypass the limitation described in the previous paragraph, we specified an inter-type declaration stating that all implementations of the **`Participant`** interface should also implement the interface **`DistributedParticipant`**. In this manner, classes defining remote objects would directly implement the remote interface. This is defined as follows.

```
01: declare parents: (Participant+ && !Participant)
02:   implements DistributedParticipant;
```

This solution works fine when no aspects define inter-type declarations affecting **`Participant`**. However, this is not the case for our implementations of the action structuring and coordinated exception handling concerns. Hence, an alternative solution should be found.

A Solution Based on Aspects and an OO Design Pattern

We have adapted the work of Alves and Borba [1] to introduce distribution in ACE. This work proposes a design pattern, called *Distributed Adapters Pattern* (DAP), for implementing distribution in layered systems. This pattern isolates distribution-related code in two classes, called distributed adapters. Our implementation of distributed adapters defines five roles: local interface, local object, remote interface, client-side adapter, and server-side adapter.

The *local interface* specifies services to be made remote. In ACE, **`Participant`**, including and all the methods added to it by means of inter-type declarations, is a local interface. Instances of classes implementing the local interface are called *local objects*. The *remote interface* **`DistributedParticipant`** defines the same methods as the local interface, but the exception **`RemoteException`** is specified in the **`throws`** clause of each of them, since they will be called remotely. The *client-side adapter* is defined by the **`DistributedClient`** class. It implements the local interface and maintains a remote reference to the server-side adapter. The client-side adapter makes distribution transparent to clients by delegating method invocations received from them to the server-side adapter. The *server-side adapter* is defined by the **`DistributedParticipantImpl`** class. It implements the remote interface and maintains a reference to the local object. This adapter delegates received remote method invocations to the local object.

The problem with using the distributed adapters pattern in a purely object-oriented manner is that adapter creation must be invoked explicitly by application code, even if factories [6] are used to hide the actual instantiation process. We have employed aspects to try to alleviate this limitation of the object-oriented approach. A new aspect has been created, **`DistributionWithDAP`**, that localizes the solution to this problem. Our approach is divided in two parts: server-side and client-side distribution.

Server-Side Distribution

A simple implementation pattern adopted by many distributed applications created with Java RMI consists in placing the code responsible for instantiating and publishing remote objects in the **main()** method of one or more server classes. In our approach, we assume that this pattern is adopted by framework users. However, instead of creating distributed objects and publishing them, the implementation of such methods should only instantiate the local objects to be made remote. The **DistributionWithDAP** aspect checks, for each instantiated local participant, if a corresponding remote participant has been published. This task is performed by a *before* advice associated to the **callToMain** pointcut, which intercepts calls to the **main()** method of a participant. The following code snippet presents part of this advice.

```
01: try {
02:   String addr = addresses.getProperty(participantId);
03:   if(addr!=null && addr.trim().length() > 0) {
04:     Remote r = Naming.lookup(addr);
05:     toBePublished[i] = false;
06:   }
07: }catch(NotBoundException nbe) { toBePublished[i] = true; }
```

The address for one of the participants, **addr**, is obtained (line 2) by means of a list of properties loaded at aspect initialization. If a participant has already been published at **addr** (line 4), it should not be published again (line 5). Otherwise, a **NotBoundException** is raised and caught, indicating that the participant was not published yet (line 7).

The other advice associated to the **callToMain** pointcut complements the one presented above and binds unpublished remote participants to the supplied identifier, after the execution of the **main()** method. Remote participants are created when participant classes are instantiated. A pointcut, **callToConstructor**, selects all calls to constructors of classes implementing the **Participant** interface. An advice creates objects of type **DistributedParticipant** when **callToConstructor** is reached. The following code snippet presents this advice.

```
01: void around(Participant p) : callToConstructor(p) {
02:   proceed(p);
03:   if(!alreadyInstantiated(p)) {
04:     DistributedParticipant dpi = null;
05:     dpi = new DistributedParticipantImpl(p);
06:     storeDistributedParticipant(dpi);
07:   }
08: }
```

First, the participant is created (line 2). The check in line 3 avoids the creation of more than one object of type **DistributedParticipant** for the same participant. This may happen due to calls to superclass constructors. In line 5, the distributed participant is actually created and in line 6 it is stored, so that it can be published later.

