Mechanisms for Access Control and Application-level Recovery in Context-Aware Applications

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Devdatta J. Kulkarni

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor Of Philosophy

December, 2009

Acknowledgements

I am deeply indebted to my advisor Prof. Anand Tripathi for providing the vision for this project. Without his continuous support, and patience, this task would have become impossible. Everything that I have learned over these years is because of my close interactions with him while working on the various projects in the group. It has been absolutely priceless.

My parents encouraged me to take up this challenge, and were by my side during the difficult periods. Aai and Baba, it is because of you that I have reached this stage.

Special thanks go to Tanvir Ahmed. He helped me get started on the middleware during the early stages of this thesis.

I want to thank the committee members, Prof. David Lilja, Prof. Eric Van Wyk, and Prof. Abhishek Chandra for serving on my committee.

It would have been impossible to work on this thesis without the financial support provided by the Computer Science department, and the National Science Foundation (NSF). Thanks to Prof. Phillip Barry, the TA coordinator in the Computer Science department, for giving me several opportunities to work as a Teaching Assistant in the Computer Science department. I would also like to thank the NSF for supporting this research through the following grants 0411961, 0087514, 0834357, and 0708604.

The office staff and the systems staff in the Computer Science department have provided significant assistance over the years. Specifically, I would like to thank Georganne Tolaas, Liz Freppert, Irene Jacobsen, Lori Vig, and Landon Thomas.

This long and periodically freezing journey would not have been so enjoyable without the warm company of a number of close, entertaining and colorful friends. Swedesh Srivastava,

Varun Chandola, Sudarshan Betigeri, Ajay Joshi, Manasi Joshi, Nitin Bhandari, Namrata Sopory, Amol Pangarkar, Sandeep Mane, Sachin Agrawal, Aravinda Reddy, Harsha Talkad, Swati Agiwal, Esha Nerurkar, Shruti Patil, Salil Bapat, Madhura Sane, Rahul Trivedi, Shantanu Jathar, Aniket Rane, Vinayak Kathare, Nikhil Kundargi, Shruti Koparkar, Gysler Castellino. You guys are simply awesome. It would have been really difficult to survive all these years without you.

Finally, thanks to Minneapolis for providing such a wonderful backdrop throughout. I hope every city that I may call home in the future has a Mississippi, a lake Harriet, a lake Calhoun, numerous coffee shops, and genuinely friendly people.

Dedication

To the loving memory of my grand parents, Aaji and Dada, who passed away before seeing the completion of this thesis.

ABSTRACT

Context-awareness is a central characteristic of several emerging application domains, characterizing the applications' ability to adapt and perform tasks based on ambient context conditions. Context refers to a situation in the physical or the virtual world that may be utilized by an application for the purpose of dynamic adaptation, for example, to acquire services needed in a given location. While the envisioned advantages of context-awareness are significant, providing access control and robustness guarantees for context-aware applications is a difficult task. This is because of the inherent dynamic nature of such applications and the environments in which they are deployed. In this thesis we develop models and mechanisms for addressing the access control and robustness problems in context-aware applications. We also develop a programming framework for building context-aware applications from their high-level design specifications.

An access control model for context-aware applications needs to support specification and enforcement of context-based access control policies. Such policies are related to assignment of context-based access privileges to users, access control for services that are dynamically integrated with an application, and context-based constraining of access to resources managed by a service. The first contribution of this thesis is the development of a context-aware role-based access control model (CA-RBAC) that addresses the above requirements of such applications. We identify the *context invalidation problem* associated with correct enforcement of context-based access control requirements, and develop a mechanism to address it.

Robustness of context-aware applications is affected due to failures in discovering the required resources and services during a context-driven reconfiguration, service crashes, and exceptions thrown by a service. Moreover, concurrent handling of context events can affect an application's correct behavior, if not properly coordinated. The second contribution of this thesis is the development of an application-level programmed error recovery model for such applications. This model combines asynchronous event handling with synchronous exception handling for building robust context-aware applications. A novel mechanism in the form of an *exception interface* is provided for roles through which users may participate in executing recovery tasks.

The third contribution of this thesis is the design and implementation of a generative programming framework for building context-aware applications from their high-level design specifications. The CA-RBAC model and the programmed error recovery mechanisms are integrated in this programming framework. This framework enables rapid construction of context-aware applications using a policy-driven middleware.

Contents

Acknowledgements i

Dedication iii

Abstract iv

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Problem Statement 4

1.1.1 Access Control for Context-Aware Applications 5

1.1.2 Robustness of Context-Aware Applications 6

1.1.3 Generative Approach for Designing Context-Aware Applications 8

1.2 Contributions 9

1.2.1 Design Model for Generative Programming of Context-Aware Applications 9

1.2.2 Context-Aware Role-based Access Control Model 10

1.2.3 Application-level Recovery Model 10

1.3 Infrastructure Facilities and Assumptions 11

1.4 Outline 11

2 High-Level Programming Model for Context-Aware Applications 13

2.1 Programming Model Requirements 14

2.1.1 Models for Context Data Integration 15

2.1.2 Context-based Dynamic Service Discovery and Binding 15

2.1.3 Representation of Users and their Tasks 16

2.1.4 Context-triggered automatic task executions 17

2.1.5 Context-based Access Control 17
2.1.6 Coordination Constraints 17
2.1.7 Programming Model Requirements Summary 18
2.2 Programming Model 18
2.2.1 Activity 19
2.2.2 Event Model 21
2.2.3 Object 22
2.2.4 Role 25
2.2.5 Reaction 27
2.2.6 Middleware 28

3 Access Control Model for Context-Aware Applications 30
3.1 Context-Aware Role-based Access Control Model 31
3.1.1 Role Admission and Validation 32
3.1.2 Context-based Permission Activation and Context Guard 34
3.1.3 Personalized Role Permissions 37
3.1.4 Context-based Role Permissions 38
3.1.5 Context-based Resource Access 39
3.2 Related Work 41

4 Primitives for Programmed Error Recovery 43
4.1 Recovery Model and Robustness Primitives 44
4.1.1 Object-level Event Handling and Recovery Model 44
4.1.2 Role-level Exception Handling Model 51
4.2 Related Work 61

5 Design of the Generative Middleware 64
5.1 Generic Managers 65
5.2 Runtime Representation of Activities 66
5.2.1 Policies 68
5.2.2 Policy Derivation for Runtime Management 68
5.3 Activity Manager Architecture 69
5.3.1 Context Event Handling 69
5.3.2 Summary of Context Event Dispatch Rules 70
5.3.3 Application Defined Notification Event Handling 71
5.4 Role Manager Architecture 71
5.4.1 Role Manager Interfaces 71

5.4.2 Events and Event Handler 71
5.4.3 User Interactions 73
5.4.4 Operation Execution 73
5.4.5 Context Guard Management 73
5.4.6 Exception Interface Manager 74
5.5 Object Manager Architecture 74
5.5.1 Object Manager Interfaces 75
5.5.2 Object Binding Policies 75
5.5.3 Binding Protocol 75
5.5.4 Binding Failure Detection and Handling 76
5.5.5 User-Service Interactions 76
5.5.6 Object-level Access Control Policies 77
5.5.7 Resource Access Constraints 78
5.6 Related Work 78

6 Design of a Context Detection Infrastructure 81
6.1 Context Detection 81
6.1.1 Context Acquisition 82
6.2 Agent-based Context Detection Framework 83
6.3 Examples of Context Agents 84
6.3.1 Context-Aware Music Player 84
6.3.2 Patient Information System 86
6.3.3 Museum Application 86
6.4 Related Work 87

7 Experiments and Evaluations 89
7.1 Support for Design Evolution 90
7.2 Code Reusability 95
7.3 Application Complexity 96
7.4 Performance of Activity Instantiation Procedures 98
7.5 Limitation 99
7.6 Extensibility 99

8 Conclusions 101
Bibliography 103

Appendix A. Activity Grammar 110
A.1 Operators 112

Appendix B. Case Study Applications **113**

B.1 Notation 113

B.2 Context-Aware Music Player 113

B.2.1 Application Components and Services 114

B.2.2 Specification 115

B.3 Context-Aware Patient Information System 118

B.3.1 Application Component and Services 118

B.3.2 Specification 119

B.4 Context-Aware Distributed Meeting 120

B.4.1 Application Components and Services 121

B.4.2 Specification 122

List of Tables

7.1 Code sizes of the middleware and various infrastructure services 95
7.2 Middleware modules 96
7.3 Application code sizes 96
7.4 Activity characteristics: various applications 97
7.5 Coupling measurements 98
7.6 Activity instantiation times: various applications 98

List of Figures

1.1 Context-aware application spanning three active spaces . . . 3
1.2 Regeneration of context-aware applications based on different designs . . . 8
2.1 Generative programming approach: overview . . . 14
2.2 Activity class diagram in UML . . . 20
2.3 Activity grammar . . . 20
2.4 Patient Information Activity specification . . . 21
2.5 Object grammar . . . 22
2.6 Context-based discovery binding example . . . 24
2.7 Parameterized CurrentRoom RDD . . . 24
2.8 Role grammar . . . 25
2.9 Model of user–service interactions . . . 25
2.10 Interaction models . . . 27
2.11 Reaction syntax . . . 27
2.12 Reaction example . . . 28
3.1 Context-Aware RBAC Model (CA-RBAC) . . . 31
3.2 Role admission and validation constraint example . . . 34
3.3 Operation precondition and context guard example . . . 35
3.4 Context Guard in exam session . . . 37
3.5 Personalized role permissions . . . 38
3.6 Context-based object binding example . . . 39
3.7 Resource access constraint example . . . 40
4.1 Object-level event model . . . 44
4.2 Object state diagram . . . 45
4.3 Object syntax with exception handlers . . . 46
4.4 Example of object-level recovery . . . 46
4.5 Object-level event handling example . . . 47
4.6 Access revocation handler . . . 49

4.7 Issue leading to the binding order requirement 49
4.8 Binding order 50
4.9 Issue leading to the requirement of atomic processing of context events 51
4.10 Relationship between role scope, operation scope, and action scope 51
4.11 Syntax for role definition with exception handlers 52
4.12 Example of action-level exception handling 53
4.13 Session-level exception handling 54
4.14 Handling disrupted transactions: Making disrupted transaction resumable . . . 55
4.15 Context guard example 57
4.16 Syntax for role exception interface 57
4.17 Handling transaction failure 59
4.18 Handling disrupted transactions using role's Exception Interface 59
4.19 Collaborative recovery actions 60
4.20 Exception interface example 61
5.1 Three phases in context-aware application generation 65
5.2 Architectural overview of context-aware application generation 66
5.3 Activity DOM tree: Patient Information System activity 67
5.4 Architecture of a Role Manager 72
5.5 Architecture of an Object Manager 74
5.6 Access constraint example: PatientDB object manager 78
6.1 Agent's event processing model 83
6.2 Event trigger hierarchy: User location detection 84
6.3 Agent configurations: Context-Aware Music Player 85
6.4 Event trigger hierarchy: RFID-based proximity detection 87
7.1 Patient information system: Satisfying modified requirements 91
7.2 AccessPatientInformation operation 93
7.3 Avoiding inter-application interference 94
B.1 Context-Aware music player architecture 114
B.2 Context-Aware patient information system architecture 118
B.3 Context-Aware distributed meeting architecture 121

Chapter 1

Introduction

Mark Weiser's seminal paper [74] outlined a vision in which computing was no longer confined to the desktops but was *ubiquitous*, everywhere and anytime. Context-awareness is a central aspect of the ubiquitous/pervasive computing applications [63], characterizing their ability to adapt and perform tasks based on ambient context conditions. The notion of *context* in pervasive computing applications relates to the characterization of ambient conditions and physical world situations that are relevant for performing appropriate actions in the computing domain for their correct or desired behavior. Such applications may utilize context information for the purpose of dynamic adaptation, for example, to acquire ambient resources and services needed in a given location.

One type of context-aware applications that have been considerably successful in recent years are those which utilize user location and time as the two most predominant types of contexts ("location-aware computing"). Other kinds of contexts that have been typically considered in context-aware applications include the devices being used, the network on which the devices are connected, and the activities in which the user is currently engaged. Additionally, there can be other conditions and characteristics that may be relevant in defining a context. For example, in some situations the temporal attributes associated with an activity, such as its duration and time of occurrence, may be important. Other factors such as device capabilities, physical proximity of devices, and available bandwidth can also be important in some situations.

In recent years there has been steady adoption of context-awareness in number of application domains such as location-based services, tour guides, smart rooms, assisted living applications, and hospital information systems. A location-based store finder application returns the list of stores based on the user's preference and which are close to the user's current geographical location. The application may dynamically change the list of returned stores if it finds that the nearest store is closed, or there is heavy traffic on the road that the user may need to take

to reach the nearest store. A context-aware tour guide application provides the visitors with the information about the surrounding areas and the activities that may be happening nearby. The application can also assist a visitor in various ways such as, preparing the tour itinerary, dynamically modifying the itinerary if some tourist destinations on the original itinerary are closed, recommending the visitors to visit certain tourist spots based on the visitor's interest, and displaying the current locations of visitor's group members on visitor's device. A context-aware assisted living application that monitors the health of an elderly person determines the current state of the person and accordingly notifies the health-care provider. It can also perform other functions such as, allowing the person to access the medical cabinet during certain times of the day, reminding a person about his/her daily medication, allowing the person to request a depleted medication, and sending notifications to the person's in-house nurse when the requested medication is ready for pickup.

During the first generation of context-aware application development researchers had identified some of the typical context-based adaptive characteristics of such applications [64]. These characteristics included, dynamic discovery and binding of resources/services based on context information, displaying information about a user's surroundings based on the context information, and automatically executing certain tasks when a specific context condition becomes true. In recent years, as the realm of context-aware computing has extended to include domains such as health-care systems, context-based characteristics such as context-based access control and context-based multi-user coordination have begun to emerge as key requirements of context-aware applications [10, 68].

Context-based access control and coordination requirements are related to constraining users' access to resources and services based on different kinds of context conditions. Such requirements are important in pervasive computing environments where it is possible for the users to seamlessly access resources and services from anywhere and anytime. Availability of appropriate access control and security models can provide the necessary barrier for preventing such environments from degenerating into a security nightmare. Here are some examples of context-based access control and coordination requirements in pervasive computing applications. In a hospital information system, a nurse may be allowed to access a doctor's confidential reports only in the presence of the doctor and if the doctor has given an approval to the nurse for such an access. Another requirement could be that a nurse can access records of only those patients on whose patient-care team the nurse currently belongs. In a context-aware meeting application, we may have a requirement that classified slides may be presented only when there is only office staff and no guests in the meeting room. As yet another example consider a city tour guide application that is used by a group of visitors during their visit to a city. This application may allow the visitors to chat with one another while they are visiting different places in the city.

In this application the chat session can be enabled by the visitors only after the visitors have paid for the chat service and after the tourist information center has given them the approval to initiate the chat client.

Certain context-aware applications, such as a museum tour guide, may run on a user's device. On the other hand, an application such as a smart meeting room may run on some infrastructure server and may involve different kinds of devices, such as printers, projectors, and machines, available in the rooms where the meeting is being held. In the literature, the computing environment in which context-aware applications are deployed is called an *active space* [60]. Such a space may span multiple physical spaces. An active space provides the infrastructure services that are required for deploying and running context-aware applications. Such services include sensor data aggregation service, context modeling and inference service, discovery services, location-independent naming service, and authorization and access control service [60, 32].

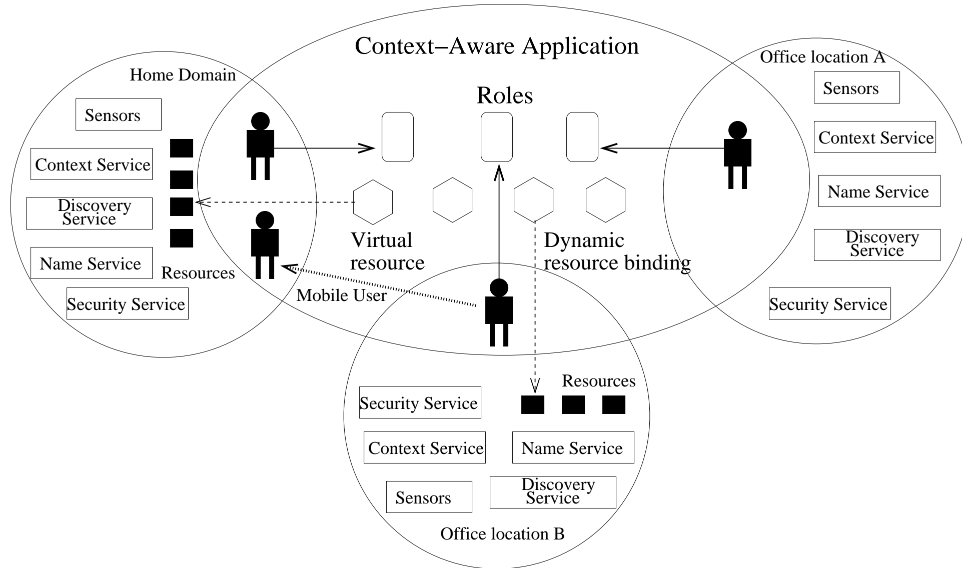


Figure 1.1: Context-aware application spanning three active spaces

In Figure 1.1 we show three active spaces consisting of the computing environments available at three different locations. We show a context-aware application that is deployed in these active spaces. The resources and services available in the active space can be accessed within the application through the *virtual resources*. The binding of a virtual resource with an active space resource may change based on the context conditions. Users of the context-aware application can perform various tasks through one or more *roles* defined in the application.

From the perspective of designing a context-aware application, the issues are related to

the representation, creation, and management of an application's execution environment. Specifically, the following issues need to be considered while designing a context-aware application. An application may be single user or multi-user. Users may need to be provided access to various resources and services based on the current and historical context information. In a multi-user application certain resources may need to be private for each user, whereas certain resources may need to be shared among all the users. An application may define one or more tasks, which may need to be performed by users or may need to be executed automatically when certain context conditions become true. To address some of the above issues researchers have developed high-level programming models for context-aware applications [60, 32, 31].

The other aspect of context-aware application development is designing the context modeling and detection infrastructure. The design of this infrastructure is driven by the context-based adaptation requirements of the various context-aware applications. The objective of defining context models is to support representation of context information to enable its programmatic use by an application. A context model needs to represent the dynamic context situations and conditions that are required within various context-aware applications. Specifically, a context model needs to represent the various *entities* and *relationships* among them. Examples of entities include users' physical and network locations, users' devices, and the characteristics of the devices such as network bandwidth and available resources, temporal conditions and ambient variables such as room-temperature. Examples of the relationships between these entities include the following: number and identity of people present in a specific location, a person's proximity to other people and physical objects, history of tasks performed by a person at a particular location, and history of locations visited by a user. To support context-based adaptations, context information is required to be integrated as part of an application's design. In certain situations the application may need to query the information, whereas in certain other situations it may need to be notified to the application. Researchers have developed toolkits that assist in context modeling, sensing, and context acquisition [24, 39].

1.1 Problem Statement

In this thesis we consider the problems of access control and robustness of context-aware applications.

- *Access control models and mechanisms for context-aware applications:* There are two main issues related to the development of access control models and mechanisms for context-aware applications. First is the specification of different kinds of context-based constraints within an access control model for such applications. A model needs to be

expressive enough to capture different kinds of access control requirements for context-aware applications. The constraints may be based on low-level information such as user location, time of the day, the devices through which the access is requested, or on high-level information such as user's current and past activities, presence of a user with some other user, or presence of a group of users in a specific location. The second issue is related to the enforcement of context-based access control requirements. Because certain aspects of the context information may be inherently dynamic in nature, it is possible for the related context condition to become false during the course of execution of certain context-dependent task. For certain applications, it may be important that such context-dependent tasks be terminated when the associated context conditions change.

- *Robustness of context-aware applications:* Robustness of context-aware applications is affected due to their inherent dynamic nature, exemplified by their context-based adaptation requirements, and also due to the dynamic nature of the environments in which such applications are deployed. An application may fail to discover the required kinds of resources and services in the environment. An application's binding with an active space service may break for various reasons. Concurrent adaptive actions, which are triggered by changes in context conditions, may affect the application's correct behavior.

1.1.1 Access Control for Context-Aware Applications

The general access control problem in system designs considers whether to grant the requested access on the specified item to the subject who is requesting the access. In context-aware applications the traditional notion of access control needs to be extended to take into consideration the context information. Role-based access control (RBAC) [62] is a widely used paradigm for designing modern access control systems. The advantages provided by RBAC over traditional access-matrix based models include ease of management, and policy neutrality [55]. The NIST RBAC model [27] is the standard reference used for developing enterprise-level access control systems. In a typical RBAC model, the abstraction of *role* is used as the basic unit of access control. *Permissions* are associated with roles, which effectively grant privileges for accessing resources and services in the system. The entities on which access control needs to be exercised are defined as the system's objects. Users access the objects by exercising permissions of the role to which they have joined.

Our focus in this thesis is on the issues related to building RBAC models and systems for context-aware applications, which utilize context information in making access control decisions. Other researchers have also developed RBAC models specifically for context-aware pervasive computing applications [61, 19, 53]. However, these models do not adequately support the

following requirements, which are crucial for designing access control systems for context-aware applications.

- *Issues related to context-based constraints specification:* Different kinds of constraints are required to be supported within an access control model for context-aware applications. Examples of such constraints include context conditions under which a user may be allowed to be a member of a role, context conditions when a user is allowed to exercise a specific role permission, under what conditions a user is allowed to access a specific service or a subset of resources being managed by a service, and conditions under which the set of resources accessible to two different users belonging to a role can be same or different.

- *Issues related to enforcement of context-based constraints:* The enforcement of context-based access control requirements is challenging because of the dynamic nature of the context information. Specifically, access to certain active space resources and continuation of a user's ongoing task may need to be permitted only when specific context conditions hold. A role member's access to such resources and services and the ongoing task executions need to be revoked when the required context conditions fail to hold.

1.1.2 Robustness of Context-Aware Applications

Through our experiences in designing several context-aware applications we realized that the robustness of context-aware applications is affected due to the following factors.

- *Issues due to dynamic application reconfiguration:* The dynamic reconfiguration mechanisms integrated in an application for context-based adaptations can themselves become a cause of robustness problems, if not properly designed. For example, the order in which an application handles concurrent context events can be important for correct application behavior. In some situations an application may fail to function correctly due to failures in finding the required resources and services during a reconfiguration. This may happen if the discovery service has failed, or if there are no matching services registered with the discovery service. A reconfiguration action could also disrupt an ongoing interactive session with the service to which the application is currently bound, causing such a session to terminate prematurely.

- *Service-level failures:* Various kinds of failures can arise during users' interactions with the services bound to the application. These include failures due to network disruptions, service crashes, and access revocations by services. A service may throw exceptions because of incompatible resource access protocols, insufficient security privileges of the application, or due to the failures in executing service functions.

- *Context Invalidations:* 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* [46]. For correct enforcement of such tasks mechanisms are needed 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.

- *Inter-Service Interference:* An interference situation may arise when actions of one context-aware application may affect some other context-aware application deployed in the same active space. For example, an application that controls window blinds in a room may affect an application that raises an alarm based on any motion of the doors or windows in the room. This kind of interference can sometimes lead to unexpected or undesirable behavior if suitable coordination mechanisms are not integrated in the applications.

Traditionally, paradigms based on backward and forward error recovery have been used for building robust systems. Backward error recovery such as roll-back recovery may not be feasible in context-aware applications in general because it may not be possible to roll-back the actions that the application has performed possibly affecting the external world. Approaches based on forward error recovery using exception handling techniques are particularly promising for context-aware applications. Such approaches try to bring an application to a consistent state after a failure by performing alternative actions which are programmed as exception handlers within an application's design. In this thesis we develop an application-level forward error recovery model for designing robust context-aware applications. The salient feature of this model is the juxtaposition of system-level recovery with human-centric recovery for designing robust context-aware applications.

Failures in context-aware applications have been addressed by other researchers [22, 30, 16, 28]. Apart from [22], none of the other models have considered exception based mechanisms for designing robust context-aware applications. The main distinguishing aspect of our model as compared to the model presented in [22] is the integration of synchronous exception handling with asynchronous event handling to design application-level programmed recovery actions in context-aware applications. Our exception handling based approach for designing robust context-aware applications has some similarities with the exception handling models designed for workflow systems [12, 37].

1.1.3 Generative Approach for Designing Context-Aware Applications

In this thesis we investigate the solutions to the above problems as part of a high-level programming model that we have developed for context-aware applications. This programming model is based on the principles of the *generative programming* paradigm [21]. In this approach, context-aware applications are designed using a high-level domain-specific *design model* and are realized through a distributed policy-driven middleware. This design model and the middleware provide a composition framework for integrating users, application-defined components, and infrastructure services to build the execution environment of a context-aware application. The process of generating an application's execution environment using this approach is shown in Figure 1.2 [1] . The central point of this figure is that using the same set of application components, resources, and services, one can generate different execution environments for a particular kind of context-aware application. The application generation framework consists of *generic components* which are specialized based on the high-level design of an application to generate a particular execution environment. Variations in an application may be required due to reasons such as changes in user requirements or changes in the domain's administrative policies.

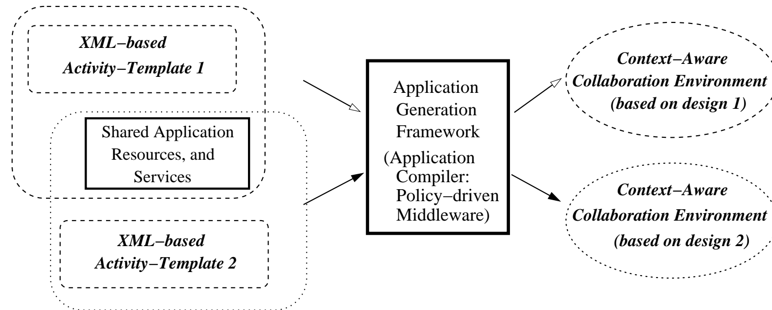


Figure 1.2: Regeneration of context-aware applications based on different designs

In contrast to the generative approach to context-aware application development taken here, the traditional approach for building such applications has been to custom build an application for some specific targeted domain, such as city tour guides [23], context-aware communication facilities [39], and interactive museums [29]. Such approaches require custom design and implementation of complex protocols for dynamic resource discovery, binding, and access control. This results in high cost of programming and development efforts. Moreover, such approaches do not easily support design modifications needed due to technology evolution

[1] Figure is modified from a similar figure in [2]

and changes in design requirements.

There are several advantages of using the generative approach for context-aware application development. The domain specific design model provides mechanisms for programming a context-aware application using high-level abstractions, thus simplifying their designs. The approach supports rapid realization of the execution environment of a context-aware application from its high-level specification. This is crucial, since an application may need to be modified based on evolving requirements or changes in an active space's administrative policies. The middleware provides generic components that encapsulate protocols for context-based service discovery and binding, user authentication, secure interactions among distributed components, and context-based access control and coordination. This relieves the programmer from the task of custom development of such protocols separately for each application. It is possible to provide tool support to verify domain specific security and coordination properties of the synthesized applications [3].

1.2 Contributions

This thesis makes the following contributions in the state of the art in context-aware application development.

1.2.1 Design Model for Generative Programming of Context-Aware Applications

We develop a high-level programming model to support the generative programming paradigm for building context-aware applications. This model provides a high-level abstraction for representing context-aware multi-user collaborative application. The model also provides abstractions for representing virtual resources, users and their tasks, and context services. Mechanisms are provided for binding a virtual resource with an active space service based on the context conditions, enforcing context-based access control for user tasks, supporting context-based coordinations policies, and automatically executing certain tasks when the specified context conditions become true. The model supports integration of different context services within the design of a context-aware application.

In this thesis we present the designs of several context-aware applications that we developed using this high-level programming model. These include a context-aware music player, a context-aware patient information system, a context-aware distributed meeting, and an emulation of a museum environment. To support the context-aware applications developed using our framework, we built an agent-based context detection and aggregation system.

