Exception Handling in CSCW Applications in Pervasive Computing Environments

Anand R. Tripathi*, Devdatta Kulkarni, and Tanvir Ahmed

Department of Computer Science,
University of Minnesota, Minneapolis, MN 55455 U.S.A.
{tripathi, dkulk, tahmed}@cs.umn.edu

Abstract. In this paper we present conceptual foundations of an exception handling model for context-aware CSCW applications. Human participation in the recovery actions is an integral part of this model. Role abstraction is provided with an exception interface through which the role members can perform exception handling actions. Exception handling involving multiple role members is also supported through inter-role exception propagation mechanisms provided in the model.

1 Introduction

There is a growing interest in building context-aware applications and pervasive computing environments that allow mobile users to seamlessly access computing resources to perform their activities while moving across different computing domains and physical spaces [1,2,3]. A typical user is generally involved in many activities such as office workflow tasks, distributed meetings, collaborative tasks, personal activities such as shopping or entertainment. Many of these activities may involve multiple users collaborating on some shared tasks. Applications built for such environments are characterized by dynamic integration of large number of components based on the user context and ambient conditions. These characteristics impart a malleable nature to these applications giving rise to different kinds of error conditions and abnormal situations which are caused by following kinds of failures:

- Context-based dynamic resource discovery and binding may fail because of unavailability of required type of resources in the environment.
- Failures could arise while accessing the resources/services because of incompatible resource access protocols or insufficient security privileges. Also a resource may encounter internal failures.
- Some users working towards a common goal in a collaborative application may fail to perform certain obligated tasks. This may affect other users in the collaboration.
- Events occurring in the physical world may affect the application. These events may violate the application's assumptions about the state of the external world.

* This work was supported by NSF grant 0411961.

C. Dony et al. (Eds.):Exception Handling, LNCS 4119, pp. 161–180, 2006.

In such applications context can be classified into two categories: *internal* and *external*. The internal context of a CSCW (computer supported collaborative work) application is related to the execution state of its various tasks. The external context represents the attributes that are related to the physical environment. A user's external context may be defined in terms of a number of different kinds of attributes, such as the user's current physical location (GPS, presence in a building or room, proximity to certain devices or users), the Internet domain in which the user is currently present, or devices through which the user is interacting with the environment.

We have developed a programming framework for building context-aware CSCW applications in pervasive computing environments [4,5]. In this framework, such applications are built from their high level specifications expressed in XML and realized through a distributed middleware [6]. The specification model provides the abstraction of *roles* for users to participate in an *activity*. This specification model is essentially a composition framework for integrating users, application-defined components, and infrastructure services/resources to build the runtime environment of a context-aware CSCW application.

In this paper we present a methodology for handling error conditions and abnormal situations arising in context-aware CSCW applications. The methodology is based on programmed error handling wherein exceptions are used for representing different types of failures encountered by an application and exception handlers are built into the application to perform recovery actions. We present different failure categories to motivate the exception handling requirements in context-aware CSCW applications. Human participation in exception handling is an integral part of error recovery in such applications. Towards that end, the exception handling model provides mechanisms for involving role members in handling exceptions and for propagating exceptions from one role to another.

Section 2 describes our specification model for building context-aware CSCW applications in pervasive computing environments. In Section 3 we present a categorization of the error conditions arising in these applications. We also present the exception handling requirements for context-aware CSCW applications in this section. In Section 4 we present a model for exception handling in our role-based programming framework and we demonstrate its capabilities through two examples in Section 5. In Section 6 we discuss related work and conclude in Section 7.

2 Specification Model for Context-Aware CSCW Systems

We present here an overview of the collaboration specification model which we have developed [5,4]. A CSCW application is modeled as an activity. Activity defines three things: a set of objects representing shared resources and infrastructure services, a set of roles through which a group of users cooperate towards some common objectives by performing tasks involving shared resources and infrastructure services, and a set of actions (called reactions) that are triggered by

events occurring within an activity or events occurring in the external physical environment.

The shared resources/services required by an activity may be created by the activity or discovered in the environment. Resources are described using RDD, which is an XML schema based on RDF [7] and WSDL [8]. An RDD for a resource includes the attribute-value pairs describing the resource, the interfaces, and the events that are exported by the resource.

A user joins one or more roles in the activity, and a role represents authorization of its members to invoke a set of operations on shared objects to perform tasks in the collaboration space. A precondition associated with a role operation must be satisfied to execute the operation. These preconditions are based on both internal and external events.

Both internal and external context can affect various aspects of the activity. The binding of the shared resources/services may need to be changed based on the context, and the role operation precondition may also depend on the context.

Activity activityName
{**Parameter** objName}
{**Object** [**Collection**] *objName* **RDD** *rddSpec* }
{**Bind** *Binding-Definition*}
{**Reaction** *Reaction-Definition*}
{**Role** *Role-Definition*}

Fig. 1. Activity Syntax

We represent activity specifications through an XML schema. Here, rather than using XML, we use a notation that is conceptually easy to follow. In Figure 1, the syntax for the XML schema for *activity* definition is shown, where `[ ]` represents optional terms, { } represents zero or more terms, | represents choice, and boldface **terms** represent tags for elements and attributes in XML schema. In this paper we do not address exception handling with nested activities.