Client-Side Distribution

Transparency at the client-side is achieved by guaranteeing that remote references to distributed objects are acquired as if they were local. Our approach consists of requiring that local instances of the remote participants be created at the client-side (as if it were not a distributed application), intercepting the moment in which a local participant is added to an action, obtaining the remote reference corresponding to the local object being added, and adding the remote reference, instead of the local one.

The **participantAddition** pointcut defined by the aspect **ActionStructuring** intercepts calls to the **addParticipant()** method of the **Action** interface. Hence, we moved this pointcut to an abstract aspect and made both **DistributionWithDAP** and **ActionStructuring** extend this aspect. A new advice, associated to **participantAddition**, was then added to **DistributionWithDAP**. The code for this advice is presented below.

```
01: void around(String id, Participant p, Action ac) :
02:     participantAddition(id, p, ac) {
03:   String addr = addresses.getProperty(id);
04:   try {
05:     Remote r = Naming.lookup(addr);
06:     if(!validParticipant(r)) proceed(id, p, ac);
07:     else proceed(id, new DistributedClient(
08:          (DistributedParticipant)r), ac);
09:   } catch(Exception e) { proceed(id, p, ac); }
10: }
```

The identifier of the participant being added is used to obtain the address of the corresponding remote object (line 3). This address is used to obtain the remote reference (line 5). If it is not possible to obtain a valid remote reference, the local participant is used instead (line 6). Otherwise, a new client-side adapter of type **DistributedClient** is created to encapsulate the remote reference. This adapter is then added to the action as a new participant (lines 8 and 9). If any error occurs during this process, the local participant is used (line 9).

Discussion

The implementation of the distribution concern described in this section is clearly superior to the purely object-oriented version. Complete transparency to application code could be achieved due to the use of aspects. In order to use the distributed version, only a small amount of Java code responsible for creating the distributed objects is required. Furthermore, application code used to build distributed applications makes no references to distribution-related issues. Hence, if a distributed application needs to be used in a local setting, no modifications are required. It is simply a matter of not weaving the **DistributionWithDAP** aspect.

Comparing our approach to the one presented by Soares et al [19] is difficult because we have not evaluated the use of aspects together with distributed adapters in more general contexts. Although the solution by Soares et al could not be employed to implement distribution in ACE, we are still not aware of all the limitations of our own approach. Hence, claiming that one is more or less general than the other would be inadequate.

The main cause for the difficulties outlined in this section is the fact that the current version of AspectJ (1.1) does not support the addition of new exceptions to the **`throws`** clause of a method. Hence, it was necessary to define another remote interface for distributed participants. This problem has been reported elsewhere [15, 19] and an extension to AspectJ that effectively solves it has been suggested [19].

4.4 Preemptive Abortion

When an exception is signaled by one of the participants of an action, all the other participants should abort their executions so that exception handling can take place. This makes execution faster and does not allow errors to spread throughout the system. Usually, implementations of CA actions based on Java use the features provided by the language for thread interruption, in order to implement participant abortion. This is the solution adopted by the DMI framework.

The problem with Java´s features for thread interruption is that they do not actually guarantee that a thread will be interrupted. Thread interruption is requested by invoking a method that marks the thread as interrupted. Actual interruption only occurs if an interrupted thread blocks waiting for locks or I/O operations. In order to guarantee that a participant will not go on executing when it should abort, we have implemented an aspect, **`PreemptiveAbortion`**, that aborts the execution of a participant almost immediately.