1.2.2 Context-Aware Role-based Access Control Model

We develop a Context-Aware Role-based Access Control model (CA-RBAC) [46] for specifying and enforcing access control requirements of context-aware applications. The design of this model arose from the unique access control requirements in context-aware applications. These requirements are related to users' memberships in roles, constraining users' access to active space services to occur under specified context conditions, constraining permission activation by role members under specified context conditions, providing access to different services to different role members through a specific role permission, and controlling access to resources based on the relationship between the user's context-based attributes and those of the resource being accessed. It was observed that these requirements are not adequately satisfied either by the NIST RBAC model or other access control models designed for context-aware applications in the literature.

The salient features of the CA-RBAC model are the following. Mechanisms are provided to control a user's admission and continued membership in a role based on the context conditions. The *dynamic discovery and binding* mechanism in the model makes it possible to control a user's access to different active space services under different context conditions. In the model we provide the *precondition* mechanism for constraining a user's permission activation under specified context conditions. We provide the *context guard* mechanism to revoke a user's permission if the context conditions which are required to hold for that permission to be active no longer hold. We introduce the notion of *personalized permissions* for a role. Through a personalized permission different users in the role are able to access different active space services. The mechanism of *resource access constraint* is provided in the model to constrain a user's access to a subset of resources that are managed by an active space service based on the context-based relationship of user and the resources being accessed.

1.2.3 Application-level Recovery Model

Towards the goal of implementing robust context-aware applications we design and implement a *forward recovery model* for such applications [45]. This model enables programming of application-level recovery actions for handling various kinds of failure conditions and erroneous situations arising in such applications. These include, service discovery failures, service binding failures, exceptions raised by a service, and context invalidations. The recovery mechanisms in this model fall into two categories: *synchronous exception handling* and *asynchronous event handling*. Exception handlers and event handlers are built into an application's design to execute programmed recovery actions.

In our experience two kinds of recovery patterns often tend to arise in context-aware

multi-user collaborative applications. In one pattern, application recovery *does not involve* participation of any user. The recovery tasks are autonomously executed by an application. In the second pattern, *human participation is required* in performing application recovery tasks. The human-centric nature of context-aware multi-user collaborative applications makes this form of recovery possible, and in certain cases, indispensable. The mechanisms of exception handling and event handling together enable programming of these recovery patterns in the designs of context-aware applications.

1.3 Infrastructure Facilities and Assumptions

We make the following assumptions regarding the active spaces in which the context-aware applications are deployed.

First, we assume that appropriate context services are available in an active space. Development of context models and context services is an important aspect of designing context-aware applications. Ambient context detection requires continuous sensing of various different kinds of conditions in the environment, possibly at different locations, and real-time aggregation of the continuous streams of sensor data to infer the context conditions of interest. We have built an agent-based distributed context detection infrastructure for supporting the context-aware applications designed by us. However, the issues arising in context modeling and designing context services fall outside the scope of this thesis.

Second, we assume that the application components that perform various management functions such as role-based coordination and access control, context-based dynamic service discovery and binding, and automatic execution of context-dependent tasks, are run on a set of trusted and robust servers. These servers are assumed to be resistant to crash failures. For building context-aware applications that are immune to crash failures one can use techniques such as replication of application execution environment and running application instances in a primary-backup mode. Such approaches will complement the application-level recovery mechanisms that we have developed here. We also do not consider the failure of infrastructure services such as discovery service failures and context service failures.

1.4 Outline

In Chapter 2 we present the high-level programming model that we have developed for context-aware applications. We use a context-aware patient information system as a running example to illustrate the various aspects of this model. In Chapter 3 we present the context-aware role-based access control (CA-RBAC) model for context-aware applications. Through several

examples we illustrate the various constructs in the model. In Chapter 4 we discuss the robustness issues arising in context-aware applications and present the recovery model that we have developed to handle various kinds of failure conditions arising in these applications. In Chapter 5 we present the design of the generative middleware for context-aware applications. We discuss the architecture of various generic components, various kinds of policies used by these components, and the procedure of specializing the generic components for generating application specific components. In Chapter 6 we present the design of an agent based context detection infrastructure. We also present the architectures of various context agents that we designed in our framework for supporting various context-aware applications. In Chapter 7 we present the evaluations of our programming model. The primary goal is to evaluate the benefits provided by this model over stand-alone development of context-aware applications. We conclude, with outline of the future work, in Chapter 8.

Chapter 2

High-Level Programming Model for Context-Aware Applications

We saw in Figure 1.1 that a context-aware application may involve multiple users who may be mobile across different physical spaces and administrative boundaries. The application may need to discover and bind to different active space services under different context conditions. Such discovery and binding actions may possibly be based on context conditions. For certain applications a user's task execution may need to be constrained based on context conditions. In this Chapter we present a high-level programming model that provides abstractions and mechanisms for representing these different aspects of a context-aware application.

The design model has been developed based on the notions of the generative programming paradigm [21]. The goal of generative programming paradigm is to ease the task of developing applications belonging to a particular application family. Figure 2.1 gives an overview of this approach. The process starts with domain experts performing domain analysis to identify the core set of requirements to be supported in any application belonging to the application family. Such an analysis is followed by domain engineering that includes development of three types of entities. First is a design and modeling notation, such as a domain specific language (DSL), for designing and modeling an application at a high-level of abstraction. Second is a generator middleware containing generic components that would be used in application synthesis. Third is a repository of configuration knowledge that specifies constraints and dependencies on the use of generic components during the application generation process.

Application generation is performed by the generator middleware by *transforming* the generic components based on the high-level design specification of the application. The transformation usually corresponds to performing *refinements* of the generic components. Different kinds of

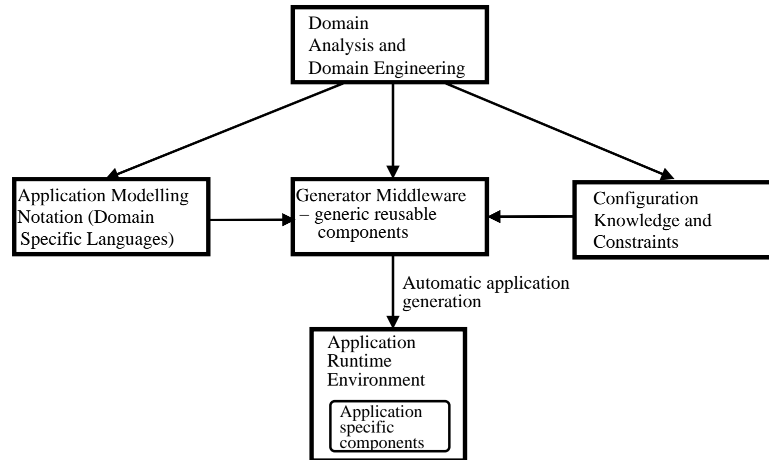


Figure 2.1: Generative programming approach: overview

refinements are possible such as [21], concretizing the generic components by adding application specific details, decomposing high-level components into specific low-level components, choosing an appropriate internal representation for the application data, and choosing the suitable algorithms.

The generative programming methodology has the following advantages over traditional approach of stand-alone application development. First, DSLs simplify the task of designing an application by providing high-level domain abstractions for application modeling. This aids in performing domain specific error checking and optimizations for the application. Second, the approach supports reuse of the generic components and the configuration knowledge as part of designing different applications of the family. Third, application development time is significantly reduced as complex programming tasks for runtime integration of components are not required.

2.1 Programming Model Requirements

Here we present the specific requirements that we considered in designing the high-level programming model for context-aware applications. These requirements are related to dynamic discovery and binding of an application to active space services, specification of tasks that may be performed by a user or which may be executed automatically by an application, context-based access control and coordination, and integration of context information within context-aware applications.

2.1.1 Models for Context Data Integration

A context-aware application needs to interface with context services for obtaining the required context information. Certain information such as the number of users present in a particular room may be provided by the context services directly. However, certain other information, such as whether a user belonging to a particular role arrived in a room, cannot be directly obtained from such services. This is because, in general, the context services would be designed to address the needs of wide variety of context-aware applications and may not understand application-specific role definitions.

Context services may be either deployed on a user's device or they may be available as infrastructure services in the environment. For example, GPS locations can be obtained from sensors that are deployed on a user's device. On the other hand, information such as identities and number of people present in a room are obtained by interfacing with a context service deployed in that room. Such services would typically support query or notification interfaces through which an application may obtain the required context information. Because context information is used by an application in enforcing access control, it is important to ensure the integrity and authenticity of the sensor data gathered by various context services.

In the programming model we need abstractions to represent the context services that may be used by a context-aware application to obtain the required context information. From an application's perspective we consider the context information as *internal* or *external*. Internal context conditions are related to an application's execution state. An example of this is the history of various tasks executed by the application users. External context conditions pertain to ambient conditions, which may change *spontaneously*. Examples of external context are user location information and devices available in an environment. The programming model also needs to support mechanisms that enable derivation of application-specific high-level context information from the low-level context information available through various context services.

2.1.2 Context-based Dynamic Service Discovery and Binding

Context-based service discovery and binding refers to the ability of a context-aware application to dynamically discover and integrate resources and services that are available in an active space. Context information may need to be used as part of this process. For example, consider a context-aware museum information application that is deployed on the mobile devices given to museum visitors. This application may be designed to detect a visitor's presence next to an museum artifact, and dynamically bind to the audio commentary service for that artifact. The specific audio commentary service that is accessible to a visitor depends on his proximity to a specific artifact. Moreover, as a visitor moves from one artifact to another, the application

needs to dynamically discover and bind to the appropriate artifact's commentary service.

In a multi-user application, we may require that certain resources and services be common to all the users. As an example consider a group-study application deployed within a museum. This application may provide a virtual whiteboard for the group members as a shared discussion facility. At the same time we may also require that each user has his/her private view of the resources and services. For example the group-study application may provide every user a private virtual diary to take notes during the museum visit.

In the programming model we need a mechanism for logical representation of the functional view of a service required by a context-aware application. We use the term *object* for such a logical representation. An object is used to represent a virtual resource defined in Figure 1.1.

Certain objects may be common to all the application users whereas certain other objects may need to be private to each individual user. An object may be dynamically bound to different services under different context conditions. Dynamic binding of an object with a service may involve discovering the appropriate service based on the current context conditions. For this purpose, an object may need to be associated with the description of the service's functional interfaces, and its attributes. Some of the attributes such as the *location* may be be specified at runtime based on the current location of the user.

2.1.3 Representation of Users and their Tasks

A context-aware application supports execution of various tasks by its users. Ambient resources and services may be accessed within such tasks. An example is a patient information system involving several kinds of users, such as doctors, nurses, and specialists. The tasks that may be performed by them include creating, updating, and accessing patient records, requesting laboratory tests, etc.

We use the concept of a *role* for representing the tasks that require user participation. In the domain of collaborative applications, the notion of *role* has been traditionally used to represent a set of responsibilities and privileges associated with a particular position within an organization [47, 18]. *Roles* provide a natural abstraction for representing users in collaborative applications, without requiring prior knowledge of their individual identities. Within a role, one can define various *operations* through which the users admitted to that role can perform various tasks [3]. A task may involve accessing a service in the active space. As an example, consider a context-aware hospital information system. Several roles may be defined in this application such as, doctor, nurse, and specialist. Some of the tasks associated with these roles include creating and updating patient reports, requesting laboratory tests, accessing patient records, and accessing the closest printer.

2.1.4 Context-triggered automatic task executions

One of the primary motivations for context-aware computing is to enhance user experiences by relieving them from performing routine tasks which can be automatically executed based on the occurrence of certain context conditions. Examples of such tasks include, dynamic discovery and binding of the application to appropriate active-space services, dispatching alerts when some predefined context conditions are true, or notification of a message to person based on the person's current location. In the programming model we define the concept of a *reaction* for representing such tasks that need to be automatically executed when certain context conditions become true. A reaction is similar to a role operation, the only difference being it is automatically executed by the application.

2.1.5 Context-based Access Control

In context-aware applications we may need to restrict execution of certain tasks by a user based on context conditions. For example, in a context-aware patient information system a nurse may be allowed to initiate workflow for accessing a doctor's reports about a patient only when in the presence of a doctor. Another example is the context-aware music player application. We may require that the user is allowed to access a room's speakers only if there is no other person present in the room. In the programming model we need a mechanism to restrict execution of a task by an user based on context conditions. We also need a mechanism to revoke a user's access to a service when the required context conditions change while the user is accessing that service.

In certain cases an active space service may be managing a set of resources and we may need to restrict a user's access to a subset of these resources based on the context conditions. For example, in the patient information system the database service may be managing and controlling access to information about all patients in different hospital wards. There might be a requirement that a nurse is allowed to access records of only those patients who are admitted to the ward where the nurse is currently located. In the programming model we need a mechanism through which the application can restrict, based on context conditions, access to a subset of resources that are being managed by a service. In Chapter 3 we present various access control mechanisms that we have designed in our programming model for context-aware applications.

2.1.6 Coordination Constraints

Coordination is concerned with the order in which various tasks are performed within a context-aware application [50]. For example, in the patient information system only specialist can modify the patient's test reports whereas the patient's doctor can access and modify any part of

a patient's report. Moreover, the test reports can be created only after the appropriate tests have been performed. Such coordination policies may widely vary from one organization to another. The nature of coordination in the above example resembles loosely coupled asynchronous interactions, which are largely characterized by workflow environments. On the other hand, in real-time synchronous collaborations, users are connected to the system simultaneously and their interactions occur through interactive sharing of multimedia objects. An example of this kind of interaction is the use of the shared whiteboard by members of a study-group in the museum application. The coordination policies may depend on both *internal* and *external* context conditions.

The programming model needs to provide mechanisms for expressing and enforcing various coordination constraints that may depend on internal or external context conditions.

2.1.7 Programming Model Requirements Summary

A high-level programming model needs to support the following abstractions and mechanisms to enable designing of context-aware applications that may involve multiple coordinating users, who may be mobile and may need to access different resources in different physical locations.

- *Object-level requirements:* A mechanism is required for logical representation of the functional view of a service required by a context-aware application. Mechanisms are required to discover and bind such a representation to the appropriate services available in an active space. Such a binding may be based on context information.

- *Task-level requirements:* Mechanisms are required to represent users and their tasks. Access control and coordination policies may be needed to constrain the execution of certain tasks. A mechanism is required to represent the tasks that may be automatically executed when certain context conditions become true.

- *Context integration requirement:* A mechanism is required to represent the context services required by a context-aware application. Model needs to support mechanisms that enable derivation of application-specific context information from low-level generic context data.

2.2 Programming Model

Here we present the high-level programming model that we have developed for designing context-aware applications. We use a context-aware patient information system as a running example to illustrate the various aspects of this model. Such a system may be deployed in a hospital and accessed by the hospital nurses and doctors. It supports a number of requirements as follows [26].

- The patient information system maintains various kinds of records and reports for the patients admitted to different wards in the hospital.
- The system permits doctors to create different kinds of reports about patients.
- For a nurse, access to doctor's reports is allowed only if some doctor is present in the ward where the nurse is located, and also if a doctor has given prior approval for such an access. In the activity specification, we use role operation preconditions to satisfy this requirement.
- A nurse is allowed to access records of only those patients who are admitted to the ward where the nurse is currently present. Here we see that a nurse's access to a fine-grain resource (in this case patient records) needs to be controlled based on the context conditions.
- The hospital staff is also able to access utility services, such as a *printer service*, based on their proximity to a particular service. In the activity specification we use context-based service discovery and binding to satisfy this requirement.
- The system allows nurses to leave *place-based alerts* corresponding to various hospital wards. Such alerts are notified to a nurse when he/she enters that ward.

The basic elements of this programming model are *activities*, *objects*, *roles*, and *reactions*. In Figure 2.2 we present a UML class diagram showing the relationships between the various entities defined in the programming model. The activity forms the highest-level abstraction. It is used to model a context-aware application. Abstractions representing users and their tasks (*roles*) and ambient resources and services (*objects*) are encapsulated within the activity abstraction. Within the role abstraction, mechanisms (*operations*) are provided to specify tasks that may be performed by the role members. Abstractions (*preconditions*) are provided to specify and enforce context-based access control and coordination constraints over such task executions. Within the object abstraction, mechanisms (*reactions*) are provided to bind the object to different active space services under different context conditions. Within the activity, *reaction* is used to specify tasks that may be automatically performed when certain context conditions become true.

2.2.1 Activity

A context-aware application is represented and programmed using an abstraction called *activity*. An activity may contain one or more roles, zero or more objects, and zero or more reactions. In an activity, users are represented by their roles, and roles are assigned privileges to perform certain tasks. Shared resources and services are represented as objects in the activity. An activity

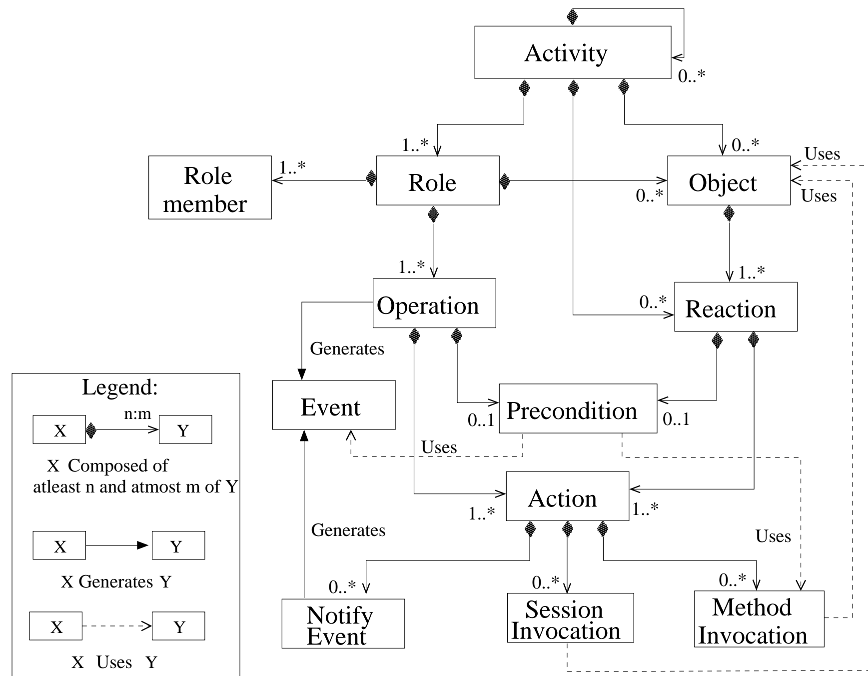


Figure 2.2: Activity class diagram in UML

defines a scope for user roles and shared objects, and specifies policies for their interactions. The objects defined within an activity are shared by all the roles defined in the activity. The reactions are used to program activity-level tasks that need to performed automatically without any user involvement.

Activity activityId
  {**Role** *Role-Definition*}
  {**Object** *Object-Definition* }
  {**Reaction** *Reaction-Definition*}

Figure 2.3: Activity grammar

In Figure 2.3, we present the grammar for an *activity* in the EBNF format. In this notation [ ] represents optional terms, { } represents zero or more terms, | represents choice, and boldface **terms** represent tags for elements and attributes in XML schema that we have developed for activity specifications. Appendix A presents the complete grammar for activity specifications in the EBNF format.

In Figure 2.4 we present the partial specification of the *PatientInformation* activity. In this activity we define two roles (*Nurse* and *Doctor*), and two objects (*PatientDB* and

```
Activity PatientInformationSystem {
    Object PatientDB { ... }
    Object LocationService{ ... }
    Role Nurse {
        Object CurrentWard { ... }
        Object CurrentRoom { ... }
        Object MyPrinter { ... }
        Operation AccessDoctorReports { ... }
        Operation AccessWardPatientInfo { ... }
        Operation Print {
            Action MyPrinter InvokeSession print
        }
    }
    Role Doctor {
        Operation ApprovePatientDataAccess { ... }
        Operation CreatePatientReports {
            Action PatientDB InvokeSession createReports
        }
    }
}
```

Figure 2.4: Patient Information Activity specification

LocationService). The *PatientDB* object refers to the database service that holds the patient records. The *LocationService* object refers to the service providing location information for nurses and doctors in an hospital. A separate instance of an activity can be created by instantiating this activity template corresponding to different departments in a hospital.

2.2.2 Event Model

Events are used within the specifications of context-based access control and coordination requirements. There are two categories of events, *internal* and *external*, that are utilized in our specification model. There are two kinds of *internal* events - system generated events and application-specific events. The system generated events are associated with the execution of role operations. There are two kinds of such events - *start* and *finish*. These are generated by the system upon execution of a role operation. The application-specific events are generated through invocation of a role operation by a role member. Specification of conditions based on internal events was developed in an earlier work [69].

These events are represented by the names of the corresponding operation. Each operation event has two predefined attributes: *invoker* and *time*. All the events related to previously executed operations represent an event list. The specification model supports various functions on event lists. The count-operator `#` returns the numbers events in a list, and a sublist can be obtained by applying a selector predicate. For example, the expression `#(opName(invoker=member(Chairperson)))` returns the number of times a member of

the *Chairperson* role has invoked the operation called `opName`.

External events are generated by context services. Such events are used in a specification to capture policies that depend on the external context. For example, an activity can specify execution of resource binding directives or reactions when certain events occur in the environment. Examples of such external events include user-presence detection, or changes in the physical environmental state. A specification explicitly declares, as part of object requirements, that such events are being "imported" from a service.

2.2.3 Object

Within an activity, an *object* is a logical representation of a service. Objects may be defined within an activity, or they may be defined within a role. Objects that are defined within the activity scope are shared by all roles, i.e. the operations in any role can include method invocations on such objects. In certain applications we may need to distinguish between objects that are shared by all the roles, and objects that are private to each individual role member. In the programming model the later kind of objects are defined within each role. Such objects are *private* to that role i.e., they can be accessed only through that role's operations. Moreover, a separate instance of such an object is created for each member in that role. An object may be dynamically bound to different active space services under different context conditions. Objects are also used to represent the context services present in the active space or on the user's device. Such objects are called as *context objects*. An activity may import context events from such objects.

Object objectId [**RDD** *rddSpec*] {**ImportEvent** *ContextEventDef FilterConditionDef* }
  {[**Reaction** reactionName
    When Event *ContextEventDef*
    Precondition *ConditionDef*]
    Action *BindingAction-Definition*}

BindingAction-Definition
Bind objectId (**Direct** (*URL*) | **Discover**({attribute=value}) | **NewObject** (*codeBase*))

Figure 2.5: Object grammar

Figure 2.5 presents the grammar for an *object*. An object's binding policies are specified as *reactions* within each object's specification. In the programming model, three mechanisms are defined for binding an object to a service. One mechanism is to *directly* specify a service through its network address. Second mechanism is based on *discovering* the required service based on the current context conditions in the environment. This form of binding is useful when the service's location is not known a priori and may need to be discovered in the ambient computing

environment based on the context conditions. The third mechanism is to bind an object to a component that is newly created when an activity is instantiated. Below we present examples from the *PatientInformation* activity to illustrate the various binding mechanisms.

Example 1: The *PatientDB* object is *directly* bound to the database service that provides access to patient records.

```
Object PatientDB { Bind Direct (//PatientDBServiceURL) }
```

Example 2: The *LocationService* object is *directly* bound to the location service running at a well-known URL in the domain. We import *UserArrivalEvent* corresponding to users who are admitted in the *Nurse* role from this service. In the programming model *ImportEvent* primitive is provided for importing context events from context services. The primitive *FilterCondition members(roleId)* can be used to subscribe to events corresponding to the users who are members of a specific role.

```
Object LocationService {
    Bind Direct (//LocationServiceURL) ImportEvent UserArrivalEvent
        FilterCondition members(Nurse)
}
```

Dynamic service discovery and binding is performed by matching an object's description with the descriptions of various services that are registered with the resource discovery service that is deployed in the active space. For this purpose, an object is associated with an *RDD (Resource Description Definition)* specification developed by us. An RDD contains specification of the functional interfaces, attribute-value pairs, and events to be imported from the service. For discovery, certain attributes in an object's RDD are specified at runtime based on the context information.

We present an example of context-based discovery binding in Figure 2.6. As part of the *Nurse* role specification, we define three objects - *CurrentWard*, *CurrentRoom*, and *MyPrinter*. The *CurrentRoom* object refers to the context service of the room in which a nurse is present. The *CurrentWard* object refers to the context service of the ward in which a nurse is present. The *MyPrinter* object refers to the printer in the room where the nurse is present. A separate instance of these objects is created for each member of the nurse role. The binding specification for the *CurrentRoom* object is shown below. The binding reaction is triggered by the *UserArrivalEvent*. Upon triggering, the object is bound to the context service of the room by discovering this service. The nurse's location information (i.e. the room in which the nurse has entered) is used for discovering the room's context service. The identity of the nurse role member is obtained from the *UserArrivalEvent*. Only the binding of the *CurrentRoom* object corresponding to the nurse for whom the *UserArrivalEvent* is generated is modified.

```
Role Nurse {
    Object CurrentRoom RDD (//RoomRDD.xml) {
        Reaction BindCurrentRoom {
            When Event UserArrivalEvent
            Bind Discover (LOCATION=LocationService.getLocation.UserArrivalEvent.getUserName())
        }
    }
    Object MyPrinter RDD (//PrinterRDD.xml) { ... }
}
```

Figure 2.6: Context-based discovery binding example

```
<RDD CATEGORY=CURRENTROOM>
    <INTERFACE>
        <METHOD NAME=presentUserCount>
            <OUTPUTPARAM TYPE=int/>
        </METHOD>
        <METHOD NAME=isPresent>
            <OUTPUTPARAM TYPE=boolean/>
        </METHOD>
    </INTERFACE>
    <IMPORTEVENT>
        <EVENT NAME=RoomStatusChangeEvent/>
    </IMPORTEVENT>
    <ATTRIBUTELIST>
        <ATTRIBUTE NAME=LOCATION VALUE=$PARAM/>
    </ATTRIBUTELIST>
</RDD>
```

Figure 2.7: Parameterized CurrentRoom RDD

The *RoomRDD*, shown in Figure 2.7, is associated with the *CurrentRoom* object. This RDD is used in discovery of and binding to the context service corresponding to the room where the nurse is present. The *LOCATION* attribute in this RDD is a *parameterized attribute*. The value $PARAM of this attribute is specified at runtime using the location information of the nurse role member corresponding to whom the *UserArrivalEvent* is generated. The event *RoomStatusChangeEvent* is specified to be imported from the room's context service.

Different criteria for discovering services in pervasive computing environments have been developed [1, 8, 59, 57]. These include use of attributes (eg: Intentional Naming System (INS) [1]), Java types (eg: Jini), intra-component contracts (eg: PCOM [8]), ontologies (eg: Gaia [59]), and *utility functions* over application requirements and services (eg: Aura [57]). We use RDD for context-based service discovery.

2.2.4 Role

Within an activity, a *role* represents user privileges for performing application tasks. A role defines the following entities.

Operation: Within a role the privileges are represented as *operations*. A role operation can contain one or more actions, which may invoke methods on objects defined in the activity. Associated with each role operation is an optional precondition. Actions are executed only if the precondition is true. The precondition mechanism is used to specify context-based access control requirements and coordination constraints.

Private Objects: Objects may be defined within a role. These are used to specify and maintain the resources that need to be managed separately for every member of the role. Separate instance of an object defined within a role is created for every member of the role.

