Application-Level Recovery Mechanisms for Context-Aware Pervasive Computing*

Devdatta Kulkarni and Anand Tripathi
Department of Computer Science
University of Minnesota Minneapolis, 55455 Minnesota USA

Abstract

We identify here various kinds of failure conditions and robustness issues that arise in context-aware pervasive computing applications. Such conditions are related to failures in an application's interactions with ambient services, failures in resource discovery and binding, and invalidation of context conditions during the execution of an application task. In this paper we present an exception handling model for integrating forward error recovery mechanisms in the designs of such applications. This model is integrated in a role-based framework and supported by a programming environment for construction of such applications.

1 Introduction

Context-awareness is a central aspect of pervasive computing applications, characterizing their ability to adapt and perform tasks based on the ambient context conditions. Typical examples of context include a person's location, proximity of people, proximity of a person to a device or an object, devices being used by a person, the activity in which a person is engaged in, etc [15]. The computing environments for supporting context-aware applications provide services for context information management, resource discovery, location-independent naming, and authorization and access control.

We designed and implemented a role-based programming framework for building such context-aware applications in which multiple users may be involved in collaborative activities [16, 17, 11]. The function of a *role* [14] in this framework is to represent a set of privileges for users to execute application tasks. One of the important characteristics of context-aware applications is their ability to adapt under changing context conditions. Context-based adaptation requires dynamic reconfiguration of the application to integrate resources and services based on the current context conditions. This programming framework provides mechanisms through which an application may discover and bind to different services in the environment under different context conditions. Using this framework we designed and implemented a number of context-aware applications, which include a single-user context-aware music player, a multi-user context-aware patient information system [11], a context-aware distributed meeting [16], a context-aware exam session [17], and an emulation of a context-aware museum information system.

The focus of this paper is on the robustness issues that we encountered in designing and implementing these applications. Our initial focus while building the programming framework was on providing appropriate mechanisms for designing context-based adaptive features. However, our experiences with the deployed applications revealed various robustness issues that arise due to the dynamic nature of such applications and also due to the dynamic nature of the environments in which the applications are deployed. We realized that the absence of application-level programmed error recovery mechanisms within this framework led to fragile applications, which were unable to cope with various failure conditions. We experienced the following three broad classes of robustness issues:

- The dynamic reconfiguration mechanisms integrated in an application for context-based adaptation can themselves become a cause of robustness problems, if not properly designed. In some situations an application may fail to function correctly due to failures in finding the required resources and services during a reconfiguration. Moreover, the order of binding various services to an application during reconfigurations and concurrent processing of context events can affect the correctness of the application.
- Various kinds of failures can arise during the users' interactions with the services bound to the application. These include failures due to network disruptions, service crashes, and access revocations by services, which may lead to disruptions of any interactive user sessions. A reconfiguration action could also disrupt an ongoing interactive session,

* This work was supported by National Science Foundation grants 0411961 and 0708604

causing it to terminate prematurely.

• An application that requires a task to be executed only while some specified ambient context conditions hold is prone to failures when such conditions are violated. We refer to this as the *context invalidation problem* [11]. This requires mechanisms to be provided for an application to monitor such context invalidation conditions and perform appropriate corrective actions when they arise. In general, we need mechanisms to deal with the physical world events that violate an application's assumptions about the external world situations.

As a primary contribution of this paper we present here mechanisms for forward error recovery based on the integration of an exception handling model with an asynchronous event communication model for building robust context-aware applications. The exception handling model is based on the framework presented in [17]. We integrate in this model asynchronous event communication mechanisms for supporting event-driven recovery actions. Our extended model also provides mechanisms for handling context invalidation conditions. The model also supports user participation in performing recovery from failures that cannot be handled automatically. Our model has similarities with workflow exception handling models [2, 9] where event-triggered rules have been used as a mechanism for performing exception handling actions.

The paper is organized as follows. In Section 2 we present our programming model for context-aware applications. In Section 3 we identify the robustness issues and failure modes in such applications. In Section 4 we present the forward recovery mechanisms. In Section 5 we compare our work with the related work, and conclude in Section 6.

2 Programming Framework Overview

A context-aware application is programmed using an abstraction called *activity*. An activity defines a namespace for *roles*, *objects*, and *reactions*. The *object* abstraction is provided in the activity for accessing various resources and services required by the application. An object may be bound to different services under different context conditions. Objects defined in an activity's namespace are *shared* by all the roles defined in the activity.

Each role defines a namespace for objects and role operations. Objects defined within the scope of a role are *private* to that role; a separate instance of such an object is created for each role member. Such objects are required because within a multi-user application we may want different members of a role to access different instances of a service type based on their individual context. A role operation represents a task that is explicitly invoked by the role members. A role operation consists of two parts: a *precondition* and an *action*. A role operation precondition must be satisfied before the action can be executed. For coordination purpose, two types of events, *start* and *finish*, are defined for a role operation. An operation's precondition may be based on predicates involving counts of *start/finish* events of various role operations, role membership predicates, and predicates involving ambient conditions queried from context services. An operation's action starts an interactive *session* as part of which a set of methods may be invoked on a *shared* or a *private* object. For each user, an interface component called *User Coordination Interface (UCI)* is dynamically created and transported to that user's device. Through the UCI, a user communicates with a role manager for executing the role operations.