Participants of an action should be aborted when at least one of them has signaled an exception. As described in Section 4.2, any exception signaled by a participant is recorded by calling the **`setActionExceptionalResult()`** method. The pointcut **`storingRaisedExceptions`** selects all invocations to this method. The following advice, associated to this pointcut, is responsible for notifying the other action participants that their executions should be aborted.

```
01: void around(Participant p, Exception e) :
02:     storingRaisedException(p, e) {
03:   proceed(p, e);
04:   Action ac = p.getEnclosingAction();
05:   notifyParticipants(ac, p);
06: }
```

First, the signaled exception is recorded (line 3). The **`Action`** object corresponding to the action of which **`p`** (line 1) is a participant is then obtained by calling the **`getEnclosingAction()`** method (line 4). Finally, all the participants within the action **`ac`** (line 4) are notified (line 5).

Preemptive abortion is implemented by checking if, at any moment during the execution of a participant, it was notified that it should abort. The following pointcut selects all the statements within the **`execute()`** method of a class that implements **`Participant`**, but not **`Action`**.

```
01: pointcut participantsExecuteMethod(Participant p):
02:   withincode(* *..Participant+.execute(..)) && target(p)
03:   && !withincode(* *..Action+.execute(..));
```

Actual abortion is implemented by a simple advice, associated to the pointcut above, which checks, after each statement, if execution should be aborted. In case it should, this advice raises a special runtime exception, **`AbortionException`**.

Discussion

The implementation of the preemptive abortion concern nicely showcased the possibilities of using aspect-oriented programming together with object-oriented programming. It would not be feasible to implement this concern transparently to user applications without employing AspectJ, since it requires intercepting all the statements in a method body. This level of granularity is too low for typical solutions for interception, such as wrappers and proxies [5].

Since the **`PreemptiveAbortion`** aspect uses inter-type declarations to introduce some new methods in **`Participant`**, the implementation of the distribution concern needed to be extended in order to accommodate these changes. This could be achieved, however, without modifying any of the aspects, classes, or interfaces implementing either distribution or preemptive abortion. A new aspect, **`DistributedPreemptiveAbortion`**, was created which complements the client and server adapters with the new methods introduced by **`PreemptiveAbortion`**.

5 Conclusions

In this paper, we have described ACE, an aspect-based framework for implementing coordinated exception handling in distributed object-oriented systems. Our main contribution was to assess the adequacy of using AspectJ to implement a reusable infrastructure for coordinated exception handling. To the best of our knowledge, all the works published on the use of aspects to structure exception handling refer exclusively to sequential exception handling. Moreover, we have compared ACE to a purely object-oriented approach to implementing coordinated exception handling. Strong and weak points of both our approach and AspectJ have been pointed out.

The aspect-based implementation of the action structuring concern did not present advantages over an object-oriented one, mainly due to the design directives described in Section 4. In spite of this, our results lead us to conclude that our aspect-based reusable implementation for coordinated exception handling is superior to a purely object-oriented one. Coordinated exception handling, distribution, and preemptive abortion have been designed so that these concerns can be added to an action-structured application in a non-intrusive manner. This level of transparency could not be achieved by employing only object-oriented techniques, such as design patterns.

In order to assess our framework from a usability viewpoint, we intend to build a large case study based on ACE, employing different communication technologies, such as Enterprise Javabeans [20]. This will allow us to better understand its advantages and, most of all, its limitations. Furthermore, it will clarify how our design decisions influence the overall framework usability and provide suggestions for features to be added in the future.

Another future work consists of extending ACE with features useful for building component-based systems. This is an ongoing work that is in its initial stages. We are

currently studying the implications of using an aspect-based framework such as ACE for building applications whose computational model requires black-box visibility.

Acknowledgements

We would like to thank the anonymous referees for helping to improve this paper. We would also like to thank Paulo Borba for helping us to understand how aspects interact, and Alexandra Barros and Lásaro Camargos for the comments and suggestions on early drafts of the paper. Fernando Castor is supported by FAPESP/Brazil under grant no. 02/13996-2. Cecília Rubira is supported by CNPq/Brazil, under grant no. 351592/97-0.