Role roleId
  {**Object** *Object-Definition*}
  {**Operation** opId
    [**Precondition** *CondifitionDef*]
    {**Action** (objectId (**InvokeMethod** methodName | **InvokeSession** methodList) |
      NotifyEvent appDefinedEventId{*AttrName*=*AttrValue*})}
  }

Figure 2.8: Role grammar

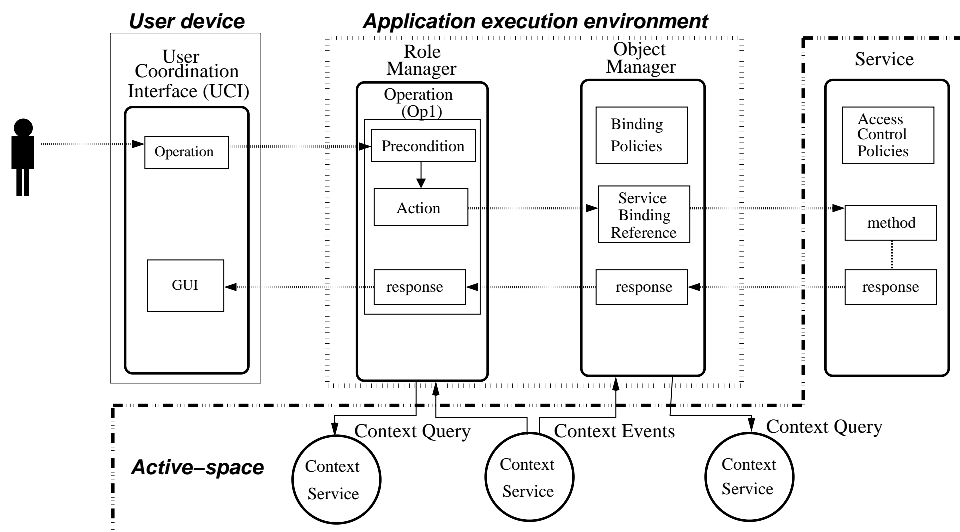


Figure 2.9: Model of user–service interactions

In Figure 2.8 we present the grammar for *role*. Figure 2.9 presents the role-based user–service interaction model in the applications designed using this programming model. A user executes a role operation through the user coordination interface component (UCI) that is created as part of generating the application's execution environment. Through the UCI a user can execute the various roles' operations corresponding to the roles in which the user is a member. The role manager executes the actions defined in an operation in a sequential manner. A role manager enforces the context-based access control policies using the specified preconditions for its operations. The role manager evaluates the precondition associated with the invoked operation. For this it may query the context services deployed in the active space. If the precondition is true, the role manager invokes the specified method on the object specified in that action. In turn, the corresponding object manager invokes that method on the currently bound service.

As shown in Figure 2.10 two models for specifying actions within a role operation are supported.

1. In the *method invocation model*, a user invokes a role operation, which results in only one invocation of a method on the object. In the programming framework the *InvokeMethod* construct is provided for this purpose.

2. In the *session interaction model*, invocation of a role operation by a user results in the initiation of an interactive session between the user and the object. Through such a session, the user may execute an arbitrary sequence of specified methods on the object. In the programming framework the *InvokeSession* construct is provided for this purpose.

Figure 2.10 shows invocation of a role operation which consists of two actions. The first action involves invocation of a single method. The second action results in an interactive session between the user interface and the object.

It is possible that certain active space services may support *transactional methods*. For example, a Java-based service may implement Java's *TransactionManager* interface defined as part of Java's Transaction API [67]. It is possible that an operation contains a sequence of actions involving execution of such transactional methods on different objects. Even though some of the service methods may provide transactional guarantees, we *do not* treat the execution of a role operation itself as a transaction. This is done for two reasons. First, when some action defined within an operation fails, it may not be possible to rollback the effects of all the prior successful actions. For example, it may not be possible to rollback the actions that affect the ambient state. This is similar to the situations that occur in certain workflow systems [37]. Second, partially executed operations may still be useful in certain situations. An example is the case where an operation initiates an interactive session and then fails on a subsequent action. In

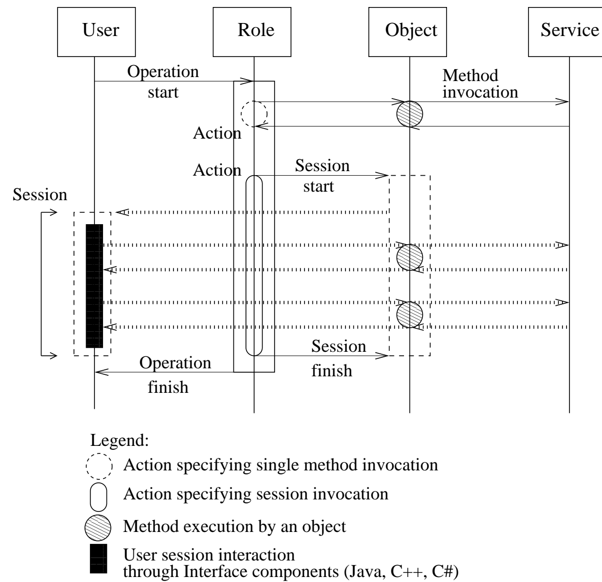


Figure 2.10: Interaction models

some situations the partial execution of the interactive session can still be useful.

We define the following operations in the nurse role: *AccessDoctorReports*, *AccessWardPatientInfo*, and *Print*. The *AccessDoctorReports* operation allows a nurse to access the doctors' reports, and the *AccessWardPatientInfo* operation allows a nurse to access the records of only those patients who are admitted to the ward where the nurse is present. The *Print* operation allows a nurse to print a document to a printer. The *InvokeSession* primitive indicates that this operation follows the session interaction model. The method invocation model is specified using the primitive *InvokeMethod*. In the *Doctor* role we define the following operations: *CreatePatientReports* and *ApprovePatientDataAccess*. The *CreatePatientReports* operation allows a doctor to create various reports about patients. The *ApprovePatientDataAccess* operation allows a doctor to give approval for accessing doctor's reports to a specific nurse.

2.2.5 Reaction

Reaction reactionName
  When Event {*ContextEventDef*}
  [**Precondition** *ConditionDef*]
  {**Action** objectId **InvokeMethod** methodName }

Figure 2.11: Reaction syntax

A reaction is used to program event-triggered actions. Figure 2.11 presents the grammar for a reaction. A reaction follows the Event-Condition-Action (ECA) [14] model of evaluation and is executed by the runtime system. It is triggered by one or more *context events*. The actions specified in a reaction are executed only if the reaction's precondition is true. In case the precondition is not true, the event is not queued. A reaction may invoke methods on one or more objects.

Reactions can be defined at the activity-level and also within each object. Activity-level reactions are used for two purposes. One is to program activity-level tasks that need to be automatically executed when certain context conditions become true. An example of this is notification of alerts to a role member. The second purpose where an activity-level reaction is used is to program activity-level configuration actions which may involve connecting one activity-level object to another. We will see an example of such a configuration in the context-aware music player application in Appendix B. Reactions are also used within an object's definition. One or more such reactions can be specified within an object. The actions that such reactions can execute are restricted to object binding actions.

In Figure 2.12 we show an example of an activity-level reaction. The *NotifyAlerts* reaction is triggered by the *UserArrivalEvent* corresponding to a nurse. Upon triggering it checks if any messages are outstanding for the ward in which the nurse has entered. If so, it notifies these messages to the nurse's UCI.

```
Reaction NotifyAlerts {
    When Event UserArrivalEvent
    Precondition MessageServer.isOutStandingMessage(UserArrivalEvent.getLocation())
    Action Nurse.uci InvokeMethod notifyAlert(UserArrivalEvent.getUserName(),
        MessageServer.getMessages(UserArrivalEvent.getLocation()))
}
```

Figure 2.12: Reaction example

2.2.6 Middleware

In the middleware, three generic components are provided - an *activity manager*, a *role manager*, and an *object manager*. The execution environment of an application is constructed by specializing these managers based on the activity'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 that are defined for an object for dynamic binding are executed by its object manager. All the 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. In Chapter 5 we provide details of the middleware.

Chapter 3

Access Control Model for Context-Aware Applications

In this Chapter we present a role-based access control model for context-aware applications. This model was designed to address the unique access control requirements of such applications. These requirements are related to specification and enforcement of context-based conditions under which users' are allowed to exercise their privileges, specification of privileges that provide access to different resources and services under different context conditions, specification of privileges that provide access to different resources and services to different users based on their context information, and specification of privileges which provide fine-grained access to a subset of resources that are managed by a service under different context conditions.

Role-based access control (RBAC) [62, 27] is a widely used paradigm for designing modern access control systems. The advantages provided by RBAC over traditional access-matrix based models include ease of management, and policy neutrality [55]. In a typical RBAC model, the abstraction of *role* is used as the basic unit of access control. *Permissions* are associated with roles, which effectively grant privileges for accessing *objects* in the system. Users access the objects by exercise permissions of the role to which they have joined. The NIST RBAC model [27] serves as the reference standard for designing RBAC systems in organizations. Here we demonstrate through examples that the NIST RBAC model is not adequate for addressing the access control requirements of context-aware applications.

Other researchers have also developed RBAC models for context-aware pervasive computing applications [61, 19, 53]. Gaia [61] defines three different role categories, corresponding to system-wide roles, active space roles, and application roles, and a mapping between them. In GRBAC model [19] context information is considered as the *environmental role*, which an

application needs to posses in order to perform context-dependent tasks. Such a definition leads to large number of roles in an access control system, as there might be potentially many environmental states that are relevant for an application. In [53], context-based constraints are associated with activation of role permissions. These models do not adequately address the following access control aspects of context-aware applications. First, the models do not support role permissions that are executed on different objects for different users. Second, the models do not support revocation of interactive sessions when context conditions that are required to hold while that session is active fail to hold. Third, these models do not support fine-grained access control requirements that depend on the relationship of a resource's attributes and the context information, such as the identity and the location of the user who is accessing the resource.

3.1 Context-Aware Role-based Access Control Model

In the previous chapter we defined the notion of a *role* as part of an activity in the programming model for context-aware applications. A role essentially represents the privileges for executing tasks that need to be performed by a user of a context-aware application. The concept of role as used by us differs from that used in the organizational RBAC systems as follows. In such systems, roles generally have a long lifetime. Users are assigned to a role by the system administrator, and such memberships also tend to have long duration. In contrast to this, in our model roles are defined as part of an application's design. Such roles come into existence only when that application is deployed and executed, and they last only during the application's lifetime.

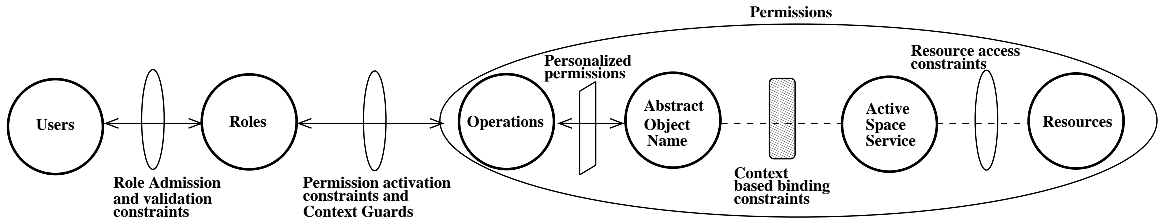


Figure 3.1: Context-Aware RBAC Model (CA-RBAC)

In Figure 3.1 we present the elements of the CA-RBAC model. In this model we support the following kinds of context-based constraints: role admission and validation, personalized permissions, permission activation and context guards, context-based service binding, and resource access constraints. The CA-RBAC model addresses the following requirements of context-aware applications.

- A user's admission to a role and his/her continued membership in a role may need to be

subject to context-based constraints.

- We may need that permission activation by role members must be constrained based on context conditions. Moreover, in certain situations the interactive operation sessions may need to be terminated when the context conditions required for that session fail to hold.
- Certain cases may require that the effect of a role permission execution by different members of a role must be different. Such a permission must allow different role members to access different services based on their individual context information.
- In certain situations we may need that a role member should be given access to different active space services under different context conditions through a specific role permission.
- In certain cases we may need to restrict a role member's access to a subset of resources that are managed by a service. Such a restriction may be based on the context-based relationship between role member context and resources' context-based attributes.

We use two representative context-aware applications to motivate the need for extending the NIST RBAC model to support context-based access control requirements. The first application is the context-aware patient information system, which we presented in Chapter 2. The second application is a context-aware music player application.

Context-Aware Music Player: This application runs on a user's mobile device and has the following context-based requirements.

- Depending on the user's physical location, it streams music either to the user's device or to the audio player service in the room where the user is currently present.
- When the user enters a room, the application should automatically start streaming music to that room's audio player service if no other user is present in the room.
- The access control system must revoke the application's access to the room's audio player service when the user leaves the room or some other person enters the room.

3.1.1 Role Admission and Validation

These constraints are related to users' admission to a role. They were designed as part of Tanvir Ahmed's PhD thesis [2]. They supported use of internal context conditions as part of constraint specifications. In context-aware applications users admission and continued memberships in a role may also depend on external context conditions such as temporal constraints and users' locations. Examples of this are seen in the patient information system. In this application we may define different roles such as *Nurse* and *Doctor* in this system. We may also define

a *NurseOnDuty* role for different wards. Temporal constraints may be used to restrict user membership in the *NurseOnDuty* role only during certain time periods. Prior role membership constraints may be used to restrict only the members of the *Nurse* role to be admitted to the *NurseOnDuty* role for a particular ward. A role member's physical location may be used to allow only those nurses who are physically present in a ward to be admitted to the *NurseOnDuty* role for that ward.

A complementary aspect to context-based role admission is the need to revoke a user's role memberships when specified context conditions fail to hold. For example, in the patient information system a nurse's membership in the *NurseOnDuty* role needs to be revoked when the nurse leaves that ward or after expiration of the specified time interval. We call the above requirement as *role validation constraint.* It determines the validity of a user's membership in a role.

In the NIST RBAC model, there is no explicit notion of role membership revocations. Role membership revocation requirements have been considered by the OASIS RBAC model [5], where a user's membership in a role may be contingent on the membership in some other roles. Some models [41, 9] have used the notion of deactivating a role under certain context conditions. A deactivated role becomes inaccessible to all of its members. In our model, a specific user's privileges for a role are revoked by removing the user from the role membership. Such role membership revocation can be crucial for enforcing policies that may be sensitive to role membership cardinalities.

The programming framework supports two constructs for programming role admission and validation constraints are specified with each role. In Figure 3.2 we illustrate the usage of these constructs as part of the patient information activity. We consider the following two requirements related to a user's memberships in a nurse-on-duty role associated with a ward. First, we require that a user should be admitted to the nurse-on-duty role corresponding to a ward only if he/she is also a member of the *Nurse* role. Second, we require that the user's membership in the nurse-on-duty role of a ward should be revoked if the user goes out of that ward, or after his/her time of duty is over, or if the user's membership in the *Nurse* role is revoked.

In Figure 3.2 we present the partial specification of the *PatientInformationSystem* activity, enforcing these two requirements[1] . We define the *NurseOnDutyEmergencyWard* role and the *EmergencyWardService* object in this activity. We bind the object to a specific ward's context service at the activity instantiation time.

The first requirement is specified through the *AdmissionConstraint* construct. As part of this constraint we check for the user's membership in the *Nurse* role. The function *member(thisUser,*

[1] While presenting activity specifications in pseudo notation we use C++ style comments (//) to write a comment line. In the XML activity specifications no commenting constructs are currently supported.

```
Activity PatientInformationSystem {
    Role NurseOnDutyEmergencyWard {
        AdmissionConstraints { member(thisUser, Nurse) }
        ValidationConstraints {
            EmergencyWardService.isPresent(thisUser) &&
            date == May 5, 2009 && time == 10:00 pm && member(thisUser, Nurse)
        }
    } // end of role
    Object EmergencyWardService {
        Bind Direct (//EmergencyWardServiceURL)
    } // end of object
}
```

Figure 3.2: Role admission and validation constraint example

RoleName) is provided in the programming framework to check the invoking users membership in the role *RoleName*. The variable *thisUser* is a special variable in the programming framework which translates to the identifier of the role member who is requesting admission to the role.

The second requirement is specified through the *ValidationConstraint* construct. As part of this constraint the role manager queries the *EmergencyWardService* object to check whether this role member is present in the ward. If the role member is not present in the ward, or if the current time is past the specified time, or if the user is no longer a member of the *Nurse* role then the user's membership in the *NurseOnDutyEmergencyWard* role is revoked.

3.1.2 Context-based Permission Activation and Context Guard

Execution of role operations by role members may need to be constrained based on context conditions. The access control model needs to support specification of context-based constraints that need to be satisfied *before* a role member may execute an operation.

Certain operation executions may lead to interactive sessions of arbitrary duration with one or more objects. In some applications we may need such sessions to remain active only under certain context conditions. Correspondingly, the access control model needs to provide mechanisms to terminate sessions initiated through the operation execution when the corresponding context conditions fail to hold. We refer to such changes in required context conditions as *context invalidations*.

We see both the above requirements in the patient information system and in the music player application. In the patient information system we need to restrict a nurse's access to patient reports only if the nurse is co-located with a doctor in a hospital ward. This requirement can be modeled as a context-based constraint on operation execution by nurses. The invocation of the operation by a nurse to access patient reports may lead to initiation of a session with the *database service*. We may require that this session should be terminated when either the nurse

or the doctor leaves the ward. This is an example of context invalidation.

In the music player application we require that the access control system should grant access for a room's audio player service to a user only if that user and no other person is present in that room. This can be modeled as a context-based constraint on permission to play the music by the *User* role defined in the application. The context condition corresponding to the user being alone in a room is invalidated when some other person enters the room while the music is being played on the room's audio player service. In this case the access control system needs to revoke the application's access to the room's audio player service. Moreover, this access also needs to be revoked when the user leaves the room.

We observe that such context-based permission activations and revocations are not supported in the NIST RBAC model. The following mechanisms are needed for this purpose. First, we need mechanisms to integrate context information as part of constraint specification on permission activation in a RBAC model. Second, we need mechanisms to continuously monitor context conditions that need to hold while a permission session is active. Third, we need revocation mechanisms to terminate such sessions when the corresponding context conditions fail to hold.

The context-based permission activation constraints are programmed as a *precondition* of a role operation. A role manager evaluates an operation's precondition before the operation's actions can be executed.

```
Role Nurse {
   Object CurrentWard { ... }
   Operation AccessDoctorReports {
      Precondition
         CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
      Action PatientDB InvokeSession accessDoctorReport
   } // end of role operation
   ContextGuard {
      When Event RoomStatusChangeEvent
      GuardCondition
         CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
   } // end of Context Guard
} // end of Nurse role
```

Figure 3.3: Operation precondition and context guard example

In the patient information system we require that a *Nurse* role member may access doctor's patient reports only if a *Doctor* role member is also present in the ward. This requirement is programmed using the precondition construct as shown in Figure 3.3. We define the private object named *CurrentWard* in the *Nurse* role. For each nurse it is bound to the room agent corresponding to the room where that nurse is present. The object's binding changes based on the current location of a nurse. In this regard it is similar to the binding of the *CurrentRoom*

object presented in Figure 3.6.

We define the *AccessDoctorReports* operation through which a *Nurse* role member may access doctor reports. As part of the operation precondition the *Nurse* role manager checks if the *Nurse* role member who is invoking the operation and some member of the *Doctor* role are present in the *CurrentWard*. The function *members(roleId)* is a special function provided in the programming model. It returns the list of the users who are currently admitted to the role *roleId*. The role manager executes the operation's actions only if the precondition is true. Execution of this operation leads to the initiation of an interactive session with the *database service*. As part of this session the role member may invoke *accessDoctorReport* method on the *PatientDB* object. We want that this session be terminated when either the nurse leaves the ward or when no member of the *Doctor* role remains in the ward. In our programming framework we provide the *ContextGuard* construct for addressing the above kind of context invalidation problem.

Context Guard

A context guard may be associated with each role operation individually. It consists of a context predicate that is required to remain valid while a role member's interactive session is active. During the operation execution the associated context guard becomes active. Context guard evaluation is triggered by one or more context events. Every time the specified context event is notified to the role manager, it evaluates the context condition specified through the context guard. If the context condition fails to hold, the role manager terminates the operation session.

We specify a context guard for the *AccessDoctorReports* operation. Its evaluation is triggered by the *RoomStatusChangeEvent* that is notified to the *Nurse* role manager by the ward agent to which the *CurrentWard* object is bound. The *Nurse* role manager terminates the interactive session if either the nurse has left the ward or if no member of the *Doctor* role is present in the ward.

Example 2: In the above example the condition specified as part of the operation precondition, and the context guard was identical. However, it may not be always the case. For example, consider a context-aware exam session application consisting of the *Student* role. Suppose that this role is provided with the *StartExamination* operation through which a student is able to initiate an exam session. We may require that the exam session may be initiated only after some specified time. This condition can be specified through the operation's precondition. We may further require that the session should remain active only for some specified time interval (say 2 hours) after its initiation. This check is specified as part of the context guard. The specification for this example is shown in Figure 3.4.

```
Role Student {
    Operation StartExamination {
        Precondition
            date == May 5, 2009 && time >= 2:00 pm
        Action ExamService InvokeSession write read
    } // end of role operation
    ContextGuard {
        When Event TimerEvent
        GuardCondition
            time <= 4:00 pm
    } // end of Context Guard
} // end of Student role
```

Figure 3.4: Context Guard in exam session

3.1.3 Personalized Role Permissions

The services that are accessible through a role permission may be different for different role members and may depend on the context information associated with a role member. In the NIST RBAC model the set of objects accessed through a permission are always the same for all the members of a role. A role permission invoked by any member of a role is executed on the *same* object. However, this model is inadequate for context-aware applications where a permission invoked by *different* role members may need to be invoked on *different* object instances based on each role member's individual context, such as the physical location.

As an example consider the permission to *print* which is associated with the *Nurse* role in the patient information system. We may require that when this permission is invoked by a nurse, the system chooses the most appropriate printer for satisfying that request. The choice of a particular printer for a specific nurse may depend on various things such as the nurse's preference for a specific printer, context information such as the nurse's location, the physical security of the printer, or the security level associated with the material being printed.

A personalized role permission is programmed as a role operation on a *private* object defined in a role. In Figure 3.5 we show how this requirement is programmed in our framework. In the *Nurse* role we define the *Print* operation through which the private *MyPrinter* object may be accessed. The *MyPrinter* object refers to separate printer service instances for each role member. Therefore, the binding of this object is maintained separately and independently for each *Nurse* role member. The binding action for this object is similar to the binding of the *CurrentRoom* object presented in Figure 3.6.

```
Role Nurse {
    Object MyPrinter RDD (//PrinterRDD.xml) {
        Reaction { ... }
    } // end of MyPrinter object definition
    Operation Print {
        Action MyPrinter InvokeSession print
    } // end of operation
} // end of Nurse Role
```

Figure 3.5: Personalized role permissions

3.1.4 Context-based Role Permissions

In the NIST RBAC model the set of objects for which access control needs to be enforced are statically known. However, in context-aware applications specific services that may be accessed by a role member may not be known a priori. They may depend on the context information associated with an application. The application may need to *discover* an appropriate service in the active space and grant access to it under certain context conditions. Because the specific services with which the application would be interfaced is generally not known a priori, the permissions may only be specified on an abstract object. At runtime, this object is dynamically bound to a specific service available in the active space.

We see above kind of requirement in the context-aware music player application. We need the application to stream music either to the user's device or to the audio player service in the room where the user is present. Moreover, we need the music to stream to the room's audio player service only if no other person is present in the room. To program this requirement we may define an abstract object named *audio player* to represent the service through which the music would be played. This object would be dynamically bound either to the audio player service in the room where the user is present, or to the audio player service on the user's device. We may also define a *User* role and provide the permission to *play* music on the *audio player* object. As part of binding the object we first authenticate the audio player service and then check for its compatibility to allow invocation of methods for playing music.

A context-based permission is programmed as a role operation on an object which may bind to different active space services under different context conditions. In order to illustrate the context-based object binding and the use of the *Reaction* construct, we consider the context-based requirements of the music player application. In Figure 3.6 we present a partial specification of this activity that addresses the above requirement.

We define two objects, *CurrentRoom* and *AudioPlayer*, within the *User* role. Both these are *private* objects of this role. Through the *Event_Dispatch_Order* construct we specify that the *CurrentRoom* object should be bound *before* binding the *AudioPlayer* object. This is crucial for

```
Role User {
    Event_Dispatch_Order {
        Event UserArrivalEvent
            Dispatch_To CurrentRoom
            Dispatch_To AudioPlayer
    }
    Object CurrentRoom RDD (//RoomRDD.xml) {
        Reaction {
            When Event UserArrivalEvent
                Bind Discover (LOCATION=LocationServiceAgent.getLocation(thisUser))
        } // end of binding reaction
    } // end of object definition
    Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
        Reaction {
            When Event UserArrivalEvent
            Precondition CurrentRoom.isPresent(thisUser) &&
                CurrentRoom.presentUserCount() == 1
                Bind Discover (LOCATION=LocationServiceAgent.getLocation(thisUser))
        } // end of reaction
    } // end of object definition
    Operation PlayMusic {
        Precondition AudioPlayer.isBound()
            Action AudioPlayer InvokeMethod play
    } // end of operation
} // end of User Role
```

Figure 3.6: Context-based object binding example

the correct execution of this activity because the *CurrentRoom* object is accessed as part of the condition evaluation of the *AudioPlayer* object's binding reaction. The binding reactions of the *CurrentRoom* object and the *AudioPlayer* object are similar to the binding of the *CurrentRoom* object which was defined in the *Nurse* role and whose specification was presented in Figure 2.6. The difference between the two specifications is that here we use *thisUser* to obtain the user's location. Because the music player application is a single user application, it is possible to utilize *thisUser* in this case.

3.1.5 Context-based Resource Access

There is a need to distinguish between a service and a resource for access control purpose. A service may be managing a number of resources of a specific type. We may need to control a role member's access to a subset of these resources based on context conditions. For example, in the patient information system the *database service* controls access to the database tables and determines who gets access to them and under what conditions. Such a database may store information about patients such as doctor reports, prescriptions, tests performed, last checkup time, and patient's ward. In this application we require that a nurse should be able to access records of only those patients who are admitted to the ward where the nurse is currently located.

This may be satisfied by requiring that the database service grants access for a patient's record to a *Nurse* role member only if that nurse is currently present in the patient's ward. We call such constraints *resource access constraints*. In the literature, similar requirements have been identified and addressed as part of the role graph model [33]. The *parameterized roles* in that model are similar to the *resource access constraints* in our model.

We observe that the requirement of context-based permission activation presented in Section 3.1.2 is inadequate for specifying such constraints requiring fine-grained access control of resources managed by a service. This is because of the following reasons. First, the specific resource to which access needs to be granted may not be known a priori. The resource is only identified at runtime, based on dynamic context information. Second, using permission activation to enforce such access control requirement may also lead to information leakage. A role member would be able to find out information about a resource's attributes simply through the failure or the success of the access attempt without ever requiring to access the resource.

A resource access constraint may be specified separately for every object invocation within a role operation. We define the *AccessConstraint* construct for this purpose in the programming framework.

```
Role Nurse {
   Operation AccessWardPatientInfo {
      Action PatientDB InvokeSession accessPatientInformation
         AccessConstraint (WardID = CurrentWard.getWardID())
   }
} // end of Nurse role
```

Figure 3.7: Resource access constraint example

The requirement that a *Nurse* role member may access the records of only those patients who are admitted to the ward where the nurse is currently present is programmed as shown in Figure 3.7. In the *Nurse* role we define the *AccessWardPatientInfo* operation through which a role member may access patient records. Through the *AccessConstraint* construct we specify the required resource access constraint. It is enforced by the *PatientDB* object manager. The constraint specification consists of the *WardID* attribute of patient records. The *PatientDB* object manager grants access to only those database records for which the *WardID* attribute has the value equal to the location of the *Nurse* role member who is invoking the operation. For this, it queries the *LocationServiceAgent* to obtain the location of the nurse.

3.2 Related Work

Constraints specification as part of RBAC models have been studied earlier [4, 20]. We focus on context-based constraint specification as part of designing context-aware applications. This requires close interaction between the access control system and the context management layer. In our model, application specific context agents are deployed in the system to collect context information and generate context events. The context predicates are evaluated by the application specific role/object managers. A generic framework for context evaluation has been developed as part of the Antigone system [51]. It provides mechanisms to enforce security of the context information used as part of authorization policies. In contrast, our work considers integration of context-based constraints in a RBAC model for pervasive computing applications. This leads to a number of requirements which are not addressed by the Antigone framework.