In Figure 1 we present the role-based user–service interaction pattern in the context-aware applications designed using our framework. A user accesses a service by executing a role operation. The role manager evaluates the precondition associated with the invoked operation. If the precondition is true, the role manager invokes the specified method on the object specified as part of an operation's action. In turn, the corresponding object manager invokes that method on the currently bound service.

Context-triggered actions are programmed as *reactions* within an activity. A reaction is similar to a role operation – it consists of a precondition and an action – the only difference being that it is triggered by an event and executed by the runtime system, rather than executed by any role member. Specifically, a *reaction* mechanism is used for two purposes in our programming framework. First, a reaction is used to perform automated task executions that do not involve user participation. Second, a reaction is used to program an object's context-triggered binding policies.

In the middleware, three generic components are provided - an *activity manager*, a *role manager*, and an *object manager*. The runtime environment of an application is constructed by specializing these managers based on the application's specification. Separate object managers are created for objects defined in the activity's namespace, and for those defined in a role's namespace. Each such object manager maintains a reference to the service to which it is currently bound. Context-triggered reactions defined for an object for dynamic binding are executed by its object manager. A role manager enforces the context-based access control policies using the specified preconditions for its operations. Additionally, it also enforces role admission control policies. These managers subscribe to the appropriate context events from the context services present in the environment. All the managers are executed on a set of trusted servers in the environment.

Testbed Environment: We programmed five context-aware applications using this framework, as noted in Section 1. Except for the music player application, all others are multi-user applications. For the purpose of our discus-

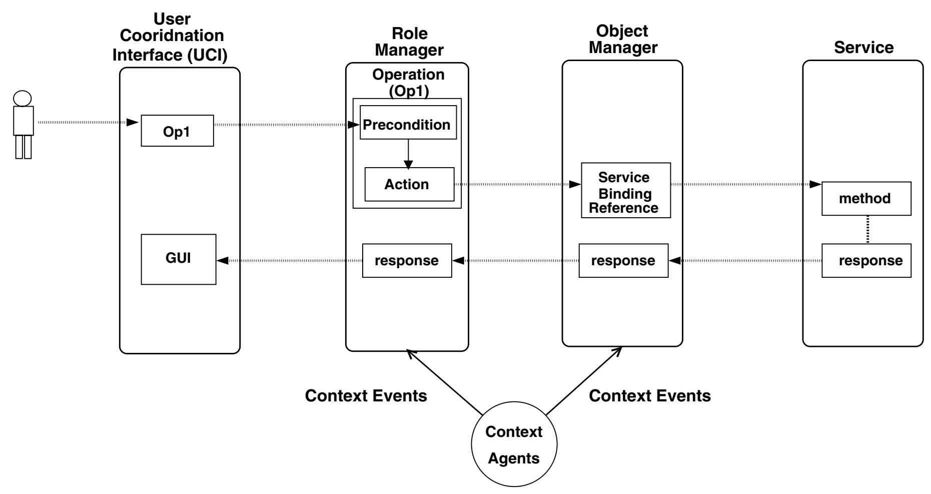

Figure 1. The Model of User–Service Interactions

sion and illustration of issues here, we use two of these applications. These include a context-aware patient information system and a context-aware music player.

We developed and deployed a resource discovery service, and *context agents* to aggregate sensor data and provide context information. Services register with the discovery service, and applications query the discovery service to find the required service matching a specific discovery criteria. To describe a service we have developed an XML schema called Resource Description Definition (RDD) in our programming framework. An RDD specifies three things - a list of attribute-value pairs for a service, service interfaces, and events that may be generated by a service and which can be subscribed to by a context-aware application. The applications designed using our framework use the *parameterized RDD* as part of the discovery process. Certain attributes in this RDD are specified at runtime, and may be based on the context information.

An agent in our framework maintains context information and generates context events. For example, one may define such an agent corresponding to each room in a building. It maintains the status information about its room, such as the list and the number of users in the room. Users' Bluetooth enabled devices, such as laptops and PDAs, are used to detect user presence in a room. It generates the *RoomStatusChangeEvent* when a person arrives or leaves the room. A *Location Service Agent* is deployed in the system for maintaining the location information for each user. For this, it subscribes to the *RoomStatusChangeEvent* from each room agent in the system. It generates the *UserArrival* and *UserDeparture* events indicating a user's arrival in or departure from a room, respectively. An application may subscribe to these events and adapt its behavior. Moreover, each user's device may also run an agent that uses an RFID reader installed on the device for detecting the user's proximity to entities that are attached with RFID tags.