Specification of a sample CSCW application, an exam session activity, is shown in Figure 2. Two roles are defined in the exam session activity, *Student* and *Examiner*. Multiple users can be present in both the roles. The *Student* role is provided with operations for taking the exam such as: *StartExam*, *WriteExam*, and *SubmitExam*. The *Examiner* role is provided with the operation *GradeExam* for grading the exam. We use the exam session activity to demonstrate the specification model in this section and extend it to incorporate the exception handling model in Section 4.

2.1 Event Based Coordination Model

Events are used for task coordination within an activity. Three types of events are defined in the model: system defined events, application defined events, and

```
1. Activity ExamSession
2.    Object examPaper
3     Object Collection StudentAnswers
4.    Bind examPaper With direct (//ExamPaperURL)
5.    Role Student
6.       Object answerBook
7.       Bind answerBook With new (//AnswerBookCodeBase)
8.       Operation StartExam
9.          Precondition true
10.         Action answerBook.startExam()
11.      Operation WriteExam
12.         Precondition #StartExam.start(invoker=thisUser) > 0
13.         Action answerBook.writeExam()
14.      Operation SubmitExam
15.         Precondition #StartExam.start(invoker=thisUser) > 0
16.         Action Bind answerBook With StudentAnswers
17.   Role Examiner
18.      Operation GradeExam
19.         Precondition #Student.SubmitExam.finish > 0
20.         Action StudentAnswers.gradeExam()
```

Fig. 2. Exam Session Activity Specification

external world events. System defined events are *start* and *finish* events associated with each role operation, and they are generated by the middleware implicitly. Application defined events are signaled explicitly using the *NotifyEvent* construct defined in the specification model. The role name, role member name, and the event to be notified are given as parameters to this construct. The external world events are notified to the activity through the shared resources and services bound in the activity scope.

Operation *start* and *finish* events have two predefined attributes: *invoker* and *time*. The history of occurrences of an event type is represented by an event list. The specification model supports various functions on event lists. The count-operator **#** returns the number of events of a given type that have occurred so far, and a sublist of these events can be obtained by applying a selector predicate.

2.2 Role, Operation, and Reaction

A *role* in an activity defines two things: an object space for the role members and a set *operations* that are executed by the role members on objects defined within the activity scope or the role scope. The objects declared within a role represent a separate namespace created for each member in the role, and binding of these names is performed independently for each member.

A role operation can only be invoked by a member in the role. A role operation can have precondition that must be satisfied before the operation is executed.

These preconditions are expressed in terms of predicates based on events occurring within the activity, role memberships of participants, and query methods of the environmental resources representing external context information. In contrast to an operation, a reaction is not invoked by a user but is automatically executed when certain events occur. Reactions are specified in the activity scope. Similar to an operation, a reaction is executed only when its precondition is true. Figure 3 presents the syntax of a role definition.

The variable `thisUser` is used in our specification framework for identifying the role member who is invoking a particular role operation. A boolean function `member(thisUser, roleId)` is defined for checking the role membership of the user invoking the role operation. The function `members(roleId)` returns the role member list. Set operations can be performed on role member lists. A *count* operator, `#`, can be applied on a member list. The count of the members in a role is `#(members(roleId))`.

In the exam session activity, the operations *WriteExam* and *SubmitExam* can only be performed by a *Student* role member if that role member has previously performed *StartExam* role operation. This is specified as the preconditions for *WriteExam* (line 12) and *SubmitExam* operations (line 15). The *Examiner* can perform *GradeExam* operation only after *Student* role members have *finished* executing the *SubmitExam* role operation. This is specified as precondition to the *GradeExam* role operation (line 19).

A role also has *admission constraints* and *activation constraints* associated with it. We do not discuss them here as they are not related to the exception handling model presented here.

Role roleName
{**Object** [**Collection**] *objName* **RDD** *rddSpec* }
{**Bind** *Binding-Definition*}
{**Operation** *Operation-Name*}
[**Precondition** *Condition*]
[**Action** {objId methodSignature methodParameter}]

Fig. 3. Syntax for role definition

2.3 Binding Specification

Resources and services required as part of the activity or a role can be specified using the *Bind* primitive in four different ways as shown below.

1. Binding to a new object: The binding primitive with `new` specifies that a new resource of the specified codebase type should be created. For example, in the exam session activity, an *AnswerBook* object is created and bound to the name *answerBook* as shown in line 7 of Figure 2. This binding is specified inside the *Student* role scope since a separate *answerBook* object needs to be created for each *Student* role member.

2. Binding to an existing resource through URL: This form of binding primitive with `direct` specifies that the resource identified by the given URL should be used in binding. For example, in the exam session activity, the URL of the *ExamPaper* might be well-known. Such a direct binding is shown in line 4 of Figure 2.
3. Binding to another object: This type of binding is used to export a resource, created as part of a role, to the activity scope. For example, in the exam session activity the *answerBook* object for every *Student* role member is exported to the *StudentAnswers* collection (defined in line 3) as part of the *SubmitExam* role operation (line 16).
4. Binding through discovery: This form of the binding primitive is useful when a resource with a particular set of attributes is needed to be discovered in the environment. In the example below, we present specification of a museum information desk activity. A separate activity is instantiated for each museum visitor. In this activity, the audio channel of user's device needs to be bound with the audio player based on the user's location and also taking into consideration the user's choice of the language. In this example, the *audioChannel* object is re-binded when there is a change in the user's location, indicated by the *LocationChangeEvent* generated by the *locationService* when the visitor's location changes. Discover primitive used in binding the *audioChannel* object specifies the location attribute and the preferred language in the *Audio-Channel-Description* to be used during resource discovery.