Context-based constraints that limit a role's visibility to specific geographic areas are presented as part of the GEO-RBAC model [9]. Similarly, the GTRBAC model [41] provides mechanisms for enabling and disabling of roles based on temporal constraints. In our model context-based constraints including temporal and spatial constraints, are specified as part of role admission/validation and role operation preconditions. We argue that this approach supports fine-grained access control requirements, as one can selectively revoke a user's membership from a role, or activate/deactivate specific role permissions, instead of enabling/disabling a role. Moreover, we also support fine-grained access control through *resource access constraints* that are specified as part of a role operation. Additionally, we also provide the context guard mechanism to revoke role operation sessions when specified context conditions fail to hold.

The UCON model [56] provides an extensive framework to model a broad range of *usage control* policies. In contrast, we focus on access control for context-aware applications. There are conceptual similarities between some of the mechanisms in our programming framework and some of the UCON modeling abstractions. For instance, the context guard mechanism in our framework is similar to the UCON's decision predicates that are evaluated during a request execution (*ongoing-authorizations*). Moreover, the UCON's attribute-based access control mechanisms can be used to specify the requirements that are addressed by the *resource access constraints* in our framework. Support for attribute-based access control is also provided by the XACML standard for distributed authorization management [52].

The *resource access constraint* mechanism in our programming framework is similar to the *parameterized roles* of the role graph model [33] and *parameterized privileges* of [34]. The mechanisms of *parameterized roles* [33] and *parameterized privileges* [34] essentially perform access control based on an object's contents. Content-based access control ideas can be traced back to the work on access control based on *data types* in programming languages [40]. In addition to the resource access constraints, we also provide the mechanism of *personalized role*

permissions in our model. The personalized role permissions and the resource access constraint mechanism serve different purposes. Personalized role permissions (private object space for a role member) provide a mechanism to control a role member's access to a service based on his/her context information. Resource access constraints on the other hand provide a mechanism to control a role member's access to a subset of resources managed by a service. A resource access constraint provides a finer granularity of access control as compared to the personalized role permissions.

Distinction between an object type and its instance for access control has also been considered in the TMAC model [68]. In contrast to that model, the abstract objects in our model may be dynamically bound to different services under different context conditions. This requires object managers to verify the authenticity of such services before binding. Such a requirement does not arise in the TMAC model, making its implementation possibly simpler than our model.

Chapter 4

Primitives for Programmed Error Recovery

Robustness of context-aware applications is affected due to their inherent dynamic nature and also due to the dynamic nature of the environments in which the applications are deployed. The dynamic and adaptive nature of context-aware applications causes problems such as incorrect application behavior due to adaptations that are performed at inappropriate times or due to concurrent execution of adaptations [45]. The dynamic nature of the active spaces leads to problems such as unavailability of required resources/services in an active space, service failures, and exceptions thrown by active space services. We experienced these issues as part of the various context-aware applications that we designed in our testbed environment using our high-level programming model.

In this Chapter we develop a model based on *forward error recovery* technique for building robust context-aware applications. Approaches that utilize this technique recover an application from a failure by bringing it to some correct state by executing the recovery actions that are programmed within an application's design. Typically, exception handling mechanisms [35] are used for this purpose. An example of this form of recovery in a context-aware application corresponds to discovering of and binding to an alternate resource if the required kind of resource is not found in an active space. The recovery action consists of the specification of the alternate resource which is used as part of handling the discovery failure.

For building robust context-aware applications the forward error recovery model that we have developed consists of two kinds of mechanisms [45] – asynchronous event communication, and synchronous exception handling. This model arose from our experiences in building several context-aware applications in our testbed environment. We realized that the absence of

application-level programmed error recovery mechanisms within our programming model led to fragile applications, which were unable to cope with various failure conditions.

4.1 Recovery Model and Robustness Primitives

We designed and implemented a recovery model for context-aware applications. It consists of object-level asynchronous event handling mechanisms and role-level synchronous exception handling mechanisms. Events represent a broad class of conditions related to an activity. These include both normal and failure conditions. Exceptions represent a subclass of events that arise synchronously during the execution of an action within a role operation or a reaction.

4.1.1 Object-level Event Handling and Recovery Model

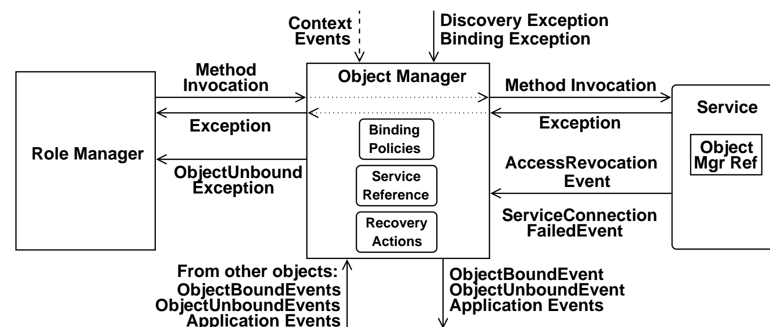


Figure 4.1: Object-level event model

The object-level recovery model consists of events that indicate failure conditions and recovery actions that can be programmed at an object. Figure 4.1 presents this model. The model contains three categories of events: context events, events indicating failure conditions, and events indicating changes in an object's binding state. The context events are delivered to the object manager by context services. The events representing failure conditions are delivered to it by the middleware. The object manager itself generates the binding state change events. Such events from other objects can be delivered to an object manager. The object manager propagates any exceptions that it receives from the bound service to all the role managers that have an on-going session with that object.

Two binding states are defined in the model for each object. These correspond to - bound and unbound. An object manager can be in one of the two states: *unbound*, or *bound* as shown in Figure 4.2. Initially the object manager is in the *unbound* state. The state changes to the *bound* state when the object manager executes a successful binding procedure (i.e. there is no

DiscoveryFailureException) when triggered by a context event. The object manager generates the *ObjectBoundEvent* in this state.

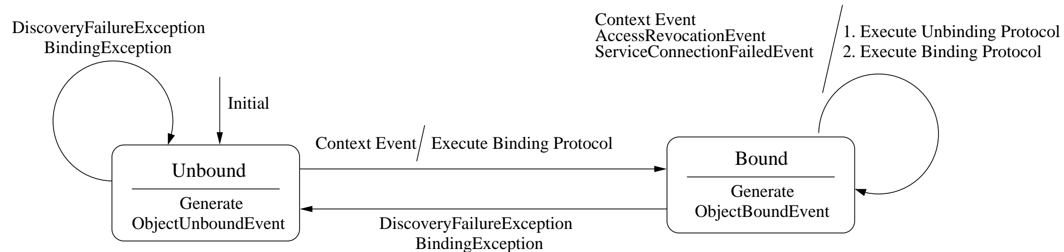


Figure 4.2: Object state diagram

In the bound state, the object manager may receive one of the three different events corresponding to *Context Events*, *AccessRevocationEvent*, and *ServiceConnectionFailedEvent* Binding reactions may be defined with the object corresponding to these events. The object manager handles these events as follows. First, the object manager engages in an unbinding protocol with the service to which the object manager is currently bound. As part of this protocol the object manager unregisters its subscription with the service. The object manager then executes the binding reaction. If the precondition for the reaction is true, then the object manager engages in a binding protocol with the new service with which the object would be bound as part of executing this reaction.

It is possible to prevent an object's binding from changing while some session is currently active with that object. Such a preventive measure can be programmed using the precondition mechanism provided for objects' binding reactions.

Recovery actions to handle local and remote failure conditions can be programmed at the object-level using the reaction mechanism. Such reactions may be defined to be triggered by the local failure events, the binding state change events corresponding to other objects defined in an activity, or any other application specific failure events that are notified within an activity. Figure 4.3 presents the grammar for an object definition with recovery reactions and discovery failure exception handler.

We now consider the various failure conditions and robustness issues encountered at an object. An object manager encounters two kinds of failure conditions - discovery failures and binding failures.

Discovery and Binding Failure

The discovery actions may fail if the required kind of service is not available in the active space. Discovery failure is communicated to the object manager by the discovery service by raising a

Object objectId [**RDD** *rddSpec*] {**ImportEvent** *ContextEventDef FilterConditionDef* }
{[**Reaction** reactionName
When Event (AccessRevocationEvent | ServiceConnectionFailedEvent)
Precondition *Precondition-Definition*]
Action *BindingAction-Definition*}

BindingAction-Definition
Bind objectReference (**Direct** (*URL*) | *DiscoveryBinding-Definition* | **NewObject** (*codeBase*))
OnException BindingException (*BindingActionDef* | **Action NotifyEvent** appDefinedEventId)

DiscoveryBinding-Definition
Bind objectReference **Action Discover**(*[attribute=value]*)
OnException DiscoveryFailureException
Action *BindingAction-Definition*

Figure 4.3: Object syntax with exception handlers

discovery failed exception in response to a discovery request. Binding exception is raised by the middleware when the object manager is not able to bind to the specified service.

Here we show how an alternate task is enabled to recover from a discovery failure. 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 discovers and binds 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.

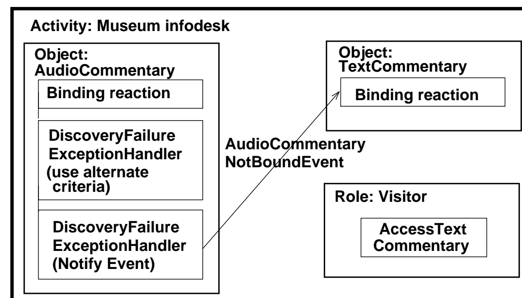


Figure 4.4: Example of object-level recovery

```
Activity MuseumInfodesk {
    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)
        }
    }
    Object TextCommentary RDD (//TextCommentaryRDD.xml) {
        Reaction BindToTextCommentary {
            When Event AudioCommentary.AudioCommentaryNotBoundEvent
            Action Bind
                Discover (ARTIFACT-ID=AudioCommentaryNotBoundEvent.getAttr(ARTIFACT-ID),
                    LANGUAGE=AudioCommentaryNotBoundEvent.getAttr(LANGUAGE))
        }
    }
    Role Visitor {
        Operation AccessTextCommentary
            Precondition TextCommentary.isBound()
            Action TextCommentary InvokeSession readCommentary
        }
    }
}
```

Figure 4.5: Object-level event handling example

In Figure 4.4 we present the schematic of this activity and present its specification in Figure 4.5. This activity will be instantiated on devices that would be given to the museum visitors. Such a device may run a RFID-based context service which would generate *RFIDEvents* when the device is brought in the proximity of an RFID sensor.

In this activity we define two objects: *AudioCommentary* and *TextCommentary*. In the *AudioCommentary* object we define a binding reaction which tries to discover English language audio commentary service. This reaction is triggered by the *RFIDEvent* which is generated when the visitor's device is held close to some RFID tag. In our testbed environment a RFID sensing service is run on user devices. This service detects a passive RFID tag when the device is brought close to a tag. We attach a *DiscoveryFailureException* handler with this binding 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 *AudioCommentaryNotBoundEvent*. 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 through

discovery. In the *Visitor* role we define the *AccessTextCommentary* operation through which the visitor can access the text commentary for an artifact.

Failures due to Binding Termination

A binding termination occurs when the service to which an object is bound crashes or when the service revokes the object's binding to it. These failure conditions (service crashes, and access revocations) are notified as events to the object manager by the middleware. The model distinguishes between binding failures caused due to service crashes and those caused due to access revocations. This is done because one may want to program different recovery strategies for these two cases in some situations.

Failure of an object's binding may require recovery actions to be performed at other objects and roles. For instance consider the music player application. In this application we define the *StreamSender* object, which is the music source. For playing music the *StreamSender* object is connected to the *AudioPlayer* object. Specifically, the address of the audio player service that is bound to the *AudioPlayer* object is configured as the *target* of the *StreamSender*. When the *AudioPlayer* object manager detects that its binding has failed, it needs to notify this event to the *StreamSender* object so that the *StreamSender* object can remove from its target list the address of the audio player service to which the *AudioPlayer* object was bound.

One way to implement the context-based requirement of the music player application to stop the music streaming on a room's audio player service when some other user enters the room is as follows. In this approach the audio player service revokes the access of the music player application when some one else enters the room. The *AudioPlayer* object manager can handle this revocation by binding to the audio player service running on the user's device, as shown in Figure 4.6. When the audio player object successfully binds to the audio player service on the user's device, the *ObjectBoundEvent* is generated. This event triggers the *ConnectStreamSenderToAudioPlayer* reaction, which causes the music streaming to begin on the user's device.

An object manager encounters two kinds of robustness issues which may affect the correct execution of a context-aware application. These are related to the order of binding multiple objects and concurrent processing of context events at an object.

Binding Order Issue

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 correct application behavior. We present here the problem that occurred due to this in the music player application. One of the

```
Activity MusicPlayer {
    Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
        Reaction AccessRevocationHandler {
            When Event AccessRevocationEvent
            Action Bind(//DeviceAudioServiceURL)
        }
    }
    Reaction ConnectStreamSenderToAudioPlayer {
        When Event AudioPlayer.ObjectBoundEvent
        Action AudioPlayer.setSender(StreamSender.getAddress())
        Action StreamSender.addTarget(AudioPlayer.getAddress())
        Action StreamSender.startStreaming()
    }
}
```

Figure 4.6: Access revocation handler

requirement in this application was that the application should automatically start streaming music to the audio player service in a room, only if there is no other person present in that room. In this application we had defined two objects, *CurrentRoom*, and *AudioPlayer*. The *CurrentRoom* object bound to the context agent corresponding to the room in which the user had entered, and the *AudioPlayer* object bound to the audio player service of that room. The binding of both these objects was triggered by the *UserArrivalEvent*, as shown in Figure 4.7.

```
1.Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
2.  Reaction BindToRoomSpeakers {
3.      When Event UserArrivalEvent
4.      Precondition CurrentRoom.presentUserCount() == 1
5.      Action Bind Discover (LOCATION=
6.          LocationServiceAgent.getLocation(UserArrivalEvent.getUserName()))
7.  }
8.} // end of AudioPlayer object
9.Object CurrentRoom RDD (//CurrentRoomRDD.xml) {
10. Reaction BindToRoomAgent {
11.     When Event UserArrivalEvent
12.     Action Bind Discover (LOCATION=
13.         LocationServiceAgent.getLocation(UserArrivalEvent.getUserName()))
14. }
15.} // end of CurrentRoom object
```

Figure 4.7: Issue leading to the binding order requirement

Occasionally, this application failed to bind the *AudioPlayer* object. This happened whenever the binding reaction of the *AudioPlayer* object was triggered before the binding of the *CurrentRoom* object by the *UserArrivalEvent*. The *AudioPlayer* object was left unbound, as its object manager could not query the *CurrentRoom* object for the purpose of precondition

evaluation (line 4), since that object was not bound by that time.

For addressing the binding order requirement, we provide a construct in the programming framework through which one can specify that the order of dispatching a context event to different object managers. For example, using this construct one can specify that the *UserArrivalEvent* should be first dispatched to the *CurrentRoom* object and then to the *AudioPlayer* object, as shown in Figure 4.8.

```
Activity MusicPlayer {
    Object AudioPlayer RDD (//AudioPlayerRDD.xml) {...}
    Object CurrentRoom RDD (//CurrentRoomRDD.xml) {...}
    Event_Dispatch_Order {
        Event UserArrivalEvent
            Dispatch_To CurrentRoom
            Dispatch_To AudioPlayer
    }
}
```

Figure 4.8: Binding order

Concurrent Event Processing Issue

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 the following problem in the music player application due to concurrent processing of context events by the manager of the *AudioPlayer* object defined in this activity. 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, the application failed to satisfy this requirement. In Figure 4.9 we present the specification of the *AudioPlayer* object used in this application. Here the *CurrentRoom* 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 (line 4) but *before* it had executed the binding action (lines 5-6). *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*

```
1.Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
2.  Reaction BindToRoomSpeakers {
3.      When Event UserArrivalEvent
4.      Precondition CurrentRoom.presentUserCount() == 1
5.      Action Bind Discover (LOCATION=
6.      LocationServiceAgent.getLocation(UserArrivalEvent.getUserName()))
7.  }
10. Reaction BindToDeviceSpeakers {
11.     When Event RoomStatusChangeEvent
12.     Precondition CurrentRoom.presentUserCount() > 1
13.     Action Bind Direct (//DeviceAudioPlayerURL)
14. }
15.} // end of AudioPlayer object
```

Figure 4.9: Issue leading to the requirement of atomic processing of context events

is completed (lines 5-6). This causes the application to stream music to the room's speakers even though there is more than one person in the room.

To address the above issue the programming model implementation needs to guarantee *atomicity* of event handling at an object. This is enforced by designing the object manager to handle concurrent context events sequentially.

4.1.2 Role-level Exception Handling Model

The role abstraction contains three *scopes* of execution as shown in Figure 4.10. These correspond to the role scope, role operation scope, and the action scope. The role scope encapsulates the role operation scope, which in turn encapsulates the action scope.

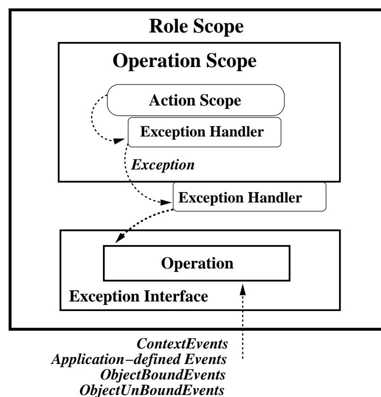


Figure 4.10: Relationship between role scope, operation scope, and action scope

Three categories of failure conditions may be encountered as part of synchronous role operation executions. These include (a) exceptions that are thrown by the service during the

execution of a method on it, (b) failure conditions arising due to the breaking of an object's binding, and (c) context invalidations during an operation execution. The first two kinds of above mentioned failure conditions are encountered at the action scope, whereas the last kind is encountered at the operation scope.

In Figure 4.11 we show the grammar for a role operation with exception handling specification.

Role roleId
{**Operation** opId
[**Precondition** *ConditionDef*]
{**Action** *Action-Definition*
{**OnException** (ObjectRef.Exception | ObjectUnboundException | SessionAbortException
Action *Action-Definition* | **Action** **Throw** TransactionDisruptedException}}
}
[**ContextGuard When Event** *ContextEventDef* **GuardCondition** *ConditionDef*]
{**OnException** (*ActionExceptionDef* | ContextInvalidationException)
Action *Action-Definition*}

Figure 4.11: Syntax for role definition with exception handlers

An exception handler can be attached to both, an action and an operation. This gives greater flexibility in handling the exceptions. The exception can be first handled at the action scope, and then, if required, at the operation scope. Limiting exception handling only at the role operation scope is not considered because of the following reasons. At the role operation scope only limited recovery actions can be performed since the precise exception occurrence context is not available. Moreover, if one action fails the operation block gets terminated. This causes the subsequent actions to not get executed. In contrast, with the ability to perform recovery through exception handling at the action level, the execution of subsequent actions can be continued.

The exception handler itself may contain one or more actions. The exception handling is based on the termination model. The execution of the action that encountered an exception is terminated and an handler for that exception is searched with that action. If a handler is specified and if the exception handling action is successful then the execution of the subsequent actions defined in the operation is resumed. If no handler is specified, the exception is propagated to the role operation scope, which is the statically defined outer scope of an action.

Whether to expose service-level failure conditions in the specification model is an issue. In several applications that we designed and implemented, we realized that sometimes it is necessary to know the exact cause of service-level failure condition in order to handle it correctly. This is particularly important when performing action-level recovery. For example, consider the operation *Play* defined in the context-aware music player. When this operation is executed, the method *playMusic* is invoked on the *StreamSender* object. This method may

throw *IncorrectCodecException* if the *StreamSender* service does not support the selected audio file's format. This exception can be handled by requiring the user to choose another audio file with the correct format.

It is possible that a method is invoked on the object through some role operation or an activity-level reaction when the object is in the unbound state. The object manager raises the *ObjectUnBoundException* in the execution context of all currently ongoing interactive sessions with that object. If no exception handler is defined for an activity-level reaction, the exceptions are propagated to the activity-scope where they can be logged by the activity.

Action-scope Exception Handling

We show an example of action-scope exception handling in Figure 4.12. The *AccessDoctorReports* operation of the *Nurse* role may encounter a *ServiceInteractionException* during its interactions with the patient database service. To handle such situations, one can program an exception handler that invokes the specified method on the backup database service.

```
Role Nurse {
  Operation AccessDoctorReports
    Action DoctorReports InvokeMethod accessReports
      OnException ServiceInteractionException
        Action DoctorReportsBackup
          InvokeMethod accessReports
}
```

Figure 4.12: Example of action-level exception handling

Session-level Exception Handling

Handling of exceptions that are encountered within an interactive session is also an issue. In Chapter 2 Figure 2.10 we showed how session interactions are executed in our model. The interactions are carried out through an interface component. Such components are designed and implemented separately from the activity's specification. The exceptions that arise within such a session would propagate to be raised in the interface component. It is possible that the interface component itself defines exception handlers and executes some recovery actions. This exception handling would be based on the exception handling model of the specific language that is used to implement the component. It is also possible that no exception handler is defined in the interface component. In this case, we require the interface component to notify an event indicating session termination to the object manager. In the model we define the *TerminateSessionEvent* for this purpose. When it receives this event, the object manager terminates the session and raises the

SessionAbortException in the execution context of the corresponding action. This is shown in Figure 4.13.

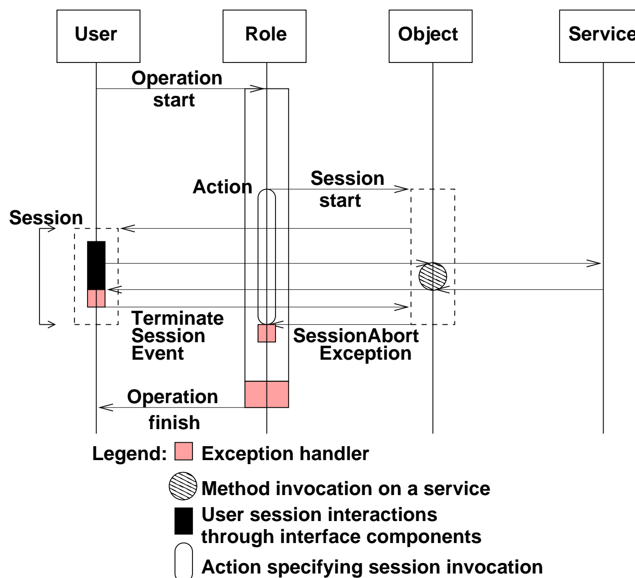


Figure 4.13: Session-level exception handling

Another issue is related to supporting transactional services in our model. A service may provide transactional methods which could be invoked through an interactive session in our model. Exceptions thrown by such methods could be handled in the corresponding action, as was discussed above.

The second type of failure encountered by an interactive session corresponds to the disruptions of transactional sessions when an object's binding fails. This results in terminating the session and the *SessionAbortException* is raised in the operation's action which initiated the session. The designer may want to handle such disrupted transactions as follows. When an object manager rebinds to the service with which the transaction was initiated, it can be directed to either commit, or abort, or resume the disrupted transaction. The design specification model provides a construct through which the designer can specify the required transaction handling policy. For example, one policy could be to direct the object manager to attempt to resume the disrupted transaction when it gets reconnected with that service again. In such a case, if the object manager is successful in resuming the transaction with the service, it would send a transaction resumption event to the role. This event can then be used in enabling an alternate session with the object.

We use the patient information system to demonstrate handling of disrupted transactions. One of the requirements for this application is that a nurse can update patient records only when

he/she is present in the patient's room. For each patient we define a service that provides access to the patient's information. This service provides a transactional session to update patient records.

```
Activity PatientInformationSystem {
    Role Nurse {
        Object PatientRecord { // proximity-based object binding }
        Operation AccessPatientInformation {
            Action PatientRecord InvokeSession update
                OnException SessionAbortException {
                    Action PatientRecord InvokeMethod setTransactionResumption
        }
    }
}
```

Figure 4.14: Handling disrupted transactions: Making disrupted transaction resumable

In Figure 4.14 we present the partial specification of the patient information system application containing handling of transaction failures. The complete specification corresponding to handling of the disrupted transaction is presented in Figure 4.18, after we discuss the mechanism of *exception interface* at a role.

In Figure 4.14 the *AccessPatientInformation* operation is provided in the *Nurse* role through which a nurse can update a patient's records. An exception handler for the *SessionAbortException* is specified with this operation. In this example we want the object manager to resume the transaction when it rebinds to the same patient record service. We program this requirement by specifying invocation of the *setTransactionResumption* interface within the exception handler. This directive tells the object manager to resume the disrupted transaction when it rebinds to that particular patient's record service. The object manager generates the *ResumeTransaction* event when it resumes the transaction. In Figure 4.18 we show how this event is used to enable an operation in the *Nurse* role through which the disrupted transaction can be resumed.

Operation-scope Exception Handling

Exceptions that remain unhandled at the action scope are propagated to the role operation scope. The second type of failure encountered at The operation scope also encounters failures due to context invalidations [46]. These are changes in context conditions that are required to hold during the execution of that operation. The model provides the *ContextInvalidationException* to represent this failure. This exception is delivered to the operation-scope exception handler. To monitor the context conditions associated with a role operation, the design model provides the *ContextGuard* mechanism, which is discussed in Chapter 3.

The recovery model needs mechanisms to handle such terminated sessions due to context invalidations. Different kinds of approaches can be followed to perform recovery when context invalidations are detected. For example, one approach could be for the application to automatically initiate a session with the same service but with reduced privileges. Another option is to allow the user to initiate an alternate session, only after requesting and gaining appropriate permissions from some other privileged users in the application.

One of the requirement in the music player application is to stop streaming music to a room's audio player service when some other person enters the room where the user is currently located. This requirement can be implemented using the context guard mechanism as shown in Figure 4.15. In this activity a context guard is specified with the *Play* operation. The context guard construct consists of two parts. One is the specification of the context events that trigger the evaluation of the context guard's condition. Second is the specification of the context guard condition. If this condition fails to hold, the *ContextInvalidationException* is raised in the operation. This results in the termination of the operation execution.

In this example the execution of the guard is triggered every time the *RoomStatusChangeEvent* is notified to the role manager. The guard condition checks if the number of users in the room where this user is present is more than one. If this is the case the operation execution is terminated and the *ContextInvalidationException* is raised in that operation execution context. An exception handler for this exception is defined with the operation. The exception handling actions consist of removing the address of the room's audio player service from *StreamSender's* target list, and binding the *AudioPlayer* object to the audio player service running on the user's device.

In the music player activity we define the *ConnectStreamSenderToAudioPlayer* reaction to configure the *StreamSender* object to connect to the *AudioPlayer* object. This reaction is triggered by the *ObjectBoundEvent*, which is generated by the AudioPlayer object whenever it is successfully bound to a service.

Role-level Exception Interface

At the role scope the *exception interface* abstraction is provided. It contains a set of operations that may be executed by role members to execute recovery tasks. In a context-aware application user participation in executing recovery tasks is required for the following reasons.

- In certain situations it may not be possible to automatically handle the exceptions that are encountered in the role operation scope.
- In collaborative multi-user applications there arise situations when certain recovery tasks may need to be performed by a role member to assist in recovering from failures that are

```
Activity MusicPlayer {
    Role User {
        Operation Play {
            Precondition AudioPlayer.isBound()
            Action AudioPlayer.setSender(StreamSender.getAddress())
            Action StreamSender.addTarget(AudioPlayer.getAddress())
            Action StreamSender InvokeSession play
        }
        ContextGuard {
                When Event RoomStatusChangeEvent
                GuardCondition LocationService.userCount(LocationService.getLoc(thisUser)) == 1
        }
        OnException ContextInvalidationException {
            Action StreamSender.removeTarget(AudioPlayer.getAddress())
            Action Bind AudioPlayer (//DeviceAudioPlayerURL)
        }
    }
    Reaction ConnectStreamSenderToAudioPlayer {
        When Event AudioPlayer.ObjectBoundEvent
        Action AudioPlayer.setSender(StreamSender.getAddress())
        Action StreamSender.addTarget(AudioPlayer.getAddress())
        Action StreamSender.startStreaming()
    }
}
```

Figure 4.15: Context guard example

encountered by some other role defined in the activity.