Example 1 – Context-Aware Patient Information System: The purpose of developing this testbed application was to test and evaluate the capabilities of our programming framework for building multi-user collaborative applications. This application consists of a *Nurse* role and a *Doctor* role. It emulates a patient information system that is accessed by nurses and doctors, supporting storage, retrieval, and access of patient information records. Additionally, it maintains the doctor's confidential reports about each patient. Doctors and nurses may access this information through their mobile personal devices. Here we consider the following context-based information access requirements for this application. A nurse is allowed access to the doctor reports for a patient only when some doctor is also present in the same ward where that nurse is located. A nurse is allowed to access a specific patient's information record while she is attending that patient.

In Figure 2 we present the partial activity specification for this application in a pseudo notation. In this notation the terms in boldface represent the XML tags. In the activity specification we define a *Nurse* role, a *Doctor* role, and a *LocationService* object in the activity's scope. The *LocationService* object is bound permanently to the *Location Service Agent* running at a known URL in the testbed environment (line 2). From this agent, the activity imports the *UserArrivalEvent* for the users admitted to the *Nurse* and *Doctor* roles. In the *Nurse* role we define the following private objects: *RFIDService* (line 4) , *WardRoomAgent* (lines 5-9), *DoctorReports* (lines 10-15), and *PatientInformation* (lines 16-20). The purpose of the *RFIDService* object is to bind to the context agent running on each nurse's device through which the application can detect the nurse's proximity to a patient. The purpose of the *WardRoomAgent* object is to dynamically bind to the context agent cor-

```
1.Activity PatientInformationSystem {
2.  Object LocationService { Bind Direct (//LocationServiceAgentURL) Import_Event UserArrivalEvent }
3.  Role Nurse {
4.      Object RFIDService { Bind Direct (//LocalDeviceRFIDService) Import_Event RFIDEvent }
5.      Object WardRoomAgent RDD (//RoomRDD.xml) {
6.          Reaction {
7.              When Event (UserArrivalEvent.getUserName() == thisUser)
8.              Action Bind Discover (LOCATION=LocationService.getLocation(UserArrivalEvent.getUserName())) }
9.      } // end of WardRoomAgent object definition
10.     Object DoctorReports RDD (//ReportRDD.xml) {
11.         Reaction {
12.             When Event (UserArrivalEvent.getUserName() == thisUser)
13.             Precondition WardRoomAgent.isPresent(thisUser) ∧ WardRoomAgent.isPresent(members(Doctor))
14.             Action Bind Discover (LOCATION=LocationService.getLocation (UserArrivalEvent.getUserName())) }
15.     } // end of DoctorReports object definition
16.     Object PatientInformation RDD (//PatientInformationRDD.xml) {
17.         Reaction {
18.             When Event RFIDEvent
19.             Action Bind Discover(PATIENT-ID=RFIDEvent.getID()) }
20.     } // end of PatientInformation object definition
21.     Operation AccessDoctorReports {
22.         Precondition WardRoomAgent.isPresent(thisUser) ∧ WardRoomAgent.isPresent(members(Doctor))
23.         Action DoctorReports SessionMethod accessReports
24.     } // end of AccessDoctorReports operation
25.     Operation AccessPatientInformation {
26.         Action PatientInformation SessionMethod access
27.     } // end of AccessPatientInformation operation
28. } // end of Nurse role
29. Role Doctor { ... }
30.} // end of Activity specification
```

Figure 2. Context-Aware Patient Information System Activity Specification

responding to the ward where a nurse is currently present. A reaction, triggered by a *UserArrivalEvent*, is defined to dynamically bind the object to the appropriate room's agent (lines 6-8). We define a *RoomRDD* as a *parameterized RDD*. The *LOCATION* attribute in this RDD is a *parameterized attribute*. The value of this attribute is specified at runtime using the location information of the role member corresponding to whom the *UserArrivalEvent* is generated.

The purpose of the *DoctorReports* object is to bind to the doctor reports. A reaction, triggered by the *UserArrivalEvent*, is defined for this purpose (lines 11-14). We want this object to bind to doctor reports only if there is some doctor present in the ward where the nurse is currently present. This requirement is specified as the precondition of the binding reaction. In the context of a private object's binding, *thisUser* is a special variable which refers to the identity of the role member with whom the object is associated. In our programming framework *members(RoleName)* is a system-defined function providing the list of users admitted to the role *RoleName*. The purpose of the *PatientInformation* object is to bind to a specific patient's medical record. Its binding reaction is triggered by an *RFIDEvent* (lines 17-19). The *Nurse* role is provided with the *AccessDoctorReports* operation through which a nurse can access doctor's reports (lines 21-24). As part of this operation's precondition, the role manager checks whether the nurse who is accessing the reports and some doctor are both present in the same ward. The *thisUser* variable, when used as part of a role operation, translates to the identifier of the role member who is invoking the role operation. A nurse is able to execute the *accessReports* method as part of this session. For accessing the patient records, the *AccessPatientInformation* operation is provided to the *Nurse* role (lines 25-27).