Activity Museum Infodesk
  Object locationService **RDD** Location-Service-Description
  Parameter userPreference
  Bind locationService **With direct***(//LocationServiceURL)*
    Role Visitor
      Object audioChannel **RDD** Audio-Channel-Description
      Bind audioChannel **When** LocationChangeEvent(thisUser)
        With discover*(<location=locationService.getLocation(thisUser),*
                language=userPreference.preferredLanguage>)
      Operation ListenAudio
        Precondition true
                Action audioChannel.listenAudio()

3 Failures in Context-Aware CSCW Applications

An activity encounters *error conditions* as a result of various failures occurring in resources/infrastructure services being used by the activity [9]. *Exceptions* are *raised* in the activity by the middleware to indicate these failures. Exception handling actions are programmed in the activity to recover from the error conditions. The exception based approach for handling failures decouples the activity from the failure monitoring tasks corresponding to various resources/services being used by the activity.

In this section we identify various failure categories for context-aware CSCW applications. We use several examples to demonstrate the nature of failures. We identify requirements that need to be supported by an exception handling model for handling these failures.

3.1 Resource Discovery and Binding Failure

Dynamic discovery and binding of resources based on the user context or application context is one of the important aspects of context-aware applications. An activity encounters an error when the discovery of the required type of resource in the environment fails. Such errors are to be anticipated for applications in the pervasive computing environments because of their highly dynamic nature. Alternate resources may need to be discovered and bound to handle these errors. In the worst case, if no appropriate resources are found, then users must be involved in handling these errors.

Consider a museum infodesk activity in which users can bind to the audio commentary of an artifact when they move close to the artifact. This discovery and binding may fail if the audio commentary resource is not available in user's preferred language or if the audio commentary resource is not available at all. The exception handling mechanism should allow the application to transparently handle this exception by discovering and binding with audio commentary in another language or may provide alternative operations to the user by binding to the textual commentary for the artifact. The exception handling mechanism thus needs to have the ability to perform *automatic rebinding actions* and should also support *alternate interfaces* that are enabled depending on the objects that are bound.

3.2 Resource Interaction Failure

Resource interactions may fail because of insufficient security privileges, incompatible interaction protocols, resources being busy, or internal resource failures. For example, consider the exam session activity involving different students. Multiple students may encounter errors while performing one of the operations provided for the *Student* role because of operation failures. Each corresponding exception should be handled separately for each student in its appropriate context. There can be multiple ways to handle this exception. One way would be to allow the student to re-execute the failed operation. Another way would be to require the examiner to grant appropriate permission to the student to allow retake of the exam. If there are multiple examiners, then any examiner may approve retaking of the exam. Still another way would be to require the particular examiner who gave out the exam to approve a retake.

The exception handling mechanism needs to support exception propagation across different roles as the recovery actions may require participation of members in different roles. Furthermore, there should be a provision to specify whether the exception should be handled by any of the role members or a specific role member. This further requires that each role supports a special interface, i.e. a set of operations, for handling exceptions. This interface would be enabled only for those role members that have to be involved in exception handling.

3.3 Obligation Failure

Obligation failures occur when the participants in a particular task fail to perform the required actions causing the progress of the task to stall. Consider the exam session activity. There might be an obligation that once the exam is started it must be submitted by the student before the alloted time is over. An obligation failure occurs when a student taking the exam does not submit it within the specified time. The handling of such an exception may consist of performing a default action for the obligated role operation, without requiring any user participation. Alternatively, it may also be communicated to some role members for some human assisted recovery.

3.4 Environmental Failure

A task may depend upon external events for its progress. For example, consider a workflow task for car reservation application involving two roles, *Agent* and *Renter*. A task is started when a renter requests a car to be booked. The agent reserves a car for the renter which the renter can pickup at the reservation time. At the reservation time it may happen that the particular rented car is not available because it may not be returned yet by the prior renter or because an accident might have occurred to the car.

Such external events represent error conditions regarding an activity's assumptions about the state of its environment. These errors may be handled in multiple ways. The workflow task can be structured so that it handles such errors by providing an alternate car to the customer without involving the agent or the renter. On the other hand, the customer might want to negotiate the type of the alternate car with the rental facility. This requires restructuring the workflow such that one of the *Agent* role members is involved in assigning an alternate car to the renter.

Appropriate mechanisms are required for translating the external world events to exceptions that represent failure of activity's assumptions about the external physical environment. Mechanisms are required for handling these exceptions without any human intervention. Mechanisms are also required for propagating these exceptions to different roles defined within an activity when human intervention is needed in performing any restructuring of the workflow for recovery.