- Abnormal situations can arise in the environment which may require actions to be taken by a user in some specific role. Such exceptions have been considered earlier as part of agent-oriented systems as environmental exceptions [71].
- It is possible that certain alternate tasks need to be enabled when an object binding failures are encountered within an activity.

```
Role roleId
    {Operation Operation-Definition}
    [Exception_Interface
        When Event (roleId.opId.Exception | appDefinedEventId | BindingEventDef | ContextEventDef)
        EnableFor (Invoker | ANY | ALL)
        {Operation Operation-Definition} ]
```

Figure 4.16: Syntax for role exception interface

In Figure 4.16 we present the syntax of role-level exception interface. An exception interface operation can be enabled by one of the following kinds of events: an unhandled exception in one of the role's operations, a context event, an application-defined event, and object unbinding

events. Each such event, which we refer to as an *anchor event*, enables execution of an exception interface operation by one or more role members as discussed later in this section. A role member can execute an exception interface operation only when an anchor event for that operation is present in the exception interface queue, and the precondition for the operation is true. An anchor event provides the context for executing the associated exception interface operations. The exception interface supports a queuing model for event delivery and handling. An exception interface operation gets enabled when the anchor event is delivered to the queue of an exception interface operation.

It is possible that multiple users are present in a role. Therefore, a question arises as to which role member should be given the privileges for executing the exception operations. Below we present three examples that motivate three different approaches for granting privileges for executing exception operations.

Execution by a Particular Role Member: Consider the context-aware patient information system in which a *Nurse* role member can access doctor reports only when some doctor is present in the same ward as the nurse. The corresponding role operation encounters a context invalidation exception when a doctor leaves the ward while a nurse is accessing doctor reports. An operation in the nurse's exception interface can be provided through which the nurse can request doctor's permission to access these reports. In this case it is crucial that only the nurse who encountered the context invalidation exception is able to make such a request.

Execution by ANY Role Member: Consider again above example in the patient information system. An operation can be provided in the doctor role's exception interface through which a doctor may grant privileges to the nurse. We can specify that *any* member of the *Doctor* role may grant this privilege to the nurse.

Execution by ALL Role Members: Consider the patient information information system again. It is possible that some new procedure for handling patient records is introduced in the hospital. In the *Nurse* role we may provide an operation corresponding to it. It is possible that this operation encounters an exception. An exception interface operation can be defined in the nurse role to handle this failure condition. It is possible that because this is a new procedure all the members of *Nurse* role need to execute this operation to ensure that every nurse is aware of the recovery actions to handle the failure condition.

The design model provides three qualifiers, *Invoker*, *ALL*, and *ANY* that can be associated with an exception interface operation. The anchor event is dequeued when the exception operation has been executed according to the above qualifiers. 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.

To demonstrate the exception interface mechanism we use two examples. The first example is continuation of the transaction failure example which was presented as part of session-level exception handling. We want that the *PatientRecord* object manager must resume the failed transaction when it rebinds to the service to which it was earlier bound. In Figure 4.17 shows the schematic of the patient information system activity that handles disrupted transactions.

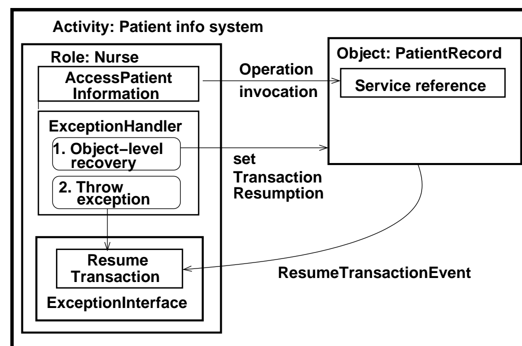


Figure 4.17: Handling transaction failure

```
Activity PatientInformationSystem {
    Role Nurse {
        Object PatientRecord { // proximity-based object binding }
        Operation AccessPatientInformation {
            Action PatientRecord InvokeSession update
                OnException SessionAbortException {
                    Action PatientRecord InvokeMethod setTransactionResumption
                    Action Throw TransactionDisruptedException
        }
        ExceptionInterface {
            When Event TransactionDisruptedException
                EnableFor Invoker
                Operation ResumeTransaction {
                    Precondition
                        #PatientRecord.ResumeTransactionEvent > 0
                    Action PatientRecord InvokeSession update
                }
        }
    }
}
```

Figure 4.18: Handling disrupted transactions using role's Exception Interface

In Figure 4.18 we show the specification corresponding to handling of transaction failures. There are two exception handling actions specified for handling the *SessionAbortException*. The

first action instructs the object manager to resume the transaction when it rebinds to the service. The second action throws the *TransactionDisruptedException* which propagates to the role's exception interface. This exception event acts as the anchor event for the *ResumeTransaction* operation in the nurse role. Through this operation, a nurse whose transaction got disrupted is able to resume the transaction when *ResumeTransactionEvent* is notified by the *PatientRecord* object upon rebinding.

The second example illustrates how recovery actions that require participation from different roles' members can be programmed in our model. For this we use the patient information system. The *Nurse* role is provided with *AccessDoctorReports* operation through which a nurse may access doctor reports. In this application there is a requirement that such an access should be granted only if some doctor is present in the ward where that nurse is present. This condition is specified as the context guard for this operation.

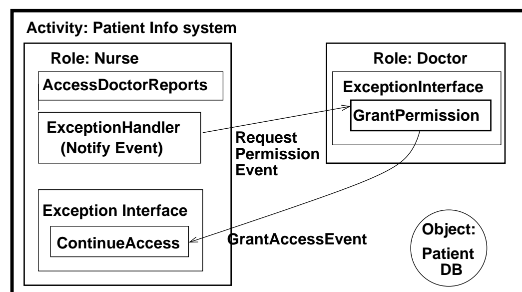


Figure 4.19: Collaborative recovery actions

Figure 4.19 presents the schematic of the patient information activity. In Figure 4.20 we present the specification of the *Nurse* and the *Doctor* role in the patient information system.

Context invalidation exception is raised in the *AccessDoctorReports* operation when no doctor remains in the ward where the nurse is present. When this happens, a design requirement could be that a nurse can resume access to the doctor's reports after requesting and obtaining appropriate privileges from some doctor. An exception handler for the *ContextInvalidationException* is attached with the *AccessDoctorReports* operation. The exception handling action consists of notifying the *RequestPermissionEvent* 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.

In the *Doctor* role, the *GrantPermission* operation is defined in the exception interface. This operation is enabled when the *RequestPermissionEvent* is delivered to the exception interface queue of this role. This operation is specified with the qualifier *ANY*. This means 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 event is subscribed by the *Nurse* role. It is used as the anchor event of the *ContinueAccess* operation, through which the nurse can continue to access doctor reports. This operation is defined in the *Nurse* role's exception interface.

```
Role Nurse {
    Operation AccessDoctorReports {
        Precondition
            CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
        Action PatientDB InvokeSession accessReports
    }
    ContextGuard {
        When Event RoomStatusChangeEvent
            GuardCondition
                CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
    }
    OnException ContextInvalidationException {
        Action NotifyEvent RequestPermissionEvent(requestor=thisUser)
    }
    ExceptionInterface {
        When Event GrantAccessEvent
            EnableFor Invoker
            Operation ContinueAccess
                Action PatientDB InvokeSession accessReports
    }
}
Role Doctor {
    ExceptionInterface {
        When Event RequestPermissionEvent
            EnableFor ANY
            Operation GrantPermission
                Action NotifyEvent
                    GrantAccessEvent(grantee=RequestPermissionEvent.getAttr(requestor))
    }
}
```

Figure 4.20: Exception interface example

4.2 Related Work

Failure handling issues in context-aware applications have been addressed by other researchers [30, 36, 16, 28, 22]. The research presented in [30] addresses issues arising due to service disconnections in pervasive computing environments. Specifically, it addresses the problem of service composition failures due to disconnections. The approach advocated there is based on *state transfer* between identical services to tolerate such disconnections. The binding failures

in our model represent such service disconnection conditions. Additionally, our work addresses other forms of failure conditions such as discovery failures, access revocations, service-level operational failures, and context invalidations. Our work provides a framework for programming application-specific recovery actions for handling such conditions. For example, using the programming primitives presented here one can program a recovery action within an object to handle a service disconnection failure by discovering and binding to another service of the same kind as the original one from a group of replicated services. The problem of coordination and state transfer across services in a group is outside the scope of the design model of our programming framework. It is a design issue for the ambient services so as to provide suitable interfaces to the ambient applications.

System-level robustness issues arising due to concurrent processing of context events have been identified by others [36]. One approach to address these issues is to provide system-level mechanisms for this purpose, as done in [36]. In the recovery model presented here the atomicity in context event processing is ensured by requiring that these events are handled sequentially by the object managers. Moreover, application robustness is also affected by the order in which context events are dispatched to various object managers [45]. In the programming framework presented here a construct is provided through which the order of binding the objects can be specified by the application designer. It is also possible to infer such an order automatically by the middleware by performing dependency analysis of activity specifications.

System-level mechanisms for monitoring application components, and for handling failure conditions have been proposed as part of the Gaia middleware [16], which provides a platform for building active-space applications. These mechanisms include heart-beat based status monitoring, redundant provisioning of alternate services/applications, and restarting failed application components. The recovery model presented here supports object-level monitoring mechanisms to detect object binding failures. System-level failure handling techniques such as replication of various managers on different servers are not supported in our current model. The focus of our work is on application-level programmed error recovery mechanisms rather than on system-level mechanisms for building robust context-aware applications. In Gaia, application level recovery is supported only to recover from policy enforcement failures [?]. In contrast, the recovery model presented here supports mechanisms to handle service discovery failures, object binding failures, service-level exceptions, and context invalidations. It also provides mechanisms that allow user participation in performing recovery tasks.

A data-flow oriented architecture for building dependable pervasive computing systems is presented in [28]. The main objective of that system is to perform failure handling *without* any user involvement. The recovery model that we have designed supports user participation in performing certain recovery tasks. For some context-aware applications, this could be the only

possible mode of recovery.

Similarities can be found in the recovery model presented here and the exception handling models designed for workflow systems [12, 37]. These similarities can be found in regard to execution of alternate actions as part of handling exceptions and user involvement for performing exception handling actions. The recovery model in [22] presents exception handling mechanisms for context-aware agent based systems, and it also identifies the need of human participation in performing recovery actions in context-aware applications. The main distinguishing aspect of our model is that it enables programming of different kinds of recovery patterns, including collaborative recovery actions by members of different roles defined in an activity.

Chapter 5

Design of the Generative Middleware

In this Chapter we present the design of the generative middleware that we developed for realizing the context-aware applications from their high-level specifications. The architecture of this middleware is based on the design of an earlier middleware developed for designing CSCW (Computer Supported Collaborative Work) applications [2]. That middleware formed the starting point for the design of our middleware for context-aware applications.

The goal of the generative middleware is to realize the execution environment of a context-aware application from its high-level XML specification. The design of this generative middleware was guided by the following requirements:

- Generic components are needed for supporting the role-based service interaction model, context-based dynamic object binding, and context-based protocols for access control and coordination.
- Mechanisms are required for specializing the generic components with application specific information to generate the execution environment of an application.
- Modification of a design and the regeneration of the corresponding execution system should be a convenient and efficient process.

The process of generating a context-aware application's execution environment is partitioned into three *phases* as shown in Figure 5.1. The first phase is the *design phase*. The designer has to develop the activity specification corresponding to the application requirements. This specification is called the *activity template*. The designer has to also develop the various context services, and any other ambient computing services that are required by the application.

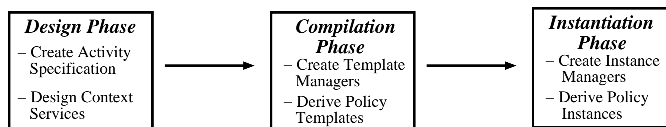


Figure 5.1: Three phases in context-aware application generation

The second phase is the *compilation phase*, which is executed by the middleware for a given activity template. The primary function of the compilation phase is to create the data structures and the policy objects required in generating the execution environment for an activity instance from the activity template. This is done just once for an activity template, thus reducing the time required in its instantiation. The two basic entities created in this phase are the activity's template manager, and the template policies. The *activity template manager* enforces the security policies which identify the roles and their operations that are allowed to instantiate that activity template. It also maintains a representation of the activity template in the form of its XML Document Object Model (DOM) tree. The policy objects created in the compilation phase are maintained in a template form with the activity template manager. The various policies used in this model are discussed in Section 5.2.1 and Section 5.2.2. These policy objects are *parameterized*. This means that the instance specific information such as the names of the various roles and objects are maintained as *parameters* within these template policy objects. The specific values for these parameters are specified only during the third phase.

The third phase is the *instantiation phase*, during which an activity manager is created for the activity being instantiated by the corresponding template manager. The template policies are refined with the instance specific information, such as the names for the various entities defined in the activity. Lastly, activity template managers are created for any activity templates that are defined as sub-activities of the activity being instantiated.

5.1 Generic Managers

The middleware provides the following generic managers.

Role Manager: A role manger provides functionalities and protocols for supporting the user interaction models for accessing various resources and services through role operations. The role manager performs the functions for enforcing admission constraints, operation precondition constraints, and role membership authentication on operation invocations.

Object Manager: An object manager implements the user-interaction models by performing the requested method invocations by role operations. It acts as a proxy for forwarding

these invocations to the service to which it is currently bound. An object manager performs event-triggered dynamic binding actions.

Activity Manager: The primary function of an activity manager is to manage a specific instance of an activity template. An activity manager creates the activity's namespace, the role managers corresponding to the roles defined in the activity, the object managers corresponding to the objects defined in the activity's object-space, and the template managers for any nested sub-activities.

In Figure 5.2 we present an overview of the execution environment and components of a context-aware application. The discovery services, and context services are available in the environment. The application execution environment consists of managers specialized according to the activity's specification. These managers are active objects that support remote interfaces through Java RMI and are run on the *trusted servers* available in the environment. For user interactions, GUI components (called *User Coordination Interfaces (UCI)*) are executed on user devices. Through these, users can perform application tasks by executing role operations corresponding to roles in which they are admitted.

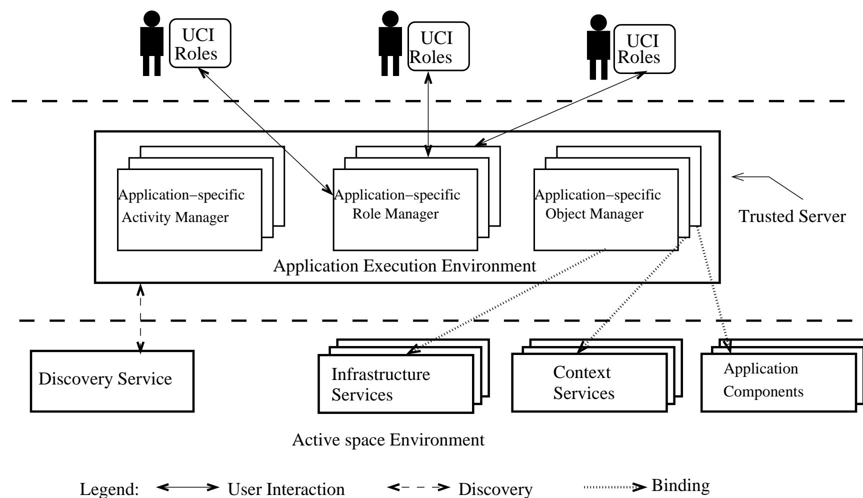
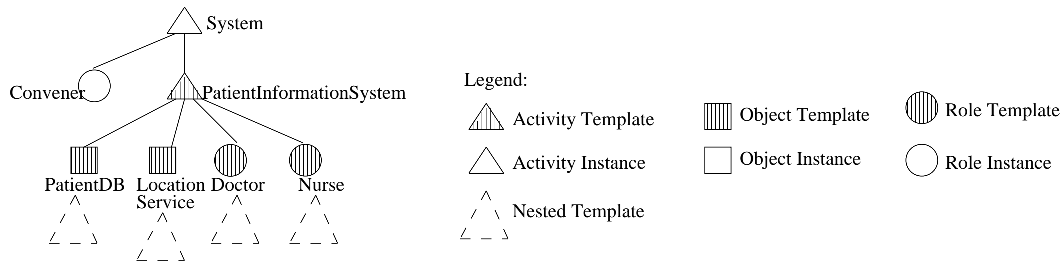


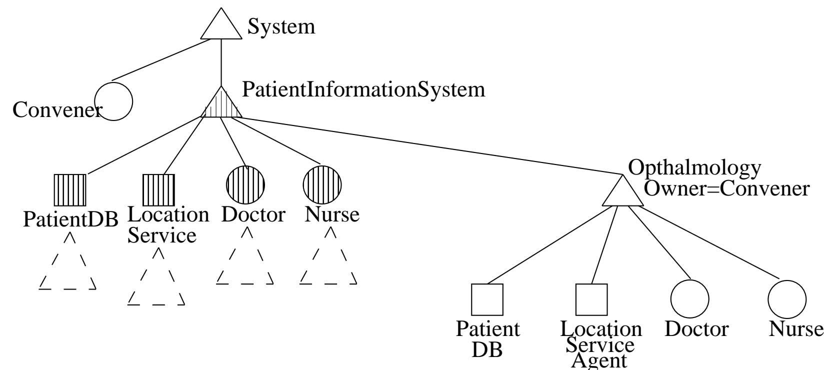
Figure 5.2: Architectural overview of context-aware application generation

5.2 Runtime Representation of Activities

At the heart of the execution management of an activity lies the activity's *XML DOM tree representation*. In Figure 5.3 we present how a template tree for an activity evolves when



(a) PatientInformationSystem activity template



(b) Opthalmology activity instance created

Figure 5.3: Activity DOM tree: Patient Information System activity

different instances of that activity are created. In the figure we only show nodes for activities, roles, and objects. Each of these nodes is a root of a subtree under that node. For example, a role node is the root of the subtree containing one or more operations, an object node is the root of the subtree containing one or more binding directives, etc. In case of an activity template node, all its sub-nodes are template-based. The partial specification for this activity was presented in Figure 2.4

A top-level *System* activity is defined for providing the top-level namespace for any activity template created in the system. In part (a) we show the template for the patient information system activity. A system-defined role, called *Convener*, is responsible for instantiating the activity templates corresponding to the top-level activities. When an activity's template is created, it is appended as a child of the *System* activity. The template managers, created in the compilation phase, maintain pointers to the corresponding nodes in the template tree. For example, the *Activity Template Manager (ATM)* for the *PatientInformationSystem* activity template maintains a pointer to the corresponding activity template node in the DOM tree in Figure 5.3 part (a).

In the instantiation phase, the template tree is *cloned* to create the instance tree for that

activity. The instance tree is appended as a child of the *PatientInformationSystem* activity node. The instance tree corresponding to the activity instance named *Opthalmology* is shown in part (b). This activity defines a new namespace containing nurse and doctor roles that are independent of other activity instances. The names of various nodes in the instance tree are part of the namespace of the activity template which is being instantiated. For example, when the activity instance *Opthalmology* is created, *System.PatientInformationSystem.Opthalmology* is the name of this instance. Because the activity instance names are strongly tied with the scope of the activity template for which the current activity instance is created, the activity instance is appended as a child of the template node. The instance DOM tree structure provides functionalities similar to an attribute tree of a compiler; it maintains tree-based representation of constraints for their execution evaluation.

5.2.1 Policies

The following types of policies are used in our middleware: (a) role admission policies, (b) role operation precondition policies, (c) context-based object binding policies, (d) object-level access control policies, (e) resource access constraints, and (f) event subscription/notification policies. The first two kinds of policies are maintained with a role manager, the following three kinds of policies are maintained with an object manager, and the event subscription/notification policies are maintained by both.

The purpose of these policies is to ensure secure event communication among various role managers, object managers, and the activity manager of an instantiated activity. The *event subscription policies* for a manager specify the other managers that are allowed to subscribe to a given type of event generated by this manager. On the other hand, the *event notification policies* for a manager specify the set of other mangers that are the valid sources for a given event type. Effectively, the notification policies are used for filtering out the events that do not originate from the valid sources. These policies are determined by analyzing the design specification during the compilation phase. Derivation of subscription policies for a role manager requires identification of all the other roles and objects that are allowed to use the events generated by this role. The basic rule in policy derivation is to determine for each role, object, and activity the various kinds of events that its design specification uses as part of the constraint specification.

5.2.2 Policy Derivation for Runtime Management

The core of the compilation phase is the policy derivation procedure. The goal of this procedure is to derive the policies required for the runtime management. Policies are represented in two basic forms: Java objects, or as a reference to the nodes of the activity's DOM tree.

Policies that are specific to only a single entity are represented as pointers to the appropriate constraint specification nodes in the activity's instance tree representation. The policies that are maintained in this form include role admission policies, role operation preconditions involving queries to context agents, context-based object binding policies, and resource access constraints. On the other hand, the event subscription/notification policies, and the object-level access control policies for an object manager are maintained in the form of Java classes. This is because these policies represent the ACLs associated with the corresponding entities and do not require dynamic evaluation of constraint expression.

5.3 Activity Manager Architecture

An activity manager performs the following functions. It creates the role and object managers corresponding to the roles and objects that are defined in the activity. It also creates the reaction threads corresponding to the reactions that are defined in the activity. It derives the role operation subscription/notification policies, and installs them with the corresponding role managers. It also subscribes to the context events that are required by the various managers in the activity from the various context agents in the environment.

5.3.1 Context Event Handling

A context event may have object managers, role managers, and activity level reaction threads, as its subscribers. An event is dispatched first to various object managers, then to role managers, finally to all the activity level reaction threads. The rationale behind this is that all the activity level reactions should be performed only after all the object bindings have been updated. This is because an activity level reaction may involve actions to be performed on objects, and it is crucial that such actions are performed only after an object has been bound to the appropriate resource/service according to current context. Processing of an event by an object manager may lead to the generation of the corresponding binding event for that object. Such a binding event is handled immediately before initiating processing of the next event in the activity's dispatch queue. This ensures that all the effects of a context event are completely known before processing the next context event.

An event may have multiple object managers as its subscribers. We saw in Chapter 4 that the order in which the event is dispatched to the various object managers may be crucial for correct application behavior. In the programming model we provide the *Event_Dispatch_Order* construct for specifying an order. An example demonstrating usage of this construct is presented in Figure 4.8.

It is also possible to automatically infer the binding order, as outlined below. We can

construct an *object dependency graph* of the subscriber objects for each event. Such a graph needs to be constructed only once, at the activity instantiation time. The nodes of this graph represent the *objects* and the directed edges correspond to the *depends-on* relation, which we define as follows. An object *O1 depends-on* an object *O2*, if:

1. *O2* is used in the precondition evaluation of any of the *O1*'s binding reactions, or

2. *O2* is used in the precondition evaluation of some object *O3*'s binding reaction, where *O3*'s binding reaction is triggered by *O1.BindingEvent*.

The *object dependency graph* has to be a *directed acyclic graph* (DAG). An error should be flagged if there are any cycles in it. We require that the programmer modify the specification appropriately in such a case. For example, the following graph will be constructed for the music player activity corresponding to the *UserArrivalEvent*: *AudioPlayer* → *CurrentRoom*. This is because the *AudioPlayer* object *depends-on* the *CurrentRoom* object, because *CurrentRoom* object is queried as part of the *AudioPlayer* object's binding reaction *BindToRoomSpeakers*. This graph can be analyzed to obtain the *object binding order*. In the above example, the *CurrentRoom* object precedes the *AudioPlayer* in the object binding order and hence is bound before the *AudioPlayer* object.

5.3.2 Summary of Context Event Dispatch Rules

- A context event is dispatched to various managers in the following order: object managers, role managers, activity-level reactions.

- If there are more than one object managers that are subscribing a context event and if these managers are related in a binding order then the event is dispatched to them sequentially in the binding order. Otherwise, the event is dispatched to them in parallel.

- The binding order can be specified by the designer, or can be inferred by the activity itself. In the programming model the *Event_Dispatch_Order* construct is provided for this purpose (refer Figure 4.8 for an example).

As a general rule the *context objects* must be bound before binding any other objects defined in the activity. This ensures that most appropriate context information would be available for binding other objects defined in the activity or any of the roles within the activity. In the programming model no distinction is made between *context objects* and other objects. This ensures uniformity in handling both kinds of objects. At the design time a designer needs to know which objects are context objects and should accordingly ensure that they are bound before binding other objects in the activity.

5.3.3 Application Defined Notification Event Handling

Application-defined notification events that are defined within an activity's specification are handled by the *AppDefinedNotifyEventManager* that is provided in the middleware for this purpose. This manager instantiates a generic event for every application-defined event specified in an activity's specification. The manager maintains the subscription/notification policies corresponding to each of these events. It also maintains the count of how many times a particular event has been generated. This manager implements interfaces to set and retrieve an event attributes. These interfaces, *setAttr* and *getAttr*, can be used to set and retrieve arbitrary attribute-value pairs from the defined events.

5.4 Role Manager Architecture

A role manager performs two functions within the constructed runtime environment of a context-aware application. First, a role manager provides mechanisms that enable users to perform application-specific tasks. Second, a role manager provides trusted environment for enforcing task specific context-based coordination and access control policies. In Figure 5.4 we present the components of a role manager and interactions among these components. The shaded components are constructed for each role manager based on the activity specification, whereas the non-shaded components are available as part of the generic role manager architecture.

5.4.1 Role Manager Interfaces

The role manager implements three interfaces: an invocation interface, a subscriber interface, and a notifier interface. The invocation interface supports operation invocations by various role members through the user-coordination interface (UCI). *Tickets* are used as part of the invocation protocol. A ticket contains an operation's name, and its parameters. The role manager verifies the ticket, extracts the operation's name from it, and then dispatches the operation execution by consulting the operation dispatch table. The subscriber interface is used by the other managers to subscribe to the operation *start/finish* events that are generated by this role manager. The notifier interface is used for notifying remote events to this manager. These include operation start/finish events from other role managers, application-defined notification events, and context events that are required for context guard evaluations.