Example 2 – Context-Aware Music Player: This is a single-user application. It runs on a person's mobile device and supports the following context-aware features. When the application starts, it streams music to the audio player service on the user's device. When the user enters a room, it discovers and binds to the room's audio player service and starts streaming music to it only if no other person is present in the room. When the user leaves the room, or when some other person enters the room, it binds to the audio player service running on the user's device and continues streaming music to it. In this activity we define objects to represent the music source and the audio sink. The audio sink object binds to the audio player service running either on the user's device or the one running in the room where the user is currently present. The context information required by this application includes the user's arrivals in and departures from different rooms, arrivals and departures of other people in the room where the user is present, and the number of people present in a room.

3 Robustness Issues and Failure Modes

The robustness issues that we encountered in the testbed applications are related to concurrent processing of context events by object managers, failures encountered by object managers in discovering the required resources and services in the environment, failures encountered during a user's interaction with a service, and changes in context conditions that are required to hold during active sessions.

3.1 Robustness Issues in Context-based Reconfigurations

1) Binding Order: Multiple objects defined in a context-aware application may be programmed to modify their bindings when a specific context event is delivered to the application. The order in which the various object managers process such an event is crucial for the correctness of the application behavior. We present here the problem that occurred due to this in the patient information system. In this application the binding of the *WardRoomAgent* object and the *DoctorReports* object is triggered by the *UserArrivalEvent*. Occasionally, this application failed to bind the *DoctorReports* object. This happened whenever the binding reaction of the *DoctorReports* object was triggered before the binding of the *WardRoomAgent* object by a *UserArrivalEvent*. The *DoctorReports* object was left unbound, as its object manager could not query the *WardRoomAgent* object for the purpose of precondition evaluation, since that object was not bound by that time.

To address the above issue we provided a construct in the programming framework through which one can specify the order of dispatching a context event to different object managers. For example, using this construct one can specify that on the occurrence of the *UserArrivalEvent* the binding reaction of the *WardRoomAgent* object should be executed prior to executing the binding reaction of the *DoctorReports* object.

2) Concurrent Processing of Context Events: An object defined in a context-aware application may be programmed to execute different binding reactions on the occurrence of different context events. If such events occur concurrently, then sequential processing of these events by the object manager is crucial for correct application behavior. We encountered this problem in the music player application. One of the requirements for this application is that when some other person enters the room, the application should stop streaming music to the room's audio player service and start streaming it to the user's device. Occasionally,

```
Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
    Reaction BindToRoomSpeakers {
        When Event UserArrivalEvent
        Precondition CurrentRoomAgent.presentUserCount() == 1
        Action Bind Discover (LOCATION=
        LocationService.getLocation(UserArrivalEvent.getUserName())) }
    Reaction BindToDeviceSpeakers {
        When Event RoomStatusChangeEvent
        Precondition CurrentRoomAgent.presentUserCount() > 1
        Action Bind Direct (//DeviceAudioPlayerURL) }
} // end of AudioPlayer object
```

Figure 3. AudioPlayer object specification

the application failed to satisfy this requirement. In Figure 3 we present the specification of the *AudioPlayer* object used in this application. Here the *CurrentRoomAgent* object represents the agent of the room in which the user is present. Consider the case where the object manager receives the *RoomStatusChangeEvent* while it is processing the *UserArrivalEvent*. The processing of the *BindToDeviceSpeakers* reaction gets initiated before the processing of the *BindToRoomSpeakers* reaction is completed. Specifically, *BindToDeviceSpeakers* may get initiated after *BindToRoomSpeakers* finishes checking that there is only one person present in the room but *before* it had executed the binding action. *BindToDeviceSpeakers* causes the object to bind to the audio player service running on the user's device because there are more than one persons in the room. However, the object is bound to the audio player service of the room when the binding action of *BindToRoomSpeakers* is completed.

In order to avoid such problems, we modified the event handling model of the programming framework to ensure that every object manager sequentially executes the binding reactions. This guarantees that a reaction's precondition evaluation and action execution are performed *atomically*.

3) Discovery Failures: During a context-triggered reconfiguration action, an application may fail to discover the required resources and services due to the dynamic, ad hoc, and autonomous nature of such computing environments. This motivated us to include mechanisms for application-defined recovery actions to discover and bind to alternate services when such failures occur.

4) Object Bindings during Active Sessions: An object defined in a context-aware application may undergo binding changes due to changes in the application's context while one or more user sessions are currently active with the service to which the object is currently bound. Such sessions get disrupted when the object's binding changes. For certain applications such disruptions can lead to failures. We encountered this situation in the patient information system when a nurse's interactive session for accessing a patient's information records got disrupted when the nurse moved into the vicinity of another patient. A context event indicating the nurse's proximity to the second patient caused the *PatientInformation* object's binding to change, disrupting the nurse's active session.