3.5 Summary of Error Handling Requirements in Context-Aware CSCW Applications

Exceptions occur in three scopes corresponding to: *role operation*, *role*, and *activity*. There is an hierarchical relationship between the three scopes. Activity is the outermost scope. It encapsulates the role scope which encapsulates the role operation scope.

Exceptions occurring in the role operation scope pertain to resource interaction failures and obligation failures. Exceptions pertaining to resource discovery and binding failures occur in the role scope. Exceptions occurring in the activity scope pertain to resource discovery and binding failures and environmental failures.

For developing an exception handling model for performing programmed error recovery in context-aware CSCW applications we need to answer the following questions:

- What mechanisms should be provided for handling exceptions in different scopes?
- What mechanisms should be provided to support exception handling involving role members? Furthermore, how to restrict the exception handler invocation by a specific role member, any role member, or all role members?
- What mechanisms should be provided for propagating exceptions from one role to another role?
- What mechanisms should be provided for restructuring an activity in response to exceptions and external world events?

4 Model for Exception Handling in Context-Aware CSCW Applications

We present the exception model that addresses the requirements identified in Section 3. We extend the programming framework presented in Section 2 with the exception handling model.

- There are three types of exception handlers: (1) those that are attached with role operations and object binding constructs; (2) those that are attached to the role abstraction; (3) those that are provided at the activity-level. Exception handlers attached to the role abstraction require participation from role members for exception handling.
- Exceptions may need to be propagated from the role operation scope to the role scope or may need to be signaled from the activity scope to the role scope. Exceptions may also be propagated from one role to another role.
- An exception encapsulates relevant information about the error occurrence that is essential for exception handling. For example, an exception occurring in the role operation scope, encapsulates the following information tuple: <role name, role member name, role operation name>.

4.1 Exception Handling in the Role Operation Scope

Exceptions in the role operation scope arise due to the failures in executing the associated action or failures corresponding to the non-execution of an obligated operation. An exception handler is statically associated with each role operation, following the *termination model*. The control-flow of the thread executing the operation is automatically transfered from the operation's action to the exception handler associated with the operation.

Exceptions are propagated from the role operation scope to the role scope if there is no exception handler attached with the operation or if the handler encounters an exception or if the handler explicitly signals the exception. A handler may explicitly signal an exception if handling the exception requires some

role member to perform certain actions. The inability to handle the exception completely in the scope of the role operation indicates that it cannot be handled automatically and an external human intervention is required to handle it. Hence the exception is propagated to the role scope and the operation execution context, corresponding to the failed operation, is terminated. Figure 4 shows the exception handling in the role operation scope and role scope.

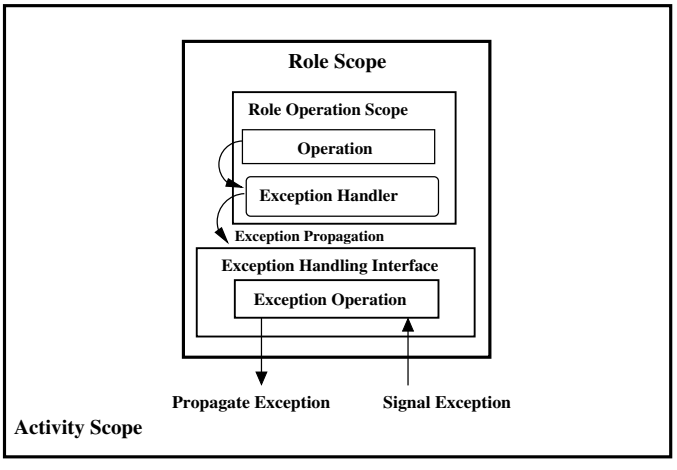

Fig. 4. Exception Handling in the scope of Role

The role operation specification is extended to include the specification of an *exception handler* as shown in Figure 5. The exception handler is specified using the *OnException* clause. The exception handler may perform a sequence of actions specified through the *Action* clause, such as resource interaction, or binding action, or it may signal the exception to the role scope using the *Signal* clause. The signaling target is specified through the *Target* clause. The role that encapsulates the role operation is used as the default target if no *Target* clause is specified.

The role operation specification is extended to incorporate the specification of an *obligation* as shown in Figure 5. The obligation specification consists of the specification of an event whose occurrence after the occurrence of *Event-of-Interest* causes the obligation to fail.

Figure 6 shows the exam session activity in which role operations are provided with exception handlers. The *SubmitExam* role operation is specified to be an *obligation* operation. The obligation exception is handled separately for each role member. The exception handling action consist of automatically performing the action specified as part of the operation (lines 9-12). It may also be propagated to the role scope by re-signaling it. On the other hand, the *WriteExam* operation failure is handled by allowing the student to retake the exam. Correspondingly

Operation *opName*
[**Obligation** {**Before** *Event* **After** *Event-of-Interest*}]
[{**OnException** *exceptionObject* **Type** *ExceptionType*
(**Action** *sequence of actions* |
Signal (*exceptionObject*) **Target** *TargetName*) }]

Fig. 5. Operation Syntax: Modified to incorporate Exception Specification

the *WriteExamFailedException* is propagated to the *Student* role scope through the *Signal* clause (line 16-17).