5.4.2 Events and Event Handler

Four kinds of events may be delivered to the role manager including, role operation start/finish events, application defined notification events, context events, and object binding/unbinding

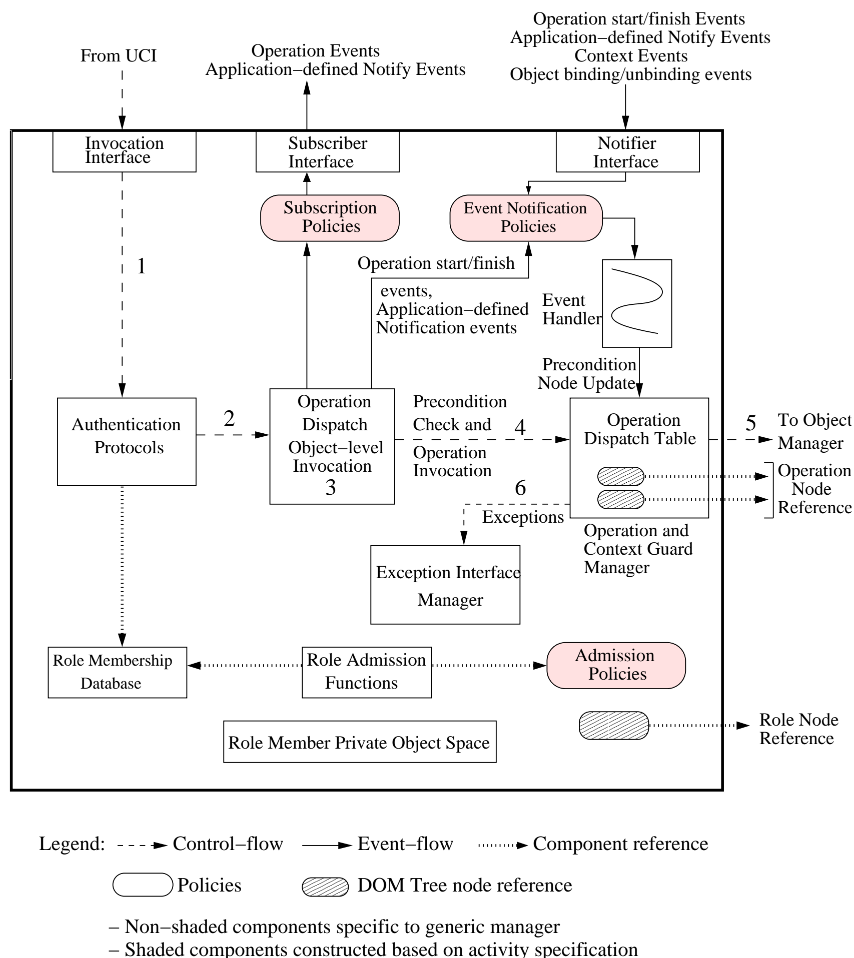


Figure 5.4: Architecture of a Role Manager

events. The event handler maintains the event handling policies that determine what actions to take when a particular event is delivered to the role manager. The operation start/finish events and the application-defined notify events are used to update the precondition trees of all the operations in whose preconditions the specified event is being used. Object binding/unbinding events and application-defined notify events may also be delivered to the role's exception interface manager, if they are specified as an anchor event for some exception interface operation. The context events are delivered to the operation manager to trigger the evaluation of any active context guards.

5.4.3 User Interactions

A user accesses a service by executing a role operation through the UCI. The steps taken by the role manager in response to this are marked as steps 1 to 5 in the Figure 5.4. Authorization *tickets* are used to ensure that role operation invocations are being performed by valid role members. The role manager evaluates the precondition associated with the invoked operation using the DOM tree representation. If the precondition is true, the role manager passes the method name and the method parameters to the object manager that is specified as part of the operation's action. For this, the role manager uses a standard interface implemented by the object manager. The object manager in turn invokes the method on the currently bound service using reflection.

5.4.4 Operation Execution

The role manager contains an operation manager which maintains an operation dispatch table containing references to the operation nodes in the instance DOM tree. An operation may contain one or more actions. Three kinds of actions are supported in our model corresponding to method invocation model, session invocation model, and an action for generating an application defined event.

An action may encounter two kinds of exceptions. Those that are thrown by the service, and *ObjectUnBoundException* which is thrown by the object manager on which the method invocation is specified. When an exception is encountered, a handler for that exception is searched with that action specification. If exception handling is successful, execution of the subsequent actions is continued. If exception handling is unsuccessful, or if no handler is defined for the exception, the underlying exception is wrapped in an instance of the *RoleOperationException*. An exception handler for this exception is then searched at the operation scope. If no handler is found, the exception is delivered to the exception interface manager.

5.4.5 Context Guard Management

When an operation session is started, a context guard manager thread is created to monitor the context guard associated with that operation. The context guard manager evaluates the guard condition whenever a context event specified as part of that guard is notified to the role manager. If the guard condition fails to hold, the context guard manager requests the object manager with whom the session has been initiated to terminate the session and raises the *ContextInvalidationException*. The context guard manager then searches for an exception handler for this exception in the role operation scope. If no handler is specified, the exception

is delivered to the exception interface manager.

5.4.6 Exception Interface Manager

The exception interface manager contains the exception interface queue. It contains interfaces to check whether anchor events corresponding to the exception interface operations that are defined with the role are present in the exception interface queue. The exception interface manager allows the execution of an exception interface operation corresponding to the special qualifiers, *Invoker*, *ANY*, and *ALL*, that may be specified with the exception interface operation. For this, it makes use of attributes that are predefined with every exception and middleware event. Every exception contains the following attributes: the role operation in which the exception was encountered, the name of the role member who encountered the exception, and the time. For enforcing the *Invoker* qualifier, the exception interface manager utilizes the role member attribute of the anchor event.

5.5 Object Manager Architecture

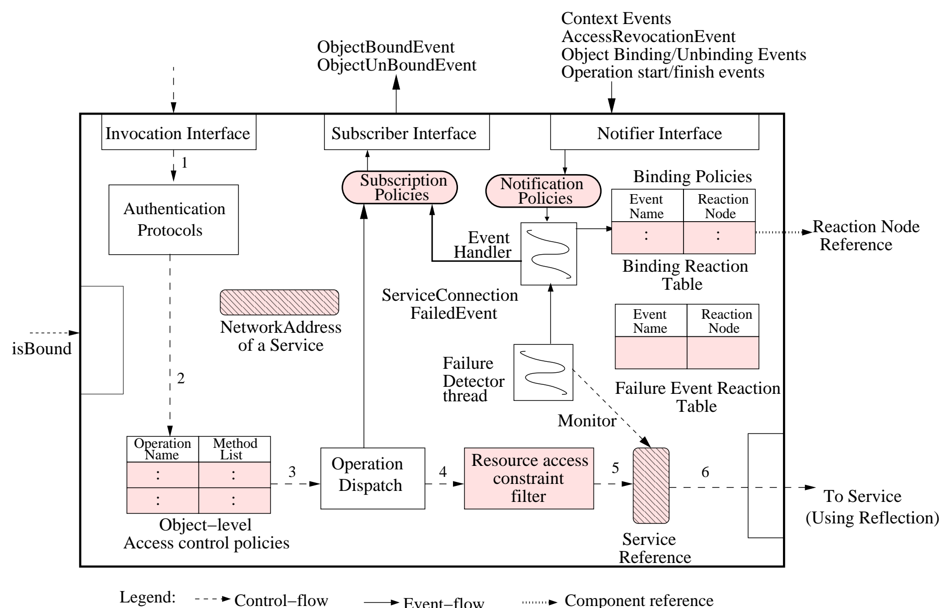


Figure 5.5: Architecture of an Object Manager

An object manager performs two functions within an activity. First, it performs context-based binding actions, thereby maintaining a reference to the appropriate service of that type

under the current context conditions. Second, it enforces method-level access control policies. This involves first, authenticating the role manager which requests method invocation, and second, verifying that the invoked method is permitted as part of the object's ACL, and third, enforcing the resource access constraints. In Figure 5.5 we present the architecture of an object manager derived from the corresponding generic manager. The shaded elements are constructed for each object manager based on the activity specification, whereas the non-shaded elements are available as part of the generic object manager architecture. An object manager contains an event handler thread that executes any event-triggered binding reactions specified for the object.

5.5.1 Object Manager Interfaces

An object manager implements an invocation interface, a subscription interface, and a notifier interface. The invocation interface is used by role managers and UCI to request method invocation on the service to which the object is bound. The subscription interface is used by other managers to subscribe to the events that are generated by the object manager. Each object manager generates *ObjectBoundEvent* and *ObjectUnBoundEvent* whenever the object's binding changes. The notification interface is used for notifying the following events to the object manager. Context events are notified by the context agents, *AccessRevocationEvent* may be notified by the service to which the object is currently bound, and the role operation execution events are notified by the various role managers. The object manager also implements a special interface, *isBound*, which can be used to check whether the object is currently bound or not.

5.5.2 Object Binding Policies

These policies specify the following: *when* to perform an object binding, *how* to identify the service to be used in binding, and *what* context information, if any, should be used in the discovery process. These policies are derived from the activity specification and maintained in the binding reaction table. This table maintains the <*context-event, binding reaction node*> pairs. Each object manager subscribes to all the context events that are required as part of its binding reactions. These reactions get triggered when the subscribed context event is delivered to the object manager. The events delivered to the object manager are handled sequentially by it.

5.5.3 Binding Protocol

An object manager engages in a binding protocol with the service to which it wants to bind. As part of this protocol, the object manager can specify the *mode* in which it wants to bind to the

service. Currently we support two modes: *shared*, and *exclusive*. Default is the *shared* mode. The shared mode indicates that multiple object managers can bind to a service at the same time. This mode is useful when multiple users in a collaborative application wants to share a service. For example, in a context-aware distributed meeting application, all the members of the *Participant* role can bind to the audio player service in the room. This will allow all the members to use that service thereby allowing them to participate in meeting discussions. The exclusive mode indicates that only a single object manager can bind to a service at a time. This mode is useful in cases where exclusive access to a service is crucial for application correctness. For example, in the context-aware music player application it is important that the access to a room's audio player service is given to only a single application at a time. Once an object manager binds to a service in a particular mode, the object manager provides the reference of the activity to the service. This reference is used by the service when it wants to revoke that activity's binding to it by notifying the *AccessRevocationEvent* to it. The object manager may also engage in a mutual authentication protocol with the service[1] .

5.5.4 Binding Failure Detection and Handling

Within an object manager a failure detector thread is defined. This thread periodically pings the service to which the object is currently bound. It generates the *ServiceConnectionFailedEvent* when the failure detector thread is not able to contact to the service. In our implementation we use Java's ConnectException (java.rmi.ConnectException) as an indication of service failure. The failure detector thread notifies this event to the object manager. The object manager maintains the failure handling policies in the failure event reaction table. One of the failure handling options is to try to rebind to the service to which the object was currently bound. For this purpose, the object manager maintains the network address of the service to which the object was last bound. Once an activity's binding with a service is broken, the activity's reference within the service needs to be purged. The service may use mechanisms such as timeouts for this purpose[2] .

5.5.5 User-Service Interactions

The object manager supports two interaction models: the method invocation model, and the session interaction model. A request for method invocation may arrive from a role manager or a UCI. The object manager handles such requests as follows (steps 1 to 6). The object manager first authenticates the role manager or the UCI that is requesting method invocation (step

[1] Currently, no authentication protocol is implemented in our middleware. We can use Ajanta's authentication protocol for this purpose.

[2] Currently, the services that we have designed in our testbed environment do not implement this functionality.

1). It then checks whether the method being invoked satisfies the object's ACLs (step 2). The object manager then constructs the method parameters by applying the specified resource access constraint filter on these parameters (steps 3 and 4). We describe resource access constraints in Section 5.5.7. Finally, using Java's reflection API the object manager invokes the specified method on the currently bound service (steps 5 and 6).

5.5.6 Object-level Access Control Policies

An access control list (ACL) for an object indicates which object interface methods are allowed to be invoked through various role operations. In our model, method invocations on an object may be specified at three places: within a role operation action, within a role operation precondition, or within a reaction. An ACL for an object contains for each method a list of <*role, operation*> pairs, indicating a role and its operation through which that method can be invoked. For the activity-scope objects, such an ACL may refer to any role defined in the activity. On the other hand, for the objects defined within a role, such an ACL can refer to that role's operations only. Similar to the subscription and notification policies, the ACLs for an object are derived and maintained in the template form during the compilation phase, and specialized with activity specific information during the instantiation phase. Below we present an example of this policy for the *PatientDB* object defined in the *PatientInformationSystem* activity specification. The ACL entry specifies that the *Doctor* in the *PatientInformationSystem X* can invoke the *createReports* method on the *PatientDB* object.

```
OBJECT=System.PatientInformationSystem.X.PatientDB
ENTRY{
  SUBJECT=System.PatientInformationSystem.X.Doctor
  OPERATION=System.PatientInformationSystem.X.Doctor.CreatePatientReports
  PERMISSION=createReports
}
```

As an activity is instantiated the above template becomes more specific incorporating the information about the instantiated activity. For example, for the *Opthalmology* activity, the instantiated access control policy looks as shown below:

```
OBJECT=System.PatientInformationSystem.Opthalmology.PatientDB
ENTRY{
  SUBJECT=System.PatientInformationSystem.Opthalmology.Doctor
  OPERATION=System.PatientInformationSystem.Opthalmology.Doctor.CreatePatientReports
  PERMISSION=createReports
}
```

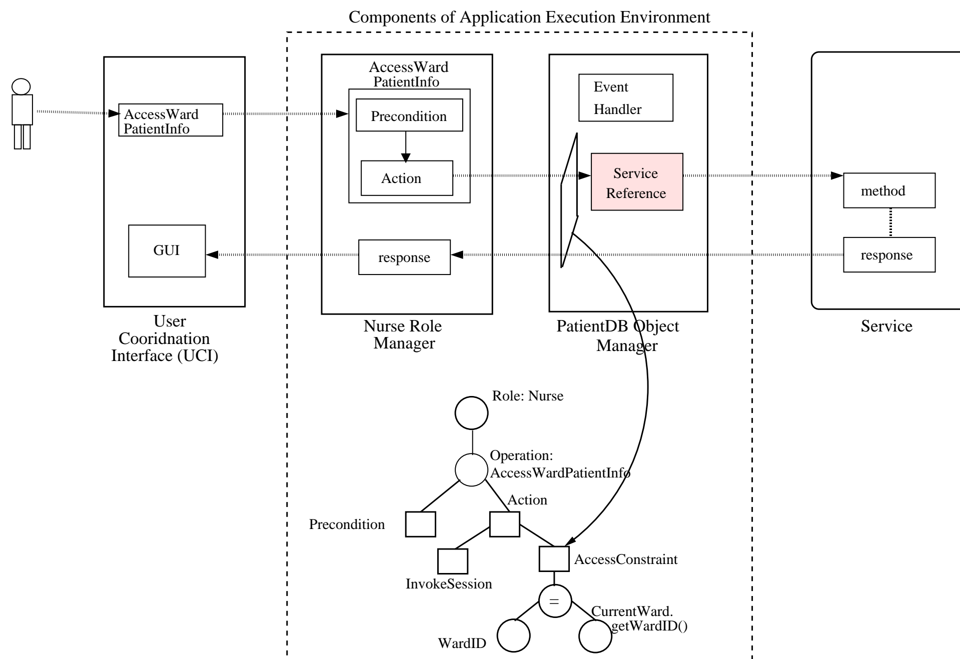


Figure 5.6: Access constraint example: PatientDB object manager

5.5.7 Resource Access Constraints

A resource access constraint is a mechanism that is used by the object manager to enforce context-based access control on resources that are managed by a service. For example, one can use this mechanism to control access to only a subset of database records that are managed by a database service. The object manager constructs a *filter* based on the specified resource access constraints and passes this filter to the service as part of the method invocation. For this purpose the object manager is given a reference to that operation's DOM tree. The service applies the access constraint filter and returns only those resources that satisfy the filter condition. In Figure 5.6 we present the runtime evaluation of the access constraint example which was earlier presented in Chapter 3. The *PatientDB* object manager grants access to only those database records for which the *WardID* attribute has the value equal to the location of the nurse role member who is invoking the operation.

5.6 Related Work

High-level programming models for context-aware applications in active-space environments have been developed and investigated by Gaia [59] and RCSM [75]. In Gaia, active-space

applications are programmed using the Olympus programming environment. A common aspect of our design model and the Gaia system is the use of roles in context-based access control. Gaia provides context-based access control by defining roles that are specific to different physical spaces in an active-space [61]. This provides a coarse-grain integration of context information in access control. In contrast, the mechanisms of role operation precondition and resource access constraint provide fine-grain integration of context in access control policies. The RCSM system provides a context-aware interface definition language for programming context-awareness in applications' components. The principal commonality between our programming framework and these systems is the availability of constructs for service discovery and binding. In our framework, the construct for service discovery and binding explicitly support integration of context information. This is not the case in Gaia and RCSM. Gaia shields context-awareness behind its high-level operators. In RCSM, the various application components are made context-aware through *wrappers* that interface with context services.

Middleware-based systems which support context-based dynamic adaptations for context-aware applications include, Chisel [44], CARISMA [11], PROSE [58], *EgoSpaces* [43], and Rainbow [31]. In Chisel, CARISMA, and PROSE, *reflection* is used as the primary mechanism for performing adaptations. Reflection is also used for dynamic reconfiguration of components in [6]. The primary focus of these systems is on adapting a context-aware application *after* it has been generated. PCOM [8] uses techniques based on distributed constraint satisfaction to find the appropriate components to be integrated with an application. The Rainbow framework [31] provides an infrastructure for hosting and externally adapting context-aware applications. In contrast to the above systems, context-based adaptations in our model are specified as an integral part of a design specification, rather than as independent constraints or policies.

Our high-level design model can be considered as an *architecture description language* (ADL) for specifying dynamic configuration of context-aware applications. The notion of dynamic modification of application configurations expressed in a general purpose ADL such as Darwin [49], or Rapide [48] is conceptually similar to the notion of dynamic object binding in our framework. Our programming framework provides a domain-specific design model for context-aware applications, rather than a general purpose architecture description language. It is specifically developed for user-centric distributed applications with the role-based model as a central element of context-based access control and coordination. It provides abstractions representing users through roles and services through objects. Their configuration and relationships are defined by the constructs of this design model. External components and services are integrated with an application through the binding rules. DiaSpec [13], Aura [57] and the system described in [6] have taken an ADL based approach for building such applications. An advantage of our approach of using a domain-specific model is that it can support use of high

level analysis techniques such as model checking for verifying properties pertaining to application domain. This was demonstrated in a prior work [3].

Our generative programming methodology has similarities with the contemporary generative programming approaches [76, 65, 7]. This similarity is seen in the separation between an activity's template and its instances, and the use of *specialization* for generating an activity instance from the corresponding activity template. There are two novel aspects in which our generative middleware differs from the above mentioned systems. First, unlike the *program generator system* of [65], our middleware is an application generator. The output of the instantiation phase is a running application in which the components may bind to different underlying services under different context conditions. In this regard, the middleware is closer to the component integration models such as Java Beans, and service integration models such as BPEL [15]. Second, we use *policies* for the purpose of specialization of generic components to instantiate the runtime environment of a context-aware collaborative application. The policies capture the dynamic state transitions of the application to be generated. Policies precisely characterize the variation points in the generic components. Policies are generated from the high-level XML specification of an application. Because of this the policies are consistent and coherent. Policies can be directly used to reason about the security properties of a generated application [3]. The high-level programming framework enables changing of the specification easily, thereby enabling realization of context-aware applications satisfying different context-based policies.

The need for providing flexible toolkits for developing CSCW and workflow applications has been recognized in the past [25, 54]. The framework presented in [25] uses *reflection* to specialize a generic toolkit for CSCW applications for the needs of the particular collaborative application that is being developed. A similar approach is followed by DiaSpec [13], which generates an application-specific framework from a generic framework for creating applications in a specific pervasive computing environment. Designers then program the required application using the generated framework. In contrast to these systems, in our framework, a designer only provides a high-level specification of the application. The execution environment is created by the middleware by specializing the generic managers with application-specific policies derived from the specification.

Chapter 6

Design of a Context Detection Infrastructure

The goal of designing context services is to support applications' context-based requirements. Applications may need to detect different kinds of context conditions such as a person's physical location, number and identity of people present in a specific location, arrivals and departures of people from a specific location, or a person's proximity to other people and/or physical objects. Detection of context information requires mechanisms that derive application specific high-level context information from low-level sensor data. This may involve real-time aggregation of sensor streams from distributed locations and inferring the application specific context of interest. Such mechanisms may be deployed at distributed locations in the environment, or they may run on a user's device.

Designing a context detection infrastructure requires mechanisms for *modeling* the required context information and mechanisms for *detecting* the required situation of interest. In the literature different approaches have been used for context modeling [66]. These include attribute-value pairs to represent context elements [64], domain specific ontologies, such as RDF and OWL [73], and the graphical approach involving Object-Role Modeling (ORM) framework [38, 39]. We model context information in terms of entities and the relations between them and use continuous query processing as a mechanism to detect the situations of interest.

6.1 Context Detection

Ambient context detection requires continuous sensing of various different kinds of conditions in the environment, possibly at different locations, and real-time aggregation of the continuous

streams of sensor data to infer the context conditions of interest. An active space may provide infrastructure-level context services that provide generic context information, such as user locations, number of people present in a location, ambient lighting and sound conditions, etc. Different applications may need specific context information. For example, a context-aware patient information system may need information about the arrivals and departures of nurses in a specific ward. Applications may create and install one or more *context services* for this purpose. The context services need to authenticate the sensors from which they would collect sensor information. An application needs to trust the context services from which it obtains the required context information, as it is used within various context-based adaptation actions.

An application may utilize multiple context services to satisfy its requirements. Some of the context services may be running at well-known URLs. An application may bind to such services at the instantiation time. It is also possible that an application may not know about certain context services a priori, but has to *discover* and bind to them at the runtime. For example, the context-aware music player application needs to discover and bind to the context agent corresponding to the room in which the user is currently present.

We saw in previous chapters that within a context-aware application context information may be used to program various context-based adaptations, such as context-based service discovery and binding, context-based access control and multi-user coordination, and context-triggered automated task executions. Expressiveness of context-based constraint specifications within these adaptations depends on the context models that are defined by the application. It is the responsibility of the application designer to define the appropriate context models based on the application requirements. Design of such models also depends on the available sensing technologies. For example, in a patient information system a nurse's location may be modeled at the granularity of a ward or based on the proximity of the nurse to a specific patient in a ward. The available location tracking sensors would determine this granularity.

6.1.1 Context Acquisition

Context-aware applications may use two different programming models for obtaining context information from the context services. These correspond to a query based *pull model* and an event notification based *push model*. In the pull model an application queries the context services for the required context conditions. The context services may support different kinds of querying methods. Certain queries may be *boolean* in nature. For example, *is a particular person present in a particular location?* Other queries may provide a list of values. For example, *who is present in a particular location?*

In the event notification based push model an application subscribes to the required context events from the context services. Such events may correspond to basic sensor events, or they

may correspond to high-level context conditions. It should be possible for applications to install mechanisms to detect context events corresponding to their needs. Such event detectors are essentially *continuous query processors* which continually monitor the specified context condition for the application, and notify an event when such conditions occur.

6.2 Agent-based Context Detection Framework

We use an agent-based distributed event monitoring system [72] for monitoring and gathering external context information. Some agents act as the sources of the basic events in the system as they interface with sensors deployed in the environment. They generate continuous data stream of such events. Multiple agents may be deployed at different locations in the network for monitoring the environment and generating basic event data streams. Some agents in the system subscribe to the event data streams generated by other agents to perform *filtering* and *correlation* functions, to generate a stream of high level context events.

An agent contains three types of basic components: *event detectors*, *event handlers*, and *event dispatchers*. A variety of application-defined event detectors and handlers may be installed in an agent. Interactions between agents are controlled by the subscription policies of the agents. The agent based event processing model is shown in Figure 6.1. An event is a Java object, and events are related in a class hierarchy. A subclass represents a specialized condition of its parent class of events.

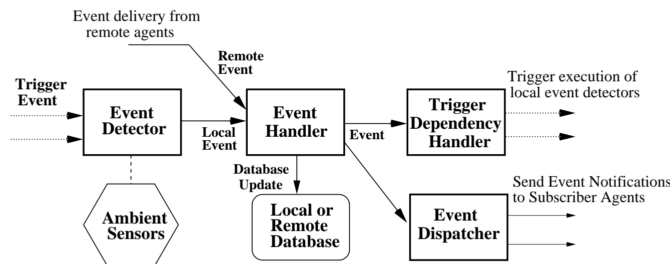


Figure 6.1: Agent's event processing model

A detector is the generator of a specific event type. It acts as the source of the event data stream of that type by periodically executing its event detection function. The event detection function involves either directly monitoring certain conditions in the physical environment for a basic type, or performing filtering or correlating on some input event data streams for detecting a higher level event. A detector's execution is *triggered* either by a periodic timer event or by the occurrence of an event in any of its input data streams. Such events may be locally generated by another co-located detector, or remotely generated by a detector installed on another agent.

Typically, the detectors for the basic event types are triggered by the timer events.

In an agent, a handler object is associated with each event type. When an event is detected or received by an agent, it is given to the handler. The handler object performs the required processing action, such as storing events in local or remote databases. After processing by the handler, the event is given to the Event Dispatcher and the Trigger Dependency Handler.

The event dispatcher sends the event to all the remote subscriber agents registered for that event type. The function of the trigger-dependency handler is to determine if the execution of any other event detector needs to be initiated because of the occurrence of the given event.

6.3 Examples of Context Agents

Here we present the context models that we used for the testbed applications. We define the context events and show the design of the context agents.

6.3.1 Context-Aware Music Player

This application runs on a user's device and streams music either to a room's audio player service or to the user's device, based on context conditions. The following high-level context information is required in this application: the user's arrival and departure from a room, arrivals and departures of other users from that room, and the number of users in that room. We define the event trigger hierarchy shown in Figure 6.2 for this purpose.

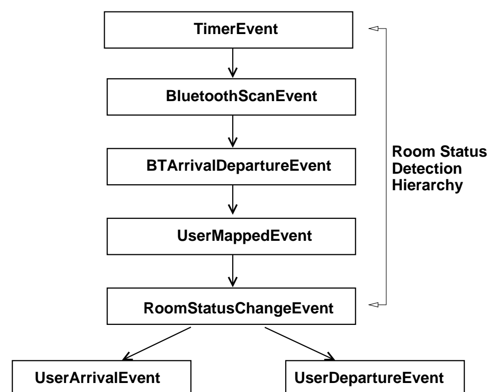


Figure 6.2: Event trigger hierarchy: User location detection

In our environment we use a person's Bluetooth enabled device to track his/her location. We define a *BluetoothScanEvent* detector that is triggered by a timer event. It periodically scans for Bluetooth devices through the Bluetooth discovery mechanism, generating a list of discovered

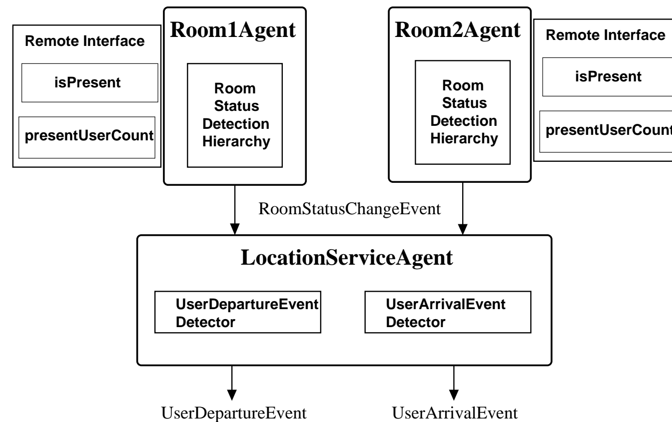


Figure 6.3: Agent configurations: Context-Aware Music Player

devices. The *BluetoothScanEvent* triggers the *BTArrivalDepartureEvent* detector. This detector maintains the Bluetooth device list for the previous scan. On triggering, it compares the current list with the previous list and generates a *BTArrivalDepartureEvent* corresponding to each Bluetooth device that is either newly added to the current list, or that which was detected in the previous scan but absent in the current scan. The arrival/departure status is stored in each event. The *BTArrivalDepartureEvent* triggers the *UserMappedEventDetector* which uses a database to map the Bluetooth device ID to a particular person's name. A *UserMappedEvent* is generated only if the Bluetooth device ID maps to a valid user name known in the system. This event triggers the *RoomStatusChangeEventDetector* which maintains the list of people present in the room and generates the *RoomStatusChangeEvent* corresponding to the user listed in the *UserMappedEvent*. The *RoomStatusChangeEvent* triggers the *UserArrivalEvent* detector and the *UserDepartureEvent* detector. These detectors respectively generate the *UserArrivalEvent* and the *UserDepartureEvent* depending on the status of the Bluetooth device, set to arrival or departure, inside the *RoomStatusChangeEvent*.

We install these detectors on the various agents as shown in Figure 6.3. We define a *room agent* corresponding to each room and deploy all the detectors in the location detection hierarchy except the *UserArrivalEvent* detector and the *UserDepartureEvent* detector. We define a location service agent and install these detectors on that agent. This agent subscribes to the *RoomStatusChangeEvent* from all the room agents.

We use this specific agent configuration for the following reason. In this configuration, the *UserArrival* and *UserDeparture* events are generated by the location service agent which may be configured to run at a well-known URL. The application can bind to this agent at the instantiation time and subscribe to these events in a straight forward manner.