This application requires that the binding of the *PatientInformation* object should not be changed while some user-session is active. However, one can also find several applications where an object's binding needs to be continually changed based on changing context conditions, irrespective of whether any sessions are active with the currently bound service. An example is an application that uses a person's current location to continuously provide information about the current traffic conditions in the neighborhood by binding to the appropriate traffic information service in the area where the person is currently located.

Both such requirements can be programmed in our model through the use of the precondition mechanism associated with the binding reactions. Using the precondition mechanism, a binding action can be disabled when a session is active. Moreover, a role operation session can also indicate to the object manager whether it is *tolerant* to changes in an object's bindings. For sessions that are not tolerant, the object manager would signal the *ObjectUnboundEvent* to the role manager when the object's binding changes. The role manager raises this event as an exception in the context of the terminated session. An exception handler could be defined with the role operation for some alternate action. For recovery from such interrupted sessions, it is the responsibility of each service to define the appropriate transaction-oriented recovery mechanisms.

3.2 Robustness Issues in User-Service Interactions

A user's interactions with a bound service can be affected due to disruption of the object's binding with the service, access revocation by the service, or exceptions thrown by the service. Certain kinds of exceptions., such as access revocations and binding failures, can be potentially handled by the object manager. For example, the object manager can try to rebind to the same or a different service. Event-driven reactions for recovery can be defined within an object's specification. If a recovery by the object manager is not possible or fails, then an exception is propagated to the role operations. Any exceptions thrown by the service as part of a role operation invocation need to be communicated to the role for handling in the context of that operation.

3.3 Robustness Issues in Context Invalidations

We encountered the context invalidation problem in both patient information system and music player application. In the context-aware patient information system, we require that a nurse should be able to access doctor reports only while some doctor is also present in the ward where that nurse is located. This condition was violated when either the nurse left the ward or no member of the doctor role remained in the ward. In the context-aware music player application we required that the application should stream music to the audio player service in a room only during the absence of other people in that room. This condition was violated when some other person walked into the room while the music was being streamed to the room's audio service.

For correct enforcement of such requirements, evaluating such context conditions only as part of the operation's precondition is not sufficient. Such context conditions need to be continually monitored during the operation execution. When any context invalidations occur, the corresponding context sensitive task needs to be terminated. To deal with such terminations, the application should be able to execute some alternate plan. For example, the patient information system could be adapted to provide the *Nurse* role with an operation through which a nurse could request the doctor role for permission to access the doctor reports when a context invalidation occurs. Upon receiving a doctor's approval, the nurse could then continue accessing the reports.

4 Forward Error Recovery using Exceptions and Events

We present here an exception handling model integrated with a model for asynchronous event communication for implementing activity-wide forward error recovery mechanisms. Events represent a broad class of conditions related to an activity. These include both normal and failure conditions. Events are communicated to different entities in an activity through the publish-subscribe mechanisms. Exceptions represent a subclass of events that arise in the context of executing an action as part of a role operation or a reaction. In our framework, exception handling is based on the termination model. An exception handler can be attached to an *action* and it may contain one or more actions, including signaling of an application-defined event based on the asynchronous communication model. The propagation of an exception is defined according to a set of rules, which we present in this section. If no handler is specified for an action, the thread of execution that encountered the exception is terminated, and the exception is propagated to the outer scope, which corresponds to the object-scope for the binding reactions, and the role-scope for role operations.

4.1 Object-level Model for Handling Events and Exceptions

In Figure 4 we present the object-level event model. In this model, the following are the system-defined

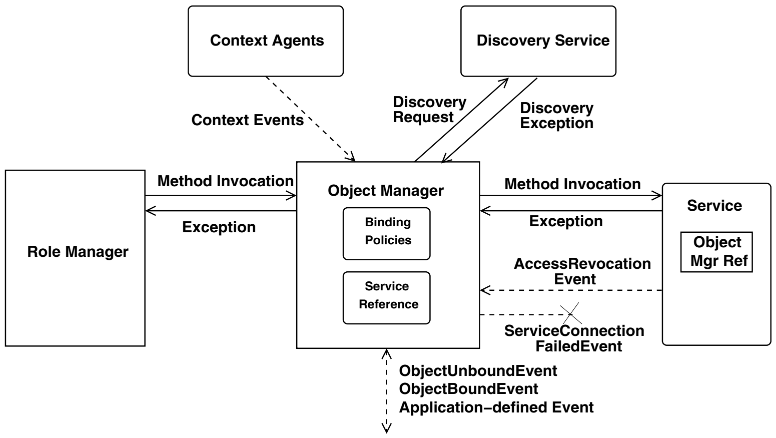

Figure 4. Object-level Event Model

events and exceptions: context events, *DiscoveryFailure-Exception*, *ServiceConnectionFailedEvent*, *AccessRevocationEvent*, *ObjectUnboundEvent*, *ObjectBoundEvent*.