4.2 Exception Handling in the Role Scope

Exceptions raised in the role scope are those that are propagated from the role operation's scope or signaled from the activity's scope. Role member participation is required for handling exceptions in the role scope. Exceptions may be required to be handled by a specific role member, any role member, or all role members. Each role is provided with an *exception interface*. The exception interface is similar to the role's operation interface in that it provides a set of operations to be performed for exception handling. However, the difference is that the exception interface operation can be executed only when a particular type of exception is delivered to the exception interface. Every exception operation is preceded by a *When* clause that specifies the exception which will enable that operation. The exception interface supports a *queuing model* for exception delivery and handling. Every exception delivered to the exception interface queue is handled separately. In conjunction with enabling the exception interface operation, the delivered exception may also be further propagated to the activity scope if exception handling needs to involve some other role's members.

Consider again the exam session activity specification in Figure 6. An exception, *WriteExamFailedException*, is raised to indicate the failure of *WriteExam* operation. *Student* role is provided with *RetakeExam* operation as part of its exception interface. This operation is enabled when an *WriteExamFailedException* object is delivered to the exception interface of the *Student* role (line 19).

There are two requirements for handling this exception. First, we require that students must receive an approval from the *Examiner* before they can invoke the *RetakeExam* operation. Hence the *WriteExamFailedException* is propagated to the activity scope (line 20) through which it is further propagated to the exception interface of the *Examiner* role. Second, we require that the *RetakeExam* operation be enabled for only those role members who have encountered the failure. Such an access restriction on the invocation of exception interface operations is achieved through the specification of a special qualifier *Invoker* in the *Enable-For* clause (line 21). In our specification model three qualifiers are defined to be used in the *Enable-For* clause. These correspond to *Invoker*, *ANY*, and *ALL*. The qualifier *Invoker* restricts the accessibility of the exception operation to only that role member whose invocation of an operation resulted in

```
1. Activity ExamSession
2.    Object examPaper
      Object Collection StudentAnswers
      Bind examPaper With direct (//ExamPaperURL)
3.    Role Student
4.       Object answerBook
         Bind answerBook With new (//AnswerBookCodeBase)
5.       Operation StartExam
         ...
6.       Operation SubmitExam
7.          Precondition #StartExam.start(invoker=thisUser) > 0
8.          Action Bind answerBook With StudentAnswers
9.          Obligation
10.            Before TimerEvent(3:00:00 hours)
               After StartExam.start(invoker=thisUser)
11.         OnException ExceptionObject Type ObligationFailedException
12.            Action Bind answerBook With StudentAnswers
13.      Operation WriteExam
14.         Precondition #StartExam.start(invoker=thisUser) > 0
15.         Action answerBook.writeExam()
16.         OnException ExceptionObject Type OperationFailedException
17.            Signal (new WriteExamFailedException)
18.      Exception Interface
19.         When ExceptionObject Type WriteExamFailedException
20.            Signal (ExceptionObject)
21.            Enable-For Invoker
22.            Operation RetakeExam
23.               Precondition
                  #Examiner.RetakeApprovedEvent(user=Invoker) > 0
24.                  Action
25.                     Bind answerBook With new (//AnswerBookCodeBase)
26.                     answerBook.writeExam()

27.   Role Examiner
28.      Exception Interface
29.         When ExceptionObject Type ExamInterruptedException
30.            Enable-For ANY
31.            Operation ApproveRetakeExam
32.               Precondition true
33.                  Action NotifyEvent
34.                  (Student, ExceptionObject.getRoleMemberName(),
                     RetakeApprovedEvent)

35.   Reaction HandleOperationFailedException
36.      When ExceptionObject Type WriteExamFailedException
37.      Precondition member(ExceptionObject.getRoleName(), Student)
38.      Signal (new ExamInterruptedException) Target Examiner
```

Fig. 6. Exam Session Activity Specification

raising the exception which is being handled. The qualifier *ANY* allows any role member to perform the exception operation, and the qualifier *ALL* requires all the role members to perform the exception operation.

Thus a *Student* role member can invoke *RetakeExam* operation only when the following two conditions are satisfied. First, the particular student role member must have encountered *OperationFailedException* while performing the *WriteExam* operation. Second, the *Examiner* role member must have explicitly approved retaking the exam for that particular *Student* role member. The second condition is specified as a precondition for the *RetakeExam* operation (lines 22-23). This precondition gets satisfied when the *Examiner* role member notifies the *RetakeApprovedEvent* as part of the exception handling action corresponding to the *ExamInterruptedException* delivered to it by the activity (lines 31-34).

4.3 Exception Handling in the Activity Scope

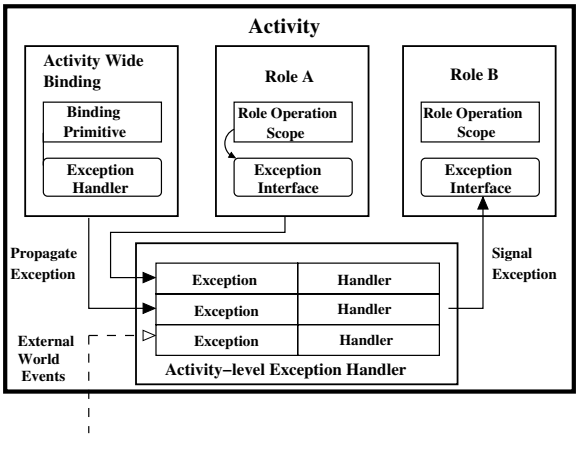

Fig. 7. Activity-level Exception Handling Model

Figure 7 shows the activity-level exception handling model. An activity-level exception handler is modeled as a reaction. There are three kinds of exceptions that occur in the activity scope: (1) the exceptions corresponding to the abnormal events occurring external to the activity; (2) exceptions occurring due to binding failures corresponding to the activity-level resource binding requirements; (3) exceptions that are propagated from the role scope.