References

[1] Alves, V. and Borba, P. (2001) "Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications". In: *Proceedings of the First Latin American Conference on Pattern Languages Programming* (SugarLoafPLoP), Rio de Janeiro, Brazil.

[2] Anderson, T., and Lee, P. (1990) *Fault Tolerance: Principles and Practice*, 2nd Edition, Prentice-Hall.

[3] Buhr, P. A. and Mok, W. Y. R. (2000) "Advanced Exception Handling Mechanisms". In: *IEEE Transactions on Software Engineering*, **26**(9), pp 820 – 836, IEEE Computer Society Press.

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

[5] Campione, M., Walrath, K., and Huml A. (2000) *The Java Tutorial: A Short Course on the Basics*, 3rd Edition, Addison-Wesley.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[7] Garcia, A., Beder, D., Rubira, C. (1999) "An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach". In: *Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering (ISSRE´99)*, USA, pp. 52 – 61, IEEE Computer Society Press.

[8] Goodenough, J. B. (1975) "Exception Handling, Issues and a Proposed Notation". In: *Communications of the ACM*, **18**(12), pp. 683 – 696.

[9] Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann.

[10] Hejlsberg, A., Wiltamuth, S., and Golde, P. (2003) *The C# Programming Language*, Addison-Wesley, Reading, MA, USA.

[11] Koening, A. and Stroustrup, B. (1990) "Exception Handling for C++", In: *Jornal of Object-Oriented Programming*, **3** (2) , pp. 16 – 33.

[12] Hollub, A. (2003) "Why Extends is Evil", In: *JavaWorld.com.* Available at: http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox_p.html

[13] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten., M., Palm, J., and Griswold, W. G. (2001) "Getting Started with AspectJ". In: *Communications of the ACM*, **44**(10), pp. 59 – 65.

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997) "Aspect-Oriented Programming". In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Finland, LNCS **1241**, Springer-Verlag, pp. 220 – 242.

[15] Kienzle, J., Guerraoui, R. (2002) "AOP: Does it Make Sense? The Case of Concurrency and Failures". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02)*, Málaga, Spain, LNCS **2374**, Springer-Verlag, pp. 37 – 61.

[16] Lippert, M. and Lopes, C. V. (2000) "A Study on Exception Detection and Handling Using Aspect-Oriented Programming". In: *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, pp 418 – 427.

[17] Romanovsky, A. and Kienzle, J. (2001) "Action-Oriented Exception Handling in Cooperative and Competitive Object-Oriented Systems". In: Romanovsky A., Dony, C., Knudsen, J. L., Tripathi, A. (Eds.), *Advances in Exception Handling Techniques*, LNCS **2022**, Springer-Verlag, pp. 147 – 164.

[18] Romanovsky, A., Periorellis, P., and Zorzo, A. F. (2003) "Structuring Integrated Web Applications for Fault Tolerance". In: *Proceedings of the 6th IEEE International Symposium on Autonomous Decentralized Systems (ISADS'03)*, Pisa, Italy, pp 99 – 106.

[19] Soares. S., Laureano, E., Borba, P. (2002) "Implementing Distribution and Persistence Aspects with AspectJ". In: *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pp 174 – 190.

[20] Sun Microsystems. (2002) *Enterprise Javabeans Specification v2.1 - Proposed Final Draft*.

[21] Tartanoglu. F., Issarny, V., Romanovsky, A., and Levy, N. (2003) "Coordinated Forward Error Recovery for Composite Web Services". In: *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, pp 167 – 176.

[22] Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R. J., Wu, Z. (1995) "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". In: *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, Pasadena, California, USA, p. 499 – 509.

[23] Zorzo, A. F. and Stroud, R. J. (1999) "A Distributed Object-Oriented Framework for Dependable Multiparty Interactions". In: *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, USA, p. 435 – 446.