Context events are used to trigger context-based binding reactions within an object. The *DiscoveryFailureException* is raised by the discovery service when it fails to find a service matching the discovery criteria specified by the object manager. The *ServiceConnectionFailedEvent* is generated by the middleware when it detects that the service to which the object is currently bound is unreachable. The *Access-RevocationEvent* is notified to the object manager by the currently bound service when the service revokes the object's binding to it. Reactions could be defined in the object to handle the later three types of events. The recovery actions may involve binding to alternate services, or generating application-defined events. The object manager generates the *ObjectUnboundEvent* when the object cannot be bound to any service, even after performing all the recovery actions. An object manager generates the *ObjectBound-Event* whenever the object successfully binds to a service. Below we present two examples of object level recovery using this model.

Example 1: Handling Discovery Failures: Consider a context-aware museum information application which allows a visitor to listen to the audio commentary about artifacts in his/her preferred language when the visitor is in the proximity of an artifact. Let us suppose that the visitor's preferred language is English. When a visitor comes in the proximity of an artifact, the application will try to discover and bind to the audio commentary service in English. However, it may happen that for certain artifacts audio commentary may be available in some different language, say Spanish. In this case, the application can be designed to discover and bind to the Spanish language commentary. Suppose that for certain artifacts even a Spanish language commentary is also not available. Instead, a *text commentary* service could be available for such artifacts. In such a case, the museum application can be designed to bind to the text commentary service when an audio commentary service cannot be discovered in the environment.

In Figure 6 we present the specification of *AudioCommentary* and *TextCommentary* objects in the museum application. In the *AudioCommentary* object we define the *BindToEnglishCommentary* reaction, which is triggered by the *RFIDEvent*. As part of the binding action, the object manager tries to discover English language audio commentary service. We attach a *DiscoveryFailureException* handler with this action. This exception handler is executed if no English language service is found in the environment. As part of the exception handling action, the object manager tries to discover an audio commentary service in Spanish. This may also fail. We attach an exception handler to this action also, which generates an *AudioCommentaryNot-BoundEvent*. This event is subscribed to by the *TextCommentary* object manager. In the *TextCommentary* object, we define a reaction which is triggered by this event. It binds the object to the text commentary service by discovering it in the environment.

Example 2: Handling Access Revocation Event: In the music player application, the audio player service in a room could revoke the application's binding to it. The *Audio-Player* object manager can handle it by binding to the audio player service running on the user's device, as shown in Figure 5.

4.2 Role-level Exception Handling Model

The role abstraction contains two *scopes* of execution. These correspond to the *role operation scope*, and the *role scope*. The role operation scope is nested within the role scope. Two kinds of exceptions may arise in the role operation scope. These correspond to the exceptions thrown

```
Object AudioCommentary RDD (//AudioCommentaryRDD.xml) {
    Reaction BindToEnglishCommentary {
        When Event RFIDEvent
        Action Bind Discover (ARTIFACT-ID=RFIDEvent.getID(), LANGUAGE=ENGLISH)
            OnException DiscoveryFailureException
                Action Bind Discover(ARTIFACT-ID=RFIDEvent.getID(), LANGUAGE=SPANISH)
                    OnException DiscoveryFailureException
                        Action NotifyEvent AudioCommentaryNotBoundEvent(ARTIFACT-ID=RFIDEvent.getID(), LANGUAGE=ENGLISH)
    } // end of reaction definition
} // end of AudioCommentary object definition
Object TextCommentary RDD (//TextCommentaryRDD.xml) {
    Reaction BindToTextCommentary {
        When Event AudioCommentary.AudioCommentaryNotBoundEvent
        Action Bind Discover (ARTIFACT-ID=AudioCommentaryNotBoundEvent.getAttr(ARTIFACT-ID),
                              LANGUAGE=AudioCommentaryNotBoundEvent.getAttr(LANGUAGE)) }
} // end of TextCommentary object definition
```

Figure 6. Object-level Event Handling Examples

```
Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
    Reaction AccessRevocationHandler {
        When Event AccessRevocationEvent
        Action Bind(//DeviceAudioServiceURL) }
} // end of AudioPlayer object definition
```

Figure 5. Object-level Access Revocation Handler

by the service being accessed as part of the operation execution, and exceptions that indicate occurrence of context invalidations. The exceptions within a role operation scope are handled as part of that operation's session. Any unhandled exception in the operation scope is propagated to the role scope. Within the role scope exceptions may be handled by any member of the role.