In Figure 6, the activity-level exception handler for the *ExamSession* activity is shown in lines 35-38. This activity-level exception handler signals *ExamInterruptedException* to the *Examiner* role if the *WriteExamFailedException* occurred as part of some *Student* role member executing the *WriteExam* operation. The exception interface specification for the *Examiner* role is shown in lines 27-34 in Figure 6. The *ApproveRetakeExam* operation can be performed by any member

of the *Examiner* role. This is specified through the qualifier *ANY* meaning that any member of the *Examiner* role can perform this operation. The *Examiner* role member uses the *ApproveRetakeExam* operation for approving retaking of the examination.

There can be multiple *Student* role members who may have encountered failures while executing the *WriteExam* operation and may require appropriate approval. A *RetakeApprovedEvent* is generated corresponding to each such failure and is notified to the appropriate *Student* role member. This event causes the precondition of the *RetakeExam* operation in the exception interface of the *Student* role to become true (lines 23 of Figure 6) for that role member.

An activity-level exception handler may seem similar to the Guardian like exception handler [10] for concurrent object-oriented systems. However there is a crucial difference between the two. The guardian model of exception handling is suited for timed asynchronous systems. The guardian handles an exception by suspending all the participant processes and signaling appropriate exceptions to the processes. Each process handles the exception according to its execution context. CSCW applications are loosely coupled and asynchronous in nature. Exception handling in such applications may require participation from human users. Also, the generated exceptions may not be relevant immediately. Their effect may be felt by the application at some later stage. This is unlike synchronous systems where exceptions affect the immediate execution of the application and hence the exceptions need to be dealt with immediately.

The exception propagation model presented here loosely resembles the one developed for mobile agent systems [11,12]. A scope defines a logical space of roles through which agents coordinate with one another in [11]. Exceptions arising in inter-agent coordination are propagated to all the agents present in the scope. In [12] a specific agent is designated as the exception handling agent to which, all the exceptions arising in an agent's context, are propagated.

4.4 Discussion

Our exception propagation model does not propagate exceptions along the dynamic call chain. Instead, exceptions are propagated along the handlers that are statically associated [13] with different scopes. We use this approach for the following reasons:

- Exception propagation along the handlers that are statically associated with different scopes allows determination of the complete exception handling path for an exception at the activity design time.
- Restructuring of an activity, as part of the exception handling actions, is specified at the activity design time. Such restructuring actions are relevant only in the scope where an exception occurs and hence such actions should be performed in the exception handlers associated with that scope.
- Propagating exceptions arising in the role operation scope along dynamic call chain would mean propagating them directly to the role member who invoked the operation. This is not appropriate in our role-based framework

where users hold only as much privileges as are provided to them by the role abstraction. Users may not have sufficient privileges or mechanisms for handling exceptions that are directly propagated to them.

Concurrent exceptions may occur in an activity. We use a simple queuing model which ensures that all the concurrent exceptions are handled in some serial order. For example, multiple students may encounter failures while executing the *WriteExam* operation. All the corresponding *WriteExamFailedExceptions* are delivered to the exception interface queue of the *Student* role where they are handled separately for each *Student* role member. Our current exception handling model does not support exception resolution.

All the exceptions requiring participation of members from multiple roles are propagated through the activity scope. This is a conscious design decision. The set of roles that can participate in handling a particular exception can be changed easily in a design by altering the activity-level exception handler for that exception type.

5 Examples of Exception Handling in CSCW Applications

In this section we present modeling examples of two context-aware CSCW applications. For each of these, we identify different error conditions and show how these errors are programmatically addressed through the exception handling model presented in Section 4.

5.1 Case Study 1: Car Rental Activity

The purpose of this example is to demonstrate handling of exceptions arising due to external world events. Consider the car rental activity consisting of an *Agent* role and a *Renter* role. A renter can reserve a car to be picked up at some time. One of the error conditions in this workflow corresponds to the case where the car is involved in an accident or has encountered a mechanical failure. Such an error condition is handled as follows. An alternate car is searched and given to the renter, if available. This action may fail if no alternate car is available. In that case, the agent needs to be involved for providing alternate car by discussing with the renter.

The specification of this activity is shown in Figure 8. The renter can book a car through the *BookCar* operation (line 6). The *carBooking* object is used to describe the requirements of the car. It is declared to be of type *carRDD* which is a *RDD* of the car. A car satisfying the requirements specified in the RDD is discovered from the car database and assigned id of the renter (lines 4-8).