In our agent-based model, causality arising in the physical world is not captured. For example, if a user departs from one room and enters another room, then it is possible that the arrival event is detected and processed before the departure event, if these events are detected by two different context agents that do not communicate with each other.

6.3.2 Patient Information System

In this application we need location information for nurses and doctors. This may be determined through the agent configuration similar to the one used for the music player application. The activity can subscribe to only the user arrival and user departure events correspoding to the nurses by using the following programming idiom.

```
Object LocationService {
    Bind Direct (//LocationServiceURL) ImportEvent UserArrivalEvent
        FilterCondition members(Nurse)
}
```

It is also possible to define a context agent with the *NurseArrivalEvent* detector and the *NurseDepartureEvent* detector. These detectors will be designed to be triggered by the *UserArrivalEvent* and *UserDepartureEvent*, respectively. The detectors have to be configured with the list of users who are currently admitted to the *Nurse* role. Upon triggering, the detectors will determine whether the triggering event denotes an event for any of the nurses. If that is the case the detector will generate the corresponding event.

For this application we also need to know whether two users (a specific nurse and a specific doctor) are co-located in a particular location. For this, the application may query the specific room agent to which it is currently bound to find out if the two users are present in that room. We provide an *isPresent* interface to every room agent using which an application may query the presence of a user in that room.

6.3.3 Museum Application

Consider a museum tour guide deployed in a museum environment and accessed by the visitors. Museum visitors may be given a special device running a *museum guide* application through which they can access this system. The system also provides facilities for a group of visitors to conduct collaborative study projects while visiting the museum. We consider the following context-based requirements for this system. As a visitor enters a museum room, the *museum guide* senses the visitor's presence in that room and automatically connects to the room's audio commentary. When a visitor goes near an exhibit the *museum guide* senses the visitor's proximity to that exhibit and automatically connects to the exhibit's commentary. A group of visitors may

utilize the museum information system for performing group activities. Through the *museum guide* a visitor may post comments and information about an exhibit, or about a museum room, for other group members. Such comments are notified to the group members when they are in the proximity of the exhibit, or when they enter a museum room.

In this application we need two kinds of context information - a user's arrival in a particular room, and a user's *proximity* to an exhibit. Detecting a user's arrival in a room is similar to that in the music player application. For detecting a user's proximity to an exhibit, we use device level detection mechanisms.

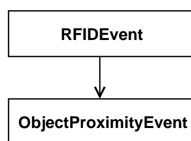


Figure 6.4: Event trigger hierarchy: RFID-based proximity detection

We have emulated a museum environment in our laboratory setting. In this environment we have attached RFID tags to different artifacts such as different posters. In our test-bed environment every user device is equipped with a *passive* RFID reader. For detecting a user's proximity to an exhibit we define the event trigger hierarchy shown in Figure 6.4. The *RFIDEvent* detector generates the *RFIDEvent* whenever a user scans his/her device over any RFID tag. This event triggers the *ObjectProximityEventDetector*, which queries a system-wide database to map the

6.4 Related Work

In the literature different approaches have been used for context modeling [66]. These include attribute-value pairs to represent context elements [64], domain specific ontologies, such as RDF and OWL [73], and the graphical approach involving Object-Role Modeling (ORM) framework [38, 39]. Attribute-value pairs are easy to handle but provide limited functionality. XML schemas support interoperability and shared understanding of a context model among different consumers of the context information. Ontologies provide mechanisms to define relationships between different context elements and domain concepts. The ORM approach [39] provides graphical mechanisms that aid in the development of context models for the domain of interest. Several programming frameworks and toolkits for context detection have been developed [24, 39, 42]. The *Context Toolkit* [24] provides an abstraction of a *context widget*, similar to GUI widgets, to collect context data from distributed sensors. The framework presented in [39] uses a graphical approach for context modeling. A *virtual database* abstraction

for collecting context information, along with a SQL-like language is defined in [42]. In our framework we have designed an agent based context detection infrastructure. The context agents act as continuous query processors that monitor and detect ambient conditions. Application-specific context models and detectors for situations of interest can be constructed from the basic context events provided in the infrastructure.

Chapter 7

Experiments and Evaluations

The goal of this Chapter is to evaluate our programming model. The main advantage provided by our programming model is easy modification and rapid realization of context-aware applications from their high-level specifications. Our first objective is to evaluate this aspect of the programming model. We do this in Section 7.1. In Section 7.2 we consider what fraction of application code is contributed by the middleware, and what fraction needs to be developed for each application separately? This metric is useful in quantifying the amount of effort required from an application developer in order to develop applications using this framework. Section 7.3 is related to characterizing the complexity of designing a context-aware application using our framework. The rationale behind this evaluation is to obtain a quantitative characterization of the difficulty of developing a context-aware application. We use the *coupling* metric for this purpose in this section. In Section 7.4 we consider the time taken to realize an application's execution environment, once the application specific components are designed and implemented. Our intention in presenting this evaluation is to show that once application components and ambient services are developed, realizing the execution environment of a context-aware application can be achieved in reasonable times such as minutes. These measurements indicate that the programming model does indeed support rapid realization of a context-aware application from its high-level specification. We consider the limitations of the programming model and the generative approach in Section 7.5 and comment on its extensibility in Section 7.6. In Appendix B we present the specifications of three representative applications - context-aware music player, context-aware patient information system, and context-aware distributed meeting. We also present the application components and active space services that we designed for these applications.

There are broadly two ways to evaluate the utility of a programming model. One approach is to let several different developers to use the programming model over several years to design

applications using it. Such an evaluation was difficult for our model at this time. Another approach is through a comparative analysis of the model with some other standard widely accepted programming model. One problem with performing such a comparative evaluation in our case is that there is no standard programming model / framework for context-aware applications against which our model and approach can be directly compared. Therefore, instead of evaluating the capabilities and performance of our generative programming model relative to other programming models, we evaluate our model to determine whether it satisfies the goal of supporting rapid design modification and application generation that we had set while designing this model.

7.1 Support for Design Evolution

A context-aware application deployed in an active space may need to be modified due to various factors such as user experiences, changes in administrative policies, or availability of new sensing technologies. A programming framework should provide support for such design modifications. The modifications may be related to minor changes in an already designed activity specification to changes that involve definition of new roles and objects, designing of new active space services, and development of new context models. Here we consider several examples of various design modifications.

Example 1: Consider the *AccessDoctorReports* operation of the *Nurse* role presented in Section 2. It is possible that there could be a change in the administrative policies related to accessing of doctor reports. Specifically, the original policy that a nurse's access to doctor reports should be terminated if no doctor is present in the ward can be modified as follows. We may require that for continuing the access the nurse is required to request a permission from the Doctor. Access can continue only after the doctor grants a permission. Such a change in administrative policy requires generation of a new patient information system application. The specification corresponding to the above policy change is presented in Figure 7.1.

In order to support this requirement the following modifications are required to the specification of the *PatientInformation* activity.

- In the *Nurse* role we provide an operation (*RequestAccess*) through which a nurse can notify an event to the doctor role to request permission for continuing the access to doctor reports. This operation is enabled only when no doctor is present in the ward and when the nurse's *AccessDoctorReports* operation has been terminated.
- In the *Doctor* role we provide an operation (*GrantAccess*) through which a doctor can notify an event to the nurse role to indicate that the role member can continue accessing

```
Role Nurse {
    Operation AccessDoctorReports {
        Precondition
            CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
        Action PatientDB InvokeMethod accessDoctorReport
    }

    Operation RequestAccess {
        Precondition !CurrentWard.isPresent(members(Doctor))
        Action NotifyEvent RequestAccessEvent(requestor=thisUser)
    }
    Operation ContinueAccess {
        Precondition #Doctor.AccessGrantedEvent(grantee=thisUser) > 0
        Action PatientDB InvokeMethod accessDoctorReport
    }
}
Role Doctor {
    Operation GrantAccess {
        Precondition #RequestAccessEvent > 0
        Action NotifyEvent AccessGrantedEvent(grantee=RequestAccessEvent.getAttr(requestor))
    }
}
```

Figure 7.1: Patient information system: Satisfying modified requirements

doctor reports in the absence of the doctor. The specific nurse who requested the access is granted the access by setting the *grantee* attribute in the *AccessGrantedEvent*.

- In the *Nurse* role we provide an operation *ContinueAccess* through which a nurse can continue accessing doctor reports. This operation is enabled only when a doctor grants access by notifying the *AccessGrantedEvent* to this nurse through the invocation of the *GrantAccess* operation.

Summary of changes: We observe that the changes there were required for supporting the modified requirement consisted of the following nature. We had to change the precondition of one operation (*AccessDoctorReports*). We had to add two other operations in the *Nurse* role and one operation in the *Doctor* role.

Example 2: We now consider how same application components and services can be used to support new context-based requirements within the context-aware patient information system. We consider the following new requirements in this application. First, we want to alert a nurse when some doctor is present in the ward where the nurse is present. A nurse can then consult that doctor if required. Second, as a nurse visits different patients' room, we want to keep a log of other members of the patient's team who were present in the patient's room. Such automatic capture of ambient information can become useful later while performing audits related to

patient care data. The specification for this activity is shown below.

```
Activity PatientInformation {
    Reaction NotifyDoctorPresence {
        When Event DoctorArrivalEvent
        Action Nurse.uci InvokeMethod notifyAlert("Doctor Arrived")
    }
    Reaction UpdatePeopleInformation {
        When Event RoomStatusChangeEvent
        Action NurseDB InvokeMethod
            updatePeopleInformation(RoomStatusChangeEvent.getPeopleList())
    }
}
```

- For notifying the presence of doctor in a ward to nurses, an activity-level reaction called *NotifyDoctorPresence* is defined. This reaction is triggered by the *DoctorArrivalEvent*. Doctor's arrival is notified to the nurse through the nurse's UCI.

- For automatically capturing the information about other members who are present in a patient's room during the medical examination, a reaction called *UpdatePeopleInformation* is defined in the activity. This reaction is triggered by the *RoomStatusChangeEvent* corresponding to the room in which a nurse is present. The reaction updates the information about the people who are present in the room by obtaining such a list from the *RoomStatusChangeEvent*.

Summary of changes: For supporting the above requirements the following changes were required. In the context model corresponding to application we had to define the *Doctor-ArrivalEvent* and a detector for this purpose. The *RoomStatuschangeEvent* had to be modified to include the list of the people who are currently present in the room. We had to modify the *NurseDB* service with the *updatePeopleInformation* interface.

Example 3: As a third example we consider the following requirement in the patient information system. In this activity a *PatientDB* object is defined. The normal behavior of this object is that it should bind to the patient database service of the ward in which the nurse is present. For this purpose, the reaction *BindOnNurseArrival* is defined, which is triggered by the *NurseArrivalEvent* indicating the nurse's arrival in a ward. A nurse initiates a session with this object through the *AccessPatientInformation* operation, as shown in Figure 7.2. Once a session is initiated, we may want to prevent the object's binding to change while that session is active. This is achieved by associating a precondition with the *PatientDB* object's binding reaction. While the nurse's session is active, the difference between the counts of the start and finish events corresponding to the *AccessPatientInformation* operation will be greater than zero. This

will cause the precondition of the *PatientDB* object's binding reaction to fail. Consequently, the object's binding will not be affected.

```
Activity PatientInformation {
  Role Nurse {
    Object PatientRecord { ...}
    Operation AccessPatientInformation
      Action PatientDB InvokeSession accessPatientInfo
}
```

Figure 7.2: AccessPatientInformation operation

```
  Object PatientRecord RDD (//PatientDBRDD.xml) {
    Reaction BindOnNurseArrivalEvent
      When Event NurseArrivalEvent
      Precondition #Nurse.AccessPatientInformation.start(invoker=thisUser) −
        #Nurse.AccessPatientInformation.finish(invoker=thisUser) == 0
      Action Bind Discover (LOCATION=LocationServiceAgent.getLocation
          (NurseArrivalEvent.getUserName())
  }
```

Summary of changes: For supporting the above requirement we had to make the following changes to the patient information system activity. The *PatientRecord* object had to be defined within the *Nurse* role. This enabled a separate instance of this object to be maintained for each member of the *Nurse* role. The reaction *BindOnNurseArrivalEvent* for this object was modified to perform the binding action only if a nurse had not initiated an interactive session with the patient database service to which the *PatientRecord* object was currently bound. In the context model corresponding to application we had to define the *NurseArrivalEvent* and a detector for this purpose.

Example 4: The fourth example we consider is related to avoiding of *interference* in context-aware applications. An example of an interference is when actions of one context-aware application erroneously trigger execution of other context-aware application. For example, two applications, one controlling a room's temperature, and another monitoring the room security, may be deployed in a room. The temperature controlling application may open/close the window blinds based on the sensed temperature in the room. The security monitoring application may be designed to raise an alarm upon sensing any motion in the room. These two applications interfere when the security monitoring application raises an alarm upon sensing the motion of the blinds when they are being moved by the temperature controller application.

This form of interference can be handled through appropriate coordination mechanisms between the applications and the ambient services. We show an example that uses status

variables as part of a service's design to avoid the interference problem. In Figure 7.3 we present the specification of the *TemperatureController* activity and the *SecurityMonitoring* activity. The above interference problem is avoided through appropriate design of the blinds controlling service.

```
Activity TemperatureController {
    Object TemperatureMonitoringService { // Direct binding. Import TemperatureChangeEvent. }
    Object BlindsService { // Direct binding }
    Reaction MoveBlinds {
        When Event TemperatureChangeEvent
            Precondition TemperatureMonitoringService.getTemperature() > 30
            Action BlindsService.openBlinds()
    }
}

Activity SecurityMonitoring {
    Object MotionSensingService { // Direct binding. Import MotionDetectedEvent }
    Object BlindsService { // Direct binding }
    Object AlarmService { // Direct binding }
    Reaction RaiseAlarm {
        When Event MotionDetectedEvent
            Precondition !BlindsService.busy()
            Action AlarmService.raiseAlarm()
    }
}
```

Figure 7.3: Avoiding inter-application interference

In the *TemperatureController* activity we define the *MoveBlinds* reaction which is triggered by the *TemperatureChangeEvent*. If the temperature is above the specified threshold, this reaction opens the blinds through the invocation of *openBlinds* method. In the *SecurityMonitoring* activity we define the *RaiseAlarm* reaction. It is triggered by the *MotionDetectedEvent*. If the *BlindsService* is not busy, this reaction raises the security alarm.

The motion sensing service will generate the *MotionDetectedEvent* when the blinds are being opened or closed. In order to avoid the interference problem we need that the *RaiseAlarm* reaction in the *SecurityMonitoring* activity should not get executed when the *MoveBlinds* reaction in the *TemperatureController* activity is moving the window blinds. One way to ensure this is to design the *BlindsService* such that it maintains a status of *busy* when its interfaces corresponding to opening or closing of the blinds are invoked. This status should be checked as part of the preconditions within the reactions that may get affected by this service. For example the *RaiseAlarm* reaction in the *SecurityMonitoring* activity checks whether the *BlindsService* is busy or not before executing its actions.

The service can reset its state to *not busy* after certain time interval. This interval can be determined by taking into account the time it takes to complete the effects of opening or

closing the blinds. For example, in the above case this interval should be set greater than the time it takes for the motion sensing service to detect the motion of the blinds and generate the *MotionDetectedEvent*.

7.2 Code Reusability

The goal of this evaluation is to determine the fraction of the overall code contributed by the middleware in developing a context-aware application. A context-aware application generated using this framework consists of the middleware components that are specialized according to the activity's specification, and the pre-designed application components. The context sensing infrastructure and the discovery service are not considered as part of the generated application.

In Table 7.1 we present the code sizes of the generative middleware, the context sensing infrastructure, and the discovery service. The context sensing infrastructure consists of the location service agent, and agents for various rooms. These agents monitor user locations by tracking their Bluetooth devices. The discovery service is a Java service that provides service registration, and discovery functionality.

Table 7.1: Code sizes of the middleware and various infrastructure services

Category	Java Classes	Codebase size (KB)
Generative Middleware	184	1197
Context sensing infrastructure	95	739.15
Discovery service	9	83.51

The generative middleware includes the generic manager components for activity, role, and object, module for X.509 based protocols for ticketing and authentication, modules for enforcing role admission policies, and role operation precondition policies, protocols for context-based service discovery and binding, modules for enforcing method-level access control policies and access constraints, the policy-derivation subsystem, and a module for creation of user interface components. In Table 7.2 we present the code sizes of these various modules.

In Table 7.3 we present the code sizes for the various applications. The realized application consists of the generative middleware components (1197KB), along with the various application specific components. These components implement an application's basic functionality. An application developer implements these only once. They can then be used as part of generating execution environments for that particular application with different kinds of policies.

We see that more than 75% of the total code for each application is contributed by the generative middleware. If one has to design a context-aware application *without* using this framework then the various middleware modules would need to be implemented by the

Table 7.2: Middleware modules

Module	Java Classes	Codebase size (KB)
X.509 authentication	21	54.13
Role-admission and precondition	29	219.66
Context-based object binding	11	201.01
Method-level access control	4	108.74
Access constraint	3	7.2
Activity management	10	228.77
Policy derivation	60	119.63
User interface management	8	106.58

Table 7.3: Application code sizes

Application	Java Classes	Codebase size (KB)	% Middleware Contribution
Patient Information App (PI)	10	65.76	94.79
Distributed Meeting (DM)	12	136.48	89.76
Music Player (MP)	14	80.08	93.73

application designer. By providing them as part of our framework, we are amortizing the cost of module development across the development of various context-aware applications. One can argue that these modules serve a purpose similar to predesigned software libraries. However, there is a crucial difference. The modules are specialized based on an activity's XML DOM specification to generate an application's execution environment. Software libraries, on the other hand, represent pre-packaged reusable functionality which can be utilized for developing the required application. One has to still code the application logic separately for each application.

7.3 Application Complexity

The execution environment of a context-aware application designed using our framework can be conceptually divided into two parts, the middleware, and the activity's DOM tree. The execution environment is generated automatically by the middleware based on the DOM tree. Based on this observation, one way to characterize the complexity of designing a context-aware application using our framework is to measure the number of components that are dynamically created as part of generating an application's execution environment. For stand-alone application development, such components would need to be designed and programmed by the application designer separately for each application.

The dynamically created components for a context-aware application include the managers for the activities, roles, and objects defined for the application, the subscription and notification

policies, object-binding policies, and the access control lists (ACLs) for various object managers. These policies for various applications are presented in Table 7.4.

Table 7.4: Activity characteristics: various applications

App.	Num of DOM tree nodes	Num of nodes in Policy subtrees	Num of Policy subtrees	Num of Subscription Policies	Num of Notification Policies	Num of Object ACLs	Num of Binding Policies
PI	85	33	6	2	1	13	4
DM	163	32	4	4	1	21	7
MP	170	38	10	3	0	45	8

The number of activity and role managers is exactly equal to the number of activities, and roles defined in the activity specification. One object manager is created for every activity-wide object. For objects that are private to a role, one object manager is created for every object corresponding to every user admitted to the role. The total number of subscription and notification policies involving role operation events depends on the number of event count-based precondition predicates defined in the activity's specification. Specifically, for a single precondition node involving a single operation event count based predicate, one instance of a subscription policy and one instance of a notification policy is derived. The total number of subscription and notification policies is therefore equal to twice the number of precondition nodes containing operation event nodes. Additionally, as part of the subscription policies, we also consider all the context events that are subscribed by an activity. The total number of object-binding policies for an object is equal to the number of binding actions defined for it. With stand-alone application development, all these components would need to be programmed by the application designer, which, in our case, are generated by the middleware.

To characterize the complexity of designing an application using our framework we consider the *coupling* measure [17] between the various components. Two components are coupled if they are related to each other through a method invocation relationship. The coupling metric can give an indication of the number of inter-component dependencies that need to be programmed for a particular application. We use *activity-level coupling* as a metric for these measurements. We define this metric as a combination of role-level coupling for various roles, object-level coupling for various objects, and reaction-level coupling for various activity-level reactions defined in the activity. Role-level coupling for a role is defined as the total number of method based interactions of that role manager with other role managers, object managers, and the policy modules. Object-level coupling for an object is defined as the total number of method based interactions of

that object manager with other object managers, context-services, and the discovery service. Reaction-level coupling for an activity-level reaction is defined as the total number of context events that trigger that reaction, and total number of method based interactions of that reaction thread with other object managers. In Table 7.5 we present coupling values for the various applications. These values are calculated from the corresponding activity specifications. The activity-level coupling indicates the number of inter-component interactions that need to be programmed for each application. These interactions are explicitly specified within an activity's specification. This makes it fairly straightforward to modify these interactions, if and when required.

Table 7.5: Coupling measurements

Application	Role-level coupling	Object-level coupling	Reaction-level coupling	Activity-level coupling
PI	10	13	0	23
DM	15	19	4	38
MP	14	25	22	61

7.4 Performance of Activity Instantiation Procedures

The goal of this evaluation is to measure how long does it take for the generative middleware to construct an application's execution environment. All the experiments were done on an Intel Pentium 4 CPU 2.6GHz machine with 1GB of RAM.

Table 7.6: Activity instantiation times: various applications

App.	Roles	Objects	XML Instance Tree Creation (seconds)	Policy Derivation (seconds)	Manager Instantiation (seconds)	Activity Instantiation (seconds)
PI	2	5	2.9	0.17	74.10	78.94
DM	2	7	6.15	0.20	100.75	110.05
MP	1	4	6.71	0.05	68.18	77.33

In Table 7.6 we present average values (over three experiments) for the three steps involved in activity instantiation. These steps are: (1) creation of XML DOM tree for the activity instance, (2) policy derivation, and (3) creation of various managers. The activity instantiation time includes the time taken by all the three steps.

We make the following observations from these two tables. First, the time for XML instance tree creation step (fourth column of Table 7.6) is directly proportional to the number of nodes present in the XML DOM tree (second column of Table 7.4). Second, the time for policy derivation step (fifth column of Table 7.6) is directly proportional to the number of subscription and notification policies derived for the activity (fifth and sixth columns of Table 7.4). Third, the time for the manager instantiation step (column sixth of Table 7.6) is directly proportional to the number of roles and objects defined in the activity, and the number of subscription policies derived for an activity. When a role manager is instantiated, it has to establish event subscriptions with other role managers.

7.5 Limitation

The generative approach does not permit modification and extensions to an application's execution environment that has been already created. The modification is not possible because the policies are derived and integrated with the middleware components as part of the application generation process. A completely new execution environment needs to be generated when any changes are required in any of the policies related to an already instantiated application.

We believe that this limitation can be overcome for those context-based requirements for which it is sufficient to evolve an activity by *adding* new roles, objects, or reactions that are independent from the previously defined entities in that activity. The middleware can generate the new managers corresponding to the newly defined entities. This approach needs further research.

7.6 Extensibility

The design of the programming model for context-aware applications presented here was conceived as an extension to an earlier programming model which was designed for building Computer Supported Collaborative Work (CSCW) applications [2]. That programming model supported the concepts of an activity, roles, and objects. It also supported mechanisms for specifying and enforcing coordination constraints. However, that model did not contain mechanisms for context-aware computing. Specifically, there were no mechanisms for context-based service discovery and binding, context-based access control, and context-triggered task executions. The model in [2] was extended in the following ways to support designing of context-aware applications.

- *Access control enhancements:* We developed the CA-RBAC model to support context-based access control requirements. We developed discovery and binding mechanisms for

supporting an object's dynamic binding to different active space services under different context conditions.

- *Robustness related enhancements:* The model presented in [2] did not support primitives for designing robust context-aware applications. In the extended programming model we developed object-level and role-level recovery primitives for supporting robust designs of context-aware applications.

- *Programming model enhancements:* A new manager for handling activity-level reactions had to be introduced. The policy derivation routines were modified to take into consideration context-based preconditions and context-based resource access constraints.

- *Infrastructure enhancements:* We developed an agent-based distributed context detection infrastructure for supporting various context-aware applications. Various context agents were developed in the testbed environment for this purpose.

Chapter 8

Conclusions

But if the general truths of Logic are of such a nature that when presented to the mind they at once command assent, wherein consists the difficulty of constructing the Science of Logic? Not, it may be answered, in collecting the materials of knowledge, but in discriminating their nature, and determining their mutual place and relation.
- George Boole, An Investigation of the Laws of Thought, 1854.

In this thesis we have made the following contributions towards robust designs of context-aware applications. We have designed a high-level programming model for supporting the generative paradigm for developing context-aware applications. This model provides abstractions for representing a context-aware application, users and their tasks within such a application, and resource and services required by the application. Mechanisms are provided in the model for dynamically discovering and binding the application to an active space service. Such binding actions may be based on context information. A policy-driven middleware is used for generating the execution environment of an application. A set of generic components are specialized based on the policy objects derived from the design specification to generate an application's execution environment. These policy objects are related to role-based access control, event communication, and dynamic binding of objects. The middleware provides mechanisms for distributed management of different components of an application's execution environment.

We have developed the Context-Aware Role-based Access Control (CA-RBAC) model for specifying and enforcing context-based access control requirements of context-aware applications. We have demonstrated the need for extending the NIST RBAC model for addressing context-based access control requirements of pervasive computing applications. We presented our context-aware RBAC model that supports several such extensions. The model supports

personalized permissions for role members, and context-based constraint specification as part of - dynamic binding of objects with active space services, user admission to roles, permission executions by role members, and granting access to a subset of a service's resources based on a role member's context information. The model also supports revocation of an ongoing task when the context conditions that are required to hold during that task fail to hold. We have designed the *ContextGuard* mechanism in the CA-RBAC model to enforce such task revocations.

In the programming model we have developed exception and event handling primitives towards the goal of building robust context-aware applications. These primitives are part of a forward error recovery model integrated in our programming framework. This model combines event handling at the object-level with exception handling at the role-level to build robust context-aware applications. At the object-level *reactions* can be defined to handle failure conditions such as service crashes and access revocations. Such handlers are used for performing automatic recovery actions at an object. At the role-level exception handlers can be defined at two scopes - role operation and role operation action. The exception handling model follows the termination semantics. 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.

During designing of the high-level programming model and the generative middleware we ran into issues related to concurrent handling of context events by a context-aware application. We saw that an application's correct behavior may depend on the order in which context events are handled. This problem came forth after extensive experimentation with the context-aware music player application. In retrospect, we believe that this problem could have been identified earlier if we would have had a way to analyze the application's specification. We believe that model checking approaches could be useful for this purpose. Model checking would enable verification of context-based properties that an application needs to satisfy when deployed in a particular active space. Several research issues arise in this regard: What are the important context-based properties? How to model these properties? Large number of contextual states may lead to large state space in the modeling. How to overcome the corresponding state space explosion problem in verification? How to realize a context-aware application from its verified model? Future research should be devoted towards seeking answers to these questions.

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," in *17th ACM Symposium on Operating Systems Principles*, December 1999, pp. 186–201.

[2] T. Ahmed, "Policy based design of secure distributed collaboration systems," Ph.D. dissertation, University of Minnesota TwinCities, 2004.

[3] T. Ahmed and A. R. Tripathi, "Specification and Verification of Security Requirements in a Programming Model for Decentralized CSCW Systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 2, p. 7, 2007.

[4] G.-J. Ahn and R. Sandhu, "Role-based Authorization Constraints Specification," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 207 – 226, November 2000.

[5] J. Bacon, K. Moody, and W. Yao, "A Model of OASIS Role-based Access Control and its support for Active Security." *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, no. 4, pp. 492–540, 2002.

[6] T. Batista, A. Joolia, and G. Coulson, "Managing Dynamic Reconfiguration in Component-Based Systems," in *EWSA 2005, LNCS 3527*. Berlin Heidelberg: Springer-Verlag, pp. 1–17.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-wise Refinement," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197.

[8] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - A Component System for Pervasive Computing," in *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, March 14-17 2004, pp. 67–76.