To monitor the context conditions associated with a role operation, we have defined the *ContextGuard* mechanism in our programming framework. During an operation's action execution, the associated context guard becomes active. Its function is to continually monitor the specified context condition and raise an exception if the condition doesn't hold. Two things associated with a context guard need to be specified. First, *when* to trigger its evaluation. Second, *what* context condition should be evaluated. Context guard evaluation is triggered by one or more context events. Every time a specified context event is notified to the role manager, it evaluates the context condition specified through the *GuardCondition* tag. If the context condition fails to hold, the role manager terminates the operation session and generates the *ContextInvalidationException*. In Figure 7 we present the specification of the *AccessDoctorReports* operation which includes a context guard. Here there is no handler specified for the *ContextInvalidationException*. Therefore, this exception is propagated to the role scope.

```
Role Nurse {
    Operation AccessDoctorReports {
        Action DoctorReports SessionMethod accessReports
            ContextGuard {
                When Event RoomStatusChangeEvent
                    GuardCondition WardRoomAgent.isPresent(thisUser)
                        ∧ WardRoomAgent.isPresent(members(Doctor))
            }
    } // end of operation
} // end of role
```

Figure 7. Context Invalidation Example

4.3 Exception Interface and Role-to-Role Exception Event Propagation

At the role scope, we define an abstraction called *exception interface*. It contains a set of operations that may be executed by a role member for executing recovery tasks. Such operations are only activated for exception handling purpose. Each exception interface operation is associated with an exception event, which we refer to as the *anchor event*. This is specified using the specification primitive `WHEN <Event-Name>`. The exception interface supports a queuing model of delivery and handling of exception events. An exception interface operation gets enabled when the anchor event is delivered to the exception interface queue. A role member can execute an exception interface operation only when an anchor event for that operation is present in the exception interface queue.

Exception interface operations could be executed by a specific role member, any role member, or all role members. Correspondingly, each exception operation is specified with one of the three special qualifiers: *Invoker*, *ANY*, and *ALL*, that are defined in the specification model. The *Invoker* qualifier allows only that role member who encountered an exception during a role operation execution to execute the exception interface operation. The qualifier *ANY* allows any member of the role to execute the exception operation. The anchor event is dequeued when the exception operation is invoked by a role member. We follow a non-deterministic evaluation model when two or more role members simultaneously try to execute an exception operation specified using the *ANY* qualifier, only one member will be successful in executing the exception operation. The qualifier *ALL* requires every member of the role to execute the exception operation before the anchor event is dequeued from the role's exception interface queue. For handling some exceptions, we may need to perform cooperative recovery actions involving members of different roles. Asynchronous event communication model is used for this purpose.

```
Role Nurse {
  ExceptionInterface {
    When ContextInvalidationException {
      EnableFor Invoker
        Operation RequestDoctorApproval
          Action NotifyEvent
            RequestPermissionEvent(requestor=thisUser)
    }
  } // end of exception interface for Nurse role.
  Operation ContinueAccess {
    Precondition count(GrantAccessEvent(grantee=thisUser)) > 0
      Action DoctorReports SessionMethod accessReports
  } // end of operation.
} // end of Nurse role.
Role Doctor {
  ExceptionInterface {
    When RequestPermissionEvent {
      EnableFor ANY
        Operation GrantPermission
          Action NotifyEvent
            GrantAccessEvent(grantee=
              RequestPermissionEvent.getAttr(requestor)) }
  } // end of exception interface.
} // end of Doctor role.
```

Figure 8. Exception Interface Example

In Figure 8 we present the specification of the *Nurse* and the *Doctor* role in the patient information system to illustrate use of a role's exception interface, and role-to-role event communication. We define the *RequestDoctorApproval* operation in the exception interface of the *Nurse* role. This operation is enabled when the *ContextInvalidationException* is propagated to the exception interface queue when the context guard for the *AccessDoctorReports* operation fails. We require that only that nurse who encountered the exception should be able to execute this operation. This is achieved by specifying *Invoker* as part of the *RequestDoctorApproval* operation's specification. When the nurse role member executes this operation, the *RequestPermissionEvent* is notified to the *Doctor* role, to request permission for continuing access to the doctor reports. The *requestor* attribute within this event is set to the role member who is invoking the role operation. We define the *GrantPermission* operation in the exception interface of the *Doctor* role. This operation is enabled when the *RequestPermissionEvent* is delivered to the exception interface queue of this role. We specify that *ANY* doctor can execute this operation. As part of the operation's action, the *GrantAccessEvent* is generated. The *grantee* attribute in this event is set to the identifier of the *Nurse* role member who encountered the context invalidation exception, and generated the *RequestPermissionEvent*. This enables *ContinueAccess* operation for that nurse.

5 Related Work

Several other groups have developed high-level programming models for context-aware applications [1, 13, 3, 19, 10]. Among these systems, failure conditions are considered by PCOM [1] and Gaia [13, 3]. PCOM considers failures that occur due to dynamic changes of an application's components. These failures are similar to session failures in our model. We consider such failures as well as other kinds of failures, for example discovery failures, binding failures, access revocations, context invalidations, and exceptions thrown by a service. In Gaia [3] failure handling mechanisms such as heart-beat based status monitoring, redundant provisioning of alternate services/applications, and restarting failed applications are proposed for handling failures related to application, services, and devices in pervasive computing environments. Other researchers have also looked at the failures arising in pervasive computing environments [7, 6]. In [7] the issue of disconnections affecting ambient service ensemble in pervasive computing environments is considered. They advocate an approach based on *state transfer* between identical services to tolerate such disconnections. In contrast, our focus is on application-level exception handling mechanisms, which can be used for discovery of and binding to different instances of a replicated service. A data-flow oriented architecture for building dependable pervasive computing systems is presented in [6]. The main objective of their design is to perform failure handling in pervasive systems *without* any user involvement. In contrast, our focus is on application-level exception handling model for designing recovery capabilities in a context-aware application's design.