The activity-level exception handler is defined as a reaction *CarFailureReaction* (line 16). This reaction gets triggered when it receives *CarFailedEvent* which is an external event. The *carRDD* (not shown) specifies that this event is imported from a *car* resource. This event is generated when there is any failure associated with a car such as an accident.

```
1. Activity CarRental
2.    Object carBooking RDD carRDD
3.    Object carDatabase
4.    Role Renter
5.       Bind carBooking With discover (category=<car-category>)
6.       Operation BookCar
7.          Precondition true
8.          Action carDatabase.reserveCar(carBooking, bookingId=thisUser)
9.    Role Agent
10.      Exception Interface
11.         When ExceptionObject Type BookedCarFailureException
12.            Enable-For ANY
13.            Operation HandleCarRelatedFailure
14.               Precondition true
15.                  Action // discuss with the renter
16.   Reaction CarFailureReaction
17.      When ExceptionObject Type CarFailedEvent
18.      Precondition true
19.      Action
20.         carDatabase.provideAlternateCar(CarFailedEvent.getBookingId())
21.         OnException ExceptionObject Type
            AlternateCarNotAvailableException
22.            Signal (new BookedCarFailureException) Target Agent
```

Fig. 8. Car Booking Activity Specification

The exception handling action consists of providing an alternate car to the renter (lines 17-20). This action may also encounter failure. In that case, *BookedCarFailureException* is signaled to the *Agent* role (lines 21-22). This exception enables the *HandleCarRelatedFailure* operation provided as part of the exception interface of the *Agent* role. Any member of the *Agent* role can handle this event. This is specified through the qualifier *ANY* for the *Enable-For* clause (line 12).

5.2 Case Study 2: Museum Infodesk Activity

The purpose of this example is to demonstrate handling of exceptions related to resource discovery/binding and resource interaction.

Consider the museum infodesk activity where a *Visitor* role member discovers and binds to the audio commentary object of an artifact based on the user's location. We consider two representative error conditions that can arise in such context-based discovery and binding. First type of errors correspond to the case where audio commentary is not available in the user's desired language. This failure occurs because there is a mismatch between the resource specification and available resources. It can be handled by changing some aspects of resource requirements and retrying resource discovery. Alternate resource specifications may be specified at the activity design time. The museum infodesk activity can bind with an audio commentary resource in another language.

```
1. Activity Museum Infodesk
2.    Object locService RDD Location-Service-Description
3.    Bind locService With direct(//LocationServiceURL)
4.    Object audioChannel RDD Audio-Channel-Description
5.    Object textInterface RDD Text-Interface-Description
6.      Role Visitor
7.          Bind audioChannel When LocationChangeEvent(thisUser)
8.          With discover(<location=locService.getLocation(thisUser),
9.          language="ENGLISH">)
10.         OnException ExceptionObject Type ResourceDiscoveryException
11.                         Action Bind audioChannel
12.                         With discover
13.                         (<location=locService.getLocation(thisUser),
14.                         language="SPANISH">)
15.         Operation ListenAudio
16.                         Precondition true
17.                         Action audioChannel.listenAudio()
18.   Exception Interface
19.         When ExceptionObject Type AudioServiceFailedException
20.         Enable-For Invoker
21.         Operation ReadText
22.                         Precondition true
23.                         Action Bind textInterface With Direct
24.                         (//textInterfaceURL)
25.                         textInterface.readText()
```

Fig. 9. Museum Infodesk Activity Specification

Second type of errors correspond to the case where audio commentary object encounters a failure while the user is listening to the commentary. As part of exception handling, the museum infodesk activity may bind to the textual commentary resource and present the appropriate interfaces to the role member.

The specification of this activity is shown in Figure 9. The activity consists of a *Visitor* role. The objects corresponding to audio interface (*audioChannel*) and text interface (*textInterface*) are declared in the activity scope. Binding of the *audioChannel* object is performed whenever an event indicating a change in the *Visitor* role member location is delivered (lines 7-9). A resource that matches *Audio-Channel-Description* for the new location and which provides commentary in English language is searched. This discovery and binding action fails if the audio commentary is not available in English. An exception, *ResourceDiscoveryException*, is raised to denote this failure. The recovery action consists of automatically retrying the discovery process by changing the requirement from English language to Spanish language (lines 10-14).

The *Visitor* role member is provided *ListenAudio* operation (line 15) through which the role member can listen the audio commentary. This operation encounters a *AudioServiceFailedException* if the audio object fails for some reason. No exception handler is attached to this operation. Hence this exception is

propagated to the *Visitor* role's exception interface (line 18-25). An operation *ReadText* is defined (line 21) in the exception interface which gets enabled on the receipt of *AudioServiceFailedException*. This operation allows the role member to bind and use the textual interface of the artifact.

6 Related Work

Our work is related to research concerning exception handling in workflow systems [14,15,16,17,18]. In [14] a model for workflow failure handling that includes both forward error recovery based on exception handling and backward error recovery based on the notion of atomic workflow tasks is presented. Exceptions generated in a subtask are propagated to its encapsulating task and are handled there. Our exception handling model differs from this model along two aspects. First, exception handlers are directly associated with role operations, roles, and activities. Second, the model in [14] propagates exceptions along the task invocation hierarchy. Our exception propagation model allows exception propagation across roles. Such inter-role exception propagation is required because of the asynchronous and multi-user nature of CSCW applications.