[9] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca, "GEO-RBAC: A Spatially Aware RBAC," in *SACMAT '05: Proceedings of the Tenth ACM Symposium on Access control Models and Technologies*, 2005, pp. 29–37.

[10] R. Campbell, J. Al-Muhtadi, P. Naldurg, G. Sampemane, and M. D. Mickunas, "Towards Security and Privacy for Pervasive Computing," in *Lecture Notes in Computer Science Software Security - Theories and Systems*, vol. 2609. Springer, 2003, pp. 77–82.

[11] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective Middleware System for Mobile Applications," *IEEE Transactions on Software Engineering*, vol. 29, pp. 929–945, 2003.

[12] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi, "Specification and Implementation of Exceptions in Workflow Management Systems," *ACM Trans. Database Syst.*, vol. 24, no. 3, pp. 405–451, 1999.

[13] D. Cassou, B. Bertran, N. Loriant, and C. Consel, "A Generative Programming Approach to Developing Pervasive Computing Systems," *Proceedings of the Eigth International Conference on Generative Programming and Component Engineering, (GPCE'09)*, Oct. 2009.

[14] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance," in *Proceedings of the Sixteenth International Conference on Very Large Databases*, 1990, pp. 566–577.

[15] D. Chakraborty and H. Lei, "Pervasive Enablement of Business Processes," in *2nd International Conference on Pervasive Computing and Communications*. IEEE, January 2004.

[16] S. Chetan, A. Ranganathan, and R. Campbell, "Towards Fault Tolerant Pervasive Computing," *IEEE Technology and Society*, vol. 24, no. 1, pp. 38 – 44, Spring 2005.

[17] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[18] M. Corts and P. Mishra, "DCWPL: A Programming Language for Describing Collaborative Work," in *Proc. of CSCW'96*, November 1996, pp. 21 – 29.

[19] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd, "Securing Context-Aware Applications Using Environment Roles," in *SACMAT '01: Proceedings of the Sixth ACM Symposium on Access control Models and Technologies*, 2001, pp. 10–20.

[20] J. Crampton, "Specifying and Enforcing Constraints in Role-based Access Control," in *SACMAT '03: Proceedings of the Eighth ACM Symposium on Access control Models and Technologies*, 2003, pp. 43–50.

[21] K. Czarnecki and U. W. Eisenecker, *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, 2000.

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

[23] N. Davies, K. Cheverst, K. Mitchell, and A. Efrat, "Using and Determining Location in a Context-Sensitive Tour Guide," *IEEE Computer*, vol. 34, no. 8, pp. 35–41, August 2001.

[24] A. K. Dey, G. D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Hum.-Comput. Interact.*, vol. 16, no. 2, pp. 97–166, 2001.

[25] P. Dourish, "Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 2, pp. 109–155, 1998.

[26] M. Evered and S. Bögeholz, "A Case Study in Access Control Requirements for a Health Information System," in *ACSW Frontiers '04: Proceedings of the Second Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*, 2004, pp. 53–61.

[27] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-based Access Control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[28] 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.

[29] M. Fleck, M. Frid, T. Kindberg, E. O'Brien-Strain, R. Rajani, and M. Spasojevic, "From Informing to Remembering: Ubiquitous Systems in Interactive Museums," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 13–21, 2002.

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

[31] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[32] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Toward Distraction-Free Pervasive Computing," *IEEE Pervasive computing*, vol. 1, no. 2, pp. 22–31, April-June 2002.

[33] M. Ge and S. L. Osborn, "A Design for Parameterized Roles," in *DBSec*, 2004, pp. 251–264.

[34] L. Giuri and P. Iglio, "Role Templates for Content-based Access Control," in *RBAC '97: Proceedings of the Second ACM Workshop on Role Based Access Control*, 1997, pp. 153–159.

[35] J. B. Goodenough, " Exception Handling: Issues and Proposed Notations," *Communications of the ACM*, pp. 683–696, December 1975.

[36] 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.*, vol. 22, no. 4, pp. 421–486, 2004.

[37] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 943–958, October 2000.

[38] T. Halpin, *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufmann Publishers Inc., 2001.

[39] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Pervasive Computing," in *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, 2004, p. 77.

[40] A. K. Jones and B. H. Liskov, "A Language Extension for Expressing Constraints on Data Access," *Commun. ACM*, vol. 21, no. 5, pp. 358–367, 1978.

[41] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A Generalized Temporal Role-Based Access Control Model," *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE)*, vol. 17, no. 1, pp. 4–23, 2005.

[42] G. Judd and P. Steenkiste, "Providing Contextual Information to Pervasive Computing Applications," in *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, 2003, p. 133.

[43] C. Julien and G.-C. Roman, "EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications," *IEEE Trans. Softw. Eng.*, vol. 32, no. 5, pp. 281–298, 2006.

[44] 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)*, 2003, pp. 3–14.

[45] D. Kulkarni and A. Tripathi, “Application-level Recovery Mechanisms for Context-Aware Pervasive Computing,” *IEEE Symposium on Reliable Distributed Systems (SRDS’2008)*, pp. 13–22, 2008.

[46] ——, “Context-Aware Role-based Access Control in Pervasive Computing Systems,” *SACMAT’08 Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, pp. 113–122, 2008.

[47] D. Li and R. Muntz, “COCA: Collaborative Objects Coordination Architecture,” in *Proc. of CSCW’98*, 1998, pp. 179–188.

[48] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and Analysis of System Architecture Using Rapide,” *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 336–355, 1995.

[49] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, “Specifying Distributed Software Architectures,” in *Proceedings of the 5th European Software Engineering Conference.* London, UK: Springer-Verlag, 1995, pp. 137–153.

[50] T. W. Malone and K. Crowston, “The Interdisciplinary Study of Coordination,” *ACM Computing Surveys (CSUR)*, vol. 26, no. 1, pp. 87 – 119, March 1994.

[51] P. McDaniel, “On Context in Authorization Policy,” in *SACMAT ’03: Proceedings of the Eighth ACM Symposium on Access control Models and Technologies*, 2003, pp. 80–89.

[52] T. Moses, “OASIS eXtensible Access Control Markup Language (XACML) Version 2.0, OASIS Standard,” pp. 1–141, 1 February 2005.

[53] G. Neumann and M. Strembeck, “An Approach to Engineer and Enforce Context Constraints in an RBAC Environment,” in *SACMAT ’03: Proceedings of the Eighth ACM Symposium on Access control Models and Technologies*, 2003, pp. 65–79.

[54] G. J. Nutt, “The Evolution towards Flexible Workflow Systems,” *Distributed Systems Engineering*, vol. 3, pp. 276–294, 1996.

[55] M. Nyanchama and S. Osborn, “Modeling Mandatory Access Control in Role-based Security Systems,” in *Proceedings of the ninth annual IFIP TC11 WG11.3 working conference on Database security IX : status and prospects.* London, UK, UK: Chapman & Hall, Ltd., 1996, pp. 129–144.

[56] J. Park and R. Sandhu, "The UCONABC Usage Control Model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004.

[57] V. Poladian, S. J. Pedro, D. Garlan, B. Schmerl, and M. Shaw, "Task-Based Adaptation for Ubiquitous Computing," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems*, vol. 36, no. 3, May 2006.

[58] A. Popovici, A. Frei, and G. Alonso, "A Proactive Middleware Platform for Mobile Computing," in *M. Endler and D. Schmidt (Eds.): Middleware 2003, LNCS 2672.*

[59] 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: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*, 2005, pp. 7–16.

[60] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Platform for Active Spaces," *Mobile Computing and Communications Review*, vol. 6, no. 4, pp. 65–67, 2002.

[61] G. Sampemane, P. Naldurg, and R. H. Campbell, "Access Control for Active Spaces," in *Annual Computer Security Applications Conference (ACSAC2002)*, 2002.

[62] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, " Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, February 1996.

[63] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *Personal Communications, IEEE [see also IEEE Wireless Communications]*, vol. 8, no. 4, pp. 10–17, Aug 2001.

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

[65] Y. Smaragdakis, S. S. Huang, and D. Zook, "Program Generators and the Tools to make them," in *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* New York, NY, USA: ACM, 2004, pp. 92–100.

[66] T. Strang and C. Linnhoff-Popien, "A Context Modeling Survey," in *Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing*, September 2004.

[67] SUN, "Java Transaction API," http://java.sun.com/javaee/technologies/jta/index.jsp, November 2001.

[68] R. K. Thomas, "Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments," in *RBAC '97: Proceedings of the Second ACM Workshop on Role-based Access Control*, 1997, pp. 13–19.

[69] A. Tripathi, T. Ahmed, and R. Kumar, "Specification of Secure Distributed Collaboration Systems," in *IEEE International Symposium on Autonomous Distributed Systems (ISADS)*, April 2003, pp. 149–156.

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

[71] A. Tripathi and R. Miller, "Exception Handling in Agent-Oriented Systems," in *Advances in Exception Handling Techniques, LNCS 2022*. Berlin Heidelberg: Springer-Verlag, January 2001, pp. 128–146.

[72] A. R. Tripathi, D. Kulkarni, H. Talkad, M. Koka, S. Karanth, T. Ahmed, and I. Osipkov, "Autonomic Configuration and Recovery in a Mobile Agent-based Distributed Event Monitoring System," *Software - Practice & Experience*, vol. 37, no. 5, pp. 493–522, 2007.

[73] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, "Ontology Based Context Modeling and Reasoning Using OWL," in *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, 2004.

[74] M. Weiser, "The Computer for the Twenty-First Century," *Scientific American*, vol. 265, pp. 94–104, September 1991.

[75] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta, "Reconfigurable Context-Sensitive Middleware for Pervasive Computing," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 33–40, 2002.

[76] H. Zhand and S. Jarzabek, "An XVCL-based Approach to Software Product Line Development," in *Internal Conference on Software Engineering and Knowledge Engineering, SEKE'03*, 2003.

Appendix A

Activity Grammar

Here we present the grammar of the activity specifications in the EBNF format. The entities enclosed in curly braces ({}) represent zero or more terms, those enclosed in square brackets ([]) represent optional terms, and | represents a choice.

ActivityDef ⟶ **Activity** activityId {*RoleDef*} {*ObjectDef*} {*ReactionDef*} {*CollectionDef*}[*EventDispatchOrderDef*]

RoleDef ⟶ **Role** roleId [**AdmissionConstraints** *ConditionDef*] [**ValidationConstraints** *ConditionDef*]
{*ObjectDef*} {*OperationDef*} [*ExceptionInterfaceDef*]

OperationDef ⟶ **Operation** opId [*PreconditionDef*] {*ActionDef*} [*ContextGuardDef*]
[*OpExceptionHandlerDef*]

PreconditionDef ⟶ **Precondition** *ConditionDef*

ActionDef ⟶ **Action** (((objectId | collectionId | roleId.uci)
(**InvokeMethod** methodName | **InvokeSession** methodList) [*MethodParamDef*] [*AccessConstraintDef*]) |
NotifyEvent appDefinedEventId{*AttrName=AttrValue*} |
Attribute *AttrName* **Value Action** objectId (**InvokeMethod** methodName | **InvokeSession** methodName))
{*ActionExceptionHandlerDef*}

AccessConstraintDef ⟶ **AccessConstraint** (*AttrName* = *AttrValue*)

ActionExceptionHandlerDef ⟶ **OnException** *ActionExceptionDef*
{(*ActionDef* | **Action Throw** TransactionDisruptedException)}

ActionExceptionDef ⟶ (objectId.Exception | ObjectUnboundException | SessionAbortException)

ContextGuardDef ⟶ **ContextGuard When Event** *ContextEventDef* **GuardCondition** *ConditionDef*

OpExceptionHandlerDef ⟶ **OnException** *OperationExceptionDef ActionDef*

OperationExceptionDef ⟶ *ActionExceptionDef* | ContextInvalidationException

ExceptionInterfaceDef ⟶ **ExceptionInterface** {*ExceptionOpDef*}

ExceptionOpDef ⟶ **When Event** (roleId.opId.Exception | appDefinedEventId | *BindingEventDef*
ContextEventDef) **EnableFor** (Invoker | ANY | ALL) *OperationDef*

ObjectDef ⟶ **Object** objectId [**RDD** rdd] {**ImportEvent** *ContextEventDef FilterConditionDef* }
BindingReactionDef

BindingReactionDef ⟶ [**Reaction** reactionName
When Event (*ContextEventDef* | *FailureEventDef*)
PreconditionDef] (*BindingActionDef* | *UnbindingActionDef*)

BindingActionDef ⟶ **Bind** objectId **Action**
(**Direct** (*URL*) | *DiscoveryBindingDef* | **NewObject** (*codeBase*))
[**OnException**] BindingException (*BindingActionDef* | **Action NotifyEvent** appDefinedNotifyEventId)

DiscoveryBindingDef ⟶ **Discover**({*AttrName=AttrValue*}) [**OnException**] DiscoveryFailureException
BindingActionDef

UnbindingActionDef ⟶ **Unbind** thisObject

ReactionDef ⟶ **Reaction** reactionName **When Event** (*ContextEventDef* | *BindingEventDef*)
{*VarDef*} *PreconditionDef* **Action** (objectId | varId) **InvokeMethod** methodName

VarDef ⟶ **Var** varId **Select Set** collectionId **Condition Equal** *Value Value*

CollectionDef ⟶ **Collection** collectionId {**ImportEvent** *ContextEventDef*} *BindingReactionDef*

EventDispatchOrderDef ⟶ **Event_Dispatch_Order Event** *ContextEventDef* {**Dispatch_To** objectId}

MethodParamDef ⟶ **MethodParams** {**Method** methodName **Param** *ParamDef*}

ParamDef ⟶ (**Action** ((objectId | collectionId) **InvokeMethod** methodName) | *AttrValue*)

FilterConditionDef ⟶ **FilterCondition** members(roleId)

ConditionDef ⟶ *OperationEventConditionDef* | *ContextConditionDef*
ConditionDef BinOp ConditionDef | !*ConditionDef*

ContextConditionDef ⟶ Queries to context services | date *Relation Value* | time *Relation Value*

OperationEventConditionDef $\longrightarrow$ *EventCount Relation Count*

EventCount $\longrightarrow$ *#EventName*

EventName $\longrightarrow$ opId.start | opId.finish | appDefinedEventId

Relation $\longrightarrow > \mid < \mid == \mid \leq \mid \geq \mid \neq$

BinOp $\longrightarrow$ && | ||

Count$\longrightarrow$ 0 | 1 | ... | N

ContextEventDef $\longrightarrow$ Context Events | ActivityCreationEvent

FailureEventDef $\longrightarrow$ AccessRevocationEvent | ServiceConnectionFailedEvent

BindingEventDef $\longrightarrow$ objectId.ObjectBoundEvent | objectId.ObjectUnBoundEvent

AttrName $\longrightarrow$ *String*
AttrValue $\longrightarrow$ *String*
Value $\longrightarrow$ *String*
String $\longrightarrow$ XML CDATA

A.1 Operators

- #: Applies to an *eventId*. Returns the count of the *eventId*.
- thisUser: When used within role operations, role admission constraints, and role validation constraints, *thisUser* translates to the identity of the role member who is invoking the operation.
- thisObject: Can be used within an object's binding reactions. It translates to the identity of the object for whom the reaction is defined.
- members(*roleId*): Returns the list of the users who are current members of the role *roleId*
- member(*userId*, *roleId*): Returns true if user with identifier *userId* is a member of role *roleId*

Appendix B

Case Study Applications

Using the programming model we developed several context-aware applications in our testbed environment. These include context-aware music player, context-aware patient information system, context-aware distributed meeting, emulation of a museum environment, and context-aware exam session. Here we present the specifications of the first three applications For each application we present the application components and services that we developed.

B.1 Notation

In presenting the activity specifications we use the following notation. The programming model elements are represented in **boldface**. For specifying method invocations on objects in certain situations, we use *ObjectName.methodName()* pattern, instead of using *ObjectName* **InvokeMethod** *methodName* pattern. This simplification is done to aid the readability of the specifications.

B.2 Context-Aware Music Player

This application runs on a user's personal mobile computing device and supports the following context-based requirements. When the user runs the application, it starts streaming music to the audio player service on the user's device. When the user enters a room, the application 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, the application binds to the audio player service running on the user's device and continues streaming music to it.

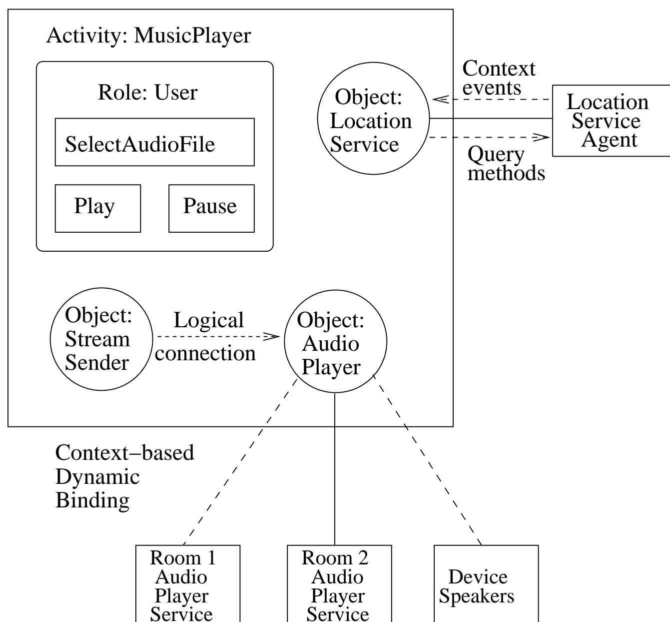


Figure B.1: Context-Aware music player architecture

In Figure B.1 we present the architecture of the music player activity. We define the *User* role and provide operations for selection, play, and pause. Three objects are defined in the activity. These include: *StreamSender*, *AudioPlayer*, and *LocationService*. The *StreamSender* object represents the audio source. The *AudioPlayer* object represents the audio player service in the room in which the user may be present. The *LocationService* binds to the location service agent.

B.2.1 Application Components and Services

We developed the component corresponding to the *StreamSender* object and an active space service corresponding to the *AudioPlayer* object. Both of these were implemented using the Java Media Framework (JMF).

The *StreamSender* component represents the music source. We designed this component to stream music to a single *target* at a time. The target's information is maintained by this component in the form of the target's IP address and port number. Below we present the interfaces of this component.

```
public class StreamSender {
    public void setAudio(String audioURL)
    public void addTarget(String target)
    public void removeTarget(String target)
    public void startStreaming()
```

```
    public String getAddress()
}
```

The *setAudio* interface is used to initialize the instantiated component with a specific audio file. The *addTarget* interface is used to add the specified <IP address, port number > pair as the target of the *StreamSender*. Similarly, *removeTarget* is used to remove the specified <IP address, port number > pair as the target of the *StreamSender*. If the passed IP address and the port number match those of the current target then the component removes that target. This causes the music streaming to stop on that particular target. The *startStreaming* interface causes the component to start streaming the music to the target. The *getAddress* interface returns the IP address of the host on which the *StreamSender* component is created. For the music player application, this is the host on which the activity is instantiated.

The *AudioPlayer* service represents the sink for playing the music. This service was designed to be able to receive streaming music from a single source. The interfaces implemented for this service are presented below.

```
public class AudioPlayer {
    public void setSender(String source)
    public String getAddress()
    public void removeSender(String source)
}
```

The *setSender* interface sets the address of the source within the *AudioPlayer* service. The service's implementation of this interface parse an IP address, and port number from this passed parameter. The *getAddress* interface returns the IP address and port number of the service. The *removeSender* interface removes the source whose address is specified as a parameter.

B.2.2 Specification

Below we present the specification of the music player activity. [1] In the activity specification we define the *StreamSender* object (lines 10-15) to represent the source of the audio stream and bind it with the corresponding application component that is instantiated in the activity.

```
1.Activity MusicPlayer {
2.  Role User {
3.      Operation SetAudio Action StreamSender InvokeSession setAudio
4.  } // end of User Role
5.  Object LocationService {
6.      ImportEvent UserArrivalEvent FilterCondition members(User)
7.      Reaction BindLocationService
8.          When ActivityCreationEvent Action Bind Direct (//LocationServiceAgentURL)
```

[1] While presenting activity specifications in pseudo notation we use C++ style comments (//) to write a comment line. In the XML activity specifications no commenting constructs are currently supported.

```
9.  }
10. Object StreamSender {
11.     Reaction R1 {
12.         When Event ActivityCreationEvent
13.         Action Bind StreamSender New (//codeBase/StreamSender)
14.     }
15. } // end of StreamSender object
16. Object CurrentRoom RDD (//RoomRDD.xml) {
17.     Reaction BindToCurrentRoom {
18.         When Event UserArrivalEvent
19.         Action Bind Discover
20.                 (LOCATION=LocationService.getLocation (UserArrivalEvent.getUserName()))
21.     }
22.     Reaction UnBindFromCurrentRoom {
23.         When Event UserDepartureEvent Action Unbind thisObject
24.     }
25. } // end of CurrentRoom object
26. Object AudioPlayer RDD (//AudioPlayerRDD.xml) {
27.     Reaction BindToDeviceSpeakers {
28.         When Event ActivityCreationEvent Action Bind Direct (//DeviceAudioPlayerURL)
29.     }
30.     Reaction BindToRoomSpeakers {
31.         When Event UserArrivalEvent
32.         Precondition CurrentRoom.presentUserCount() == 1
33.         Action Bind Discover
34.                 (LOCATION=LocationService.getLocation(UserArrivalEvent.getUserName()))
35.     }
36.     Reaction UnBindOnUserDeparture {
37.         When Event UserDepartureEvent
38.             Precondition AudioPlayer.isBound()
39.             Acion StreamSender.removeTarget(AudioPlayer.getAddress())
40.             Acion AudioPlayer.removeSender(StreamSender.getAddress())
41.             Action Bind Direct (//DeviceAudioPlayerURL)
42.     }
43.     Reaction UnBindWhenOthersEnter {
44.         When Event RoomStatusChangeEvent
45.         Precondition CurrentRoom.presentUserCount() > 1
46.         Acion StreamSender.removeTarget(AudioPlayer.getAddress())
47.         Acion AudioPlayer.removeSender(StreamSender.getAddress())
48.         Action Bind Direct (//DeviceAudioPlayerURL)
49.     }
50. } // end of AudioPlayer object
51. Reaction ConnectStreamSenderToAudioPlayer {
52.     When Event AudioPlayer.ObjectBoundEvent
53.     Precondition AudioPlayer.isBound() && StreamSender.isBound()
54.     Action AudioPlayer.setSender(StreamSender.getAddress())
```

```
55.     Action StreamSender.addTarget(AudioPlayer.getAddress())
56.     Action StreamSender InvokeMethod startStreaming
57. }
58. Event_Dispatch_Order {
59.     Event UserArrivalEvent
60.         Dispatch_To CurrentRoom
61.         Dispatch_To AudioPlayer
62. }
63.} // end of MusicPlayer activity
```

We define the *CurrentRoom* object (lines 16-25) to refer to the room's agent where the user is present. We define the *AudioPlayer* object (lines 26-50) to refer to either the audio player service on the user's device or the audio player service in the room where the user is present. Because this is a single user activity, we can define all the objects either in the activity's objectspace or in the role's objectspace. Here we choose to define the objects in the activity's objectspace.

The reactions *BindToCurrentRoom*, and *UnBindFromCurrentRoom* are defined for the *CurrentRoom* object. Reaction *BindToCurrentRoom* is triggered by the *UserArrivalEvent* and causes the object to bind to the context agent of the room in which the user is currently present. Reaction *UnBindFromCurrentRoom* is triggered by the *UserDepartureEvent* and causes unbinding of the object. In our specification model *thisObject* is a special variable which refers to the object manager of the object in whose definition the variable is used.

The reactions *BindToDeviceSpeakers*, *BindToRoomSpeakers*, *UnBindOnUserDeparture*, and *UnBindWhenOthersEnter* are defined for the *AudioPlayer* object. Reaction *BindToDeviceSpeakers* is triggered by the *ActivityCreactionEvent* (lines 27-29). This event is generated by the middleware when an activity is instantiated. Upon triggering this reaction causes the object to bind to the audio player service running on the user's device.

Reaction *BindToRoomSpeakers* is triggered by the *UserArrivalEvent*. It binds the object to the audio player service in the room, if only this user and no other person is present in the room. For this it queries the *CurrentRoom* object. We define the *event dispatch order* (lines 58-62) for handling of the *UserArrivalEvent*. This ensures that the *CurrentRoom* object would be bound before binding the *AudioPlayer* object, thus avoiding the problem discussed in Section 4.7. Successful binding of the *AudioPlayer* object leads to the generation of the *AudioPlayer.ObjectBoundEvent*. This triggers the reaction *ConnectStreamSenderToAudioPlayer* (lines 51-57). It connects the *AudioPlayer* object with the *StreamSender* object, and invokes the *startStreaming* method on the *StreamSender* object. This causes music streaming to begin on the audio player service to which the *AudioPlayer* object is currently bound.

Reaction *UnBindOnUserDeparture* defined for the *AudioPlayer* object is triggered by the *UserDepartureEvent* (lines 36-42). When the user leaves a room, this reaction first stops music streaming to the audio player service of the room. This is done by removing the source from the

AudioPlayer service. It then binds the object to the audio player service running on the user's device. This leads to the generation of the *AudioPlayer.ObjectBoundEvent*, which causes the *ConnectStreamSenderToAudioPlayer* reaction to trigger and start streaming music to the user's device.

We define the reaction *UnBindWhenOthersEnter* for the *AudioPlayer* object that is triggered by the *RoomStatusChangeEvent* (lines 43-49). This event is generated by the room agent whenever some user enters the room. In the precondition of this reaction we check whether the number of users in the room are more than one. If so, the reaction binds the object to the music player service on the user's device.

B.3 Context-Aware Patient Information System

The requirements for the patient information system are presented in Chapter 2. In Figure B.2 we present the architecture of this activity. For this application we developed the following component.

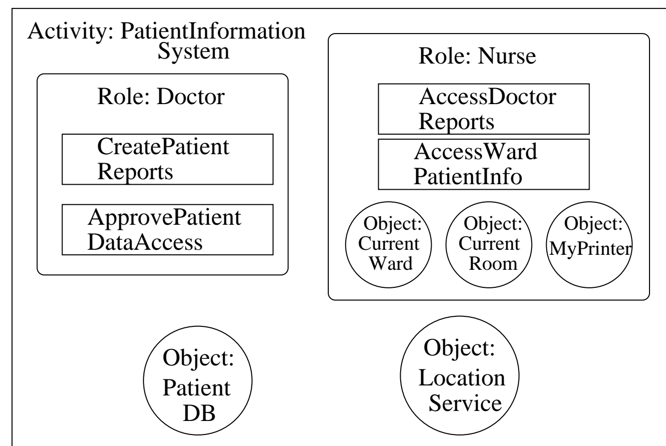


Figure B.2: Context-Aware patient information system architecture

B.3.1 Application Component and Services

The *PatientDB* component was developed to represent the database service that maintains patient information and doctor reports. This information is maintained in a SQL Database. Below we present the interfaces of this component.

```
public class PatientDB {
    public void createReports(String report)
    public String accessDoctorReports()
```

```
        public String accessPatientInformation(String queryString)
    }
```

The *createReports* interface is used to create reports. The *accessDoctorReports* interface is used to retrieve doctor reports. The *accessPatientInformation* interface is used to access the patient information. This interface takes a *queryString* as input. The service's implementation of this interface uses this query string to construct a SQL query which is used to query the database.

B.3.2 Specification

Below we present the specification of the patient information system. In the activity we define the *PatientDB* object and bind it to the patient database service that we run at a well-known URL (lines 2-5).

```
1.Activity PatientInformationSystem {
2.  Object PatientDB {
3.      Reaction
4.          When Event ActivityCreationEvent Bind Direct (//PatientDBServiceURL)
5.  }
6.  Object LocationService {
7.      Reaction
8.          ImportEvent UserArrivalEvent FilterCondition members(Nurse)
9.          When Event ActivityCreationEvent Bind Direct (//