Issues arising due to asynchronous handling of context events within pervasive computing applications have been identified by others [8]. They present an approach based on partitioning an application into a *logic* part – which never fails, and an *operation* part – which may encounter failures. In our model, the policies that are enforced by the various managers form the stable core that never fails, whereas role operations may encounter exceptions.

There has been considerable work done on exception handling in workflow systems [2, 9, 4]. There are many similarities between our exception handling model and these models. These similarities can be found in regard to reconfiguration of a task as part of exception handling, automatic exception handler execution in the context of a task, and user involvement for performing exception handling actions. The event-based exception handling model in our framework is conceptually similar to the model for workflow systems presented in [2]. The main distinguishing aspect of our model is the integration of synchronous exception handling with asynchronous event handling to enable coordinated recovery by members of different roles defined within an activity.

Exception handling is used as a forward error recovery mechanism in structuring fault tolerant concurrent object-oriented and distributed systems [18, 12]. Our future research plan is to investigate the suitability of such models for activity-level coordinated recovery involving multiple object managers. The need for cooperative exception handling in context-aware applications is also recognized by others [5]. In their model, context-sensitive exception handlers are used for handling exceptional context situations. In our model, the *reaction* mechanism is used for this purpose.

6 Conclusions

The primary contribution of this paper is the presentation of an application-level forward recovery model for role-based pervasive computing applications. The model combines event handling at the object-level with exception handling at the role-level to build robust context-aware applications. A novel mechanism in the form of *exception interface* is provided for roles which provides the ability for users to handle exceptions. The design of this model arose from our experiences in designing and implementing a variety of context-aware applications in our testbed environment. We have implemented this exception handling model in our programming framework and presented here several testbed examples to illustrate its capabilities.

References

[1] C. Becker, M. Handte, G. Schiele, and K. Rothermel. PCOM - A Component System for Pervasive Computing. In *PERCOM '04: Proc. of the Second IEEE Intl Conf. on Pervasive Computing and Communications (PerCom'04)*, pages 67–76, March 14-17 2004.

[2] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999.

[3] S. Chetan, A. Ranganathan, and R. Campbell. Towards Fault Tolerant Pervasive Computing. *IEEE Technology and Society*, 24(1):38 – 44, Spring 2005.

[4] D. K. W. Chiu, Q. Li, and K. Karlapalem. Web Interface-driven Cooperative Exception Handling in Adome Workflow Management System. *Information Systems*, 26(2):93–120, 2001.

[5] K. Damasceno, N. Cacho, A. Garcia, A. Romanovsky, and C. Lucena. Exception Handling in Context-Aware Agent Systems: A Case Study. *Springer LNCS*, 4408:57–76, 2007.

[6] C. Fetzer and K. Hogstedt. Self*: A Data-Flow Oriented Component Framework for Pervasive Dependability. In *Eigth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

[7] M. Fredj, N. Georgantas, and V. Issarny. Adaptation to Connectivity Loss in Pervasive Computing Environments. In *Proceedings of the 4th MiNEMA Workshop*, 2006.

[8] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System Support for Pervasive Applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[9] C. Hagen and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.

[10] J. Keeney and V. Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. In *Fourth IEEE Intl. Workshop on Policies for Distributed Systems and Networks (POLICY '03)*, pages 3–14, 2003.

[11] D. Kulkarni and A. Tripathi. Context-Aware Role-based Access Control in Pervasive Computing Systems. *SACMAT'08 Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, pages 113–122, 2008.

[12] R. Miller and A. Tripathi. The Guardian Model and Primitives for Exception Handling in Distributed Systems. *IEEE Transactions on Software Engineering*, 30(12):1008 – 1022, December 2004.

[13] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *PERCOM '05: Proc. of the Third IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom'05)*, pages 7–16, 2005.

[14] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-based Access Control: Towards a Unified Standard. In *RBAC '00: Proceedings of the Fifth ACM Workshop on Role-based Access Control*, pages 47–63. ACM Press, 2000.

[15] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, US, 1994.

[16] A. Tripathi, D. Kulkarni, and T. Ahmed. A Specification Model for Context-Based Collaborative Applications. *Elsevier Journal on Pervasive and Mobile Computing*, 1(1):21 – 42, May-June 2005.

[17] A. Tripathi, D. Kulkarni, and T. Ahmed. Exception Handling in CSCW Applications in Pervasive Computing Environments. *Springer LNCS Advanced Topics in Exception Handling Techniques*, 4119:161–180, 2006.

[18] J. Xu, A. Romanovsky, and B. Randell. Concurrent Exception Handling and Resolution in Distributed Object Systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1019–1032, October 2000.

[19] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.