A language for exception modeling based on event-condition-action (ECA) paradigm, independent of any particular workflow system, is presented in [15]. Exception rules are compiled from this language and integrated with the underlying workflow system. In contrast to this, exceptions and exception handlers are directly integrated with the activity specification in our approach.

A uniform framework for addressing data and process irregularities in workflow systems based on context-sensitive exception handling is presented in [16]. In their model, *unanticipated* exceptions are handled by human agents. Thus human assistance is enlisted as a last failure handling option. Human involvement in exception handling is an intrinsic part of our model. Exception interfaces are associated with roles and these interfaces are enabled only for those role members that need to participate in exception handling.

Failures arising in pervasive computing environments are considered in [19]. The failure categories presented in [19] correspond to *device failures*, *application failures*, *network failures*, and *service failures*. The resource interaction failure defined in this paper can effectively model application, network and service failures. We also consider errors arising due to failures in context-aware resource discovery and binding, obligation failures, and errors corresponding to the external world events that represent violations of application's assumptions about the external world. They propose failure handling through heart-beat based status monitoring, redundant provisioning of alternate services/applications, and restarting failed applications. In contrast to this, we present application level exception handling mechanisms for programmed error recovery in such applications.

The interference issues arising in software services deployed in home environments are considered in [20]. Interference can be considered as a form of failure occurring in concurrent resource access in home environments. A resource access

model based on locking primitives for preventing such interferences is presented in [20]. Such concurrent resource access can be modeled as error conditions corresponding to the external world events that represent resource interference.

7 Conclusions

In this paper we have presented an exception handling model for programmed error recovery in context-aware CSCW applications. The salient feature of this model is the exception interface abstraction through which role members can participate in exception handling. The exception propagation model consists of propagating exceptions along the handlers that are statically determined based on the exception occurrence scope. Our earlier specification model [5] is extended to incorporate the exception handling model. Capabilities of the exception handling model are demonstrated through three different context-aware CSCW applications.

References

1. MIT: (Project Oxygen) Available at url http://oxygen.lcs.mit.edu/.
2. Satyanarayanan, M.: Pervasive Computing: Vision and Challenges. IEEE Personal Communications **8** (2001) 10–17
3. Schilit, B., Adams, N., Want, R.: Context-Aware Computing Applications. In: IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, US (1994) 85–90
4. Tripathi, A., Kulkarni, D., Ahmed, T.: A Specification Model for Context-Based Collaborative Applications. Elsevier Journal on Pervasive and Mobile Computing **1** (2005) 21 – 42
5. Tripathi, A., Ahmed, T., Kumar, R.: Specification of Secure Distributed Collaboration Systems. In: IEEE International Symposium on Autonomous Distributed Systems (ISADS). (2003) 149–156
6. Tripathi, A., Ahmed, T., Kumar, R., Jaman, S.: Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS). (2002) 393 – 400
7. RDF: Resource Description Framework (RDF) (1999) Available at url http://www.w3.org/RDF/.
8. WSDL: Web Services Description Language (WSDL) 1.1 (2001) Available at url http://www.w3.org/TR/wsdl.
9. Randell, B., Lee, P., Treleaven, P.C.: Reliability Issues in Computing System Design. ACM Comput. Surv. **10** (1978) 123–165
10. Miller, R., Tripathi, A.: The Guardian Model and Primitives for Exception Handling in Distributed Systems. IEEE Transactions on Software Engineering **30** (2004) 1008 – 1022
11. Iliasov, A., Romanovsky, A.: CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents. In: ECOOP Workshop on Exception Handling. (2005)

12. Tripathi, A., Miller, R.: Exception Handling in Agent-Oriented Systems. In: Advances in Exception Handling Techniques, LNCS 2022, Berlin Heidelberg, Springer-Verlag (January 2001) 128–146
13. Knudsen, J.L.: Better Exception-Handling in Block-Structured Systems. IEEE Software **4** (1987) 40 – 49
14. Hagen, C., Alonso, G.: Exception Handling in Workflow Management Systems. IEEE Transactions on Software Engineering **26** (2000) 943–958
15. Casati, F., Ceri, S., Paraboschi, S., Pozzi, G.: Specification and Implementation of Exceptions in Workflow Management Systems. ACM Trans. Database Syst. **24** (1999) 405–451
16. Murata, T., Borgida, A.: Handling of Irregularities in Human Centered Systems: A Unified Framework for Data and Processes. IEEE Transactions on Software Engineering **26** (2000) 959–977
17. Li, J., Mai, Y., Butler, G.: Implementing Exception Handling Policies for Workflow Management System. In: Software Engineering Conference, 2003. Tenth Asia-Pacific. (2003) 564 – 573
18. Chiu, D.K.W., Li, Q., Karlapalem, K.: Exception Handling with Workflow Evolution in ADOME-WFMS: a Taxonomy and Resolution techniques. SIGGROUP Bull. **20** (1999) 8–8
19. Chetan, S., Ranganathan, A., Campbell, R.: Towards Fault Tolerant Pervasive Computing. IEEE Technology and Society **24** (2005) 38 – 44
20. Kolberg, M., Magill, E., Wilson, M.: Compatibility Issues Between Services Supporting Networked Appliances. IEEE Communications Magazine **41** (2003) 136 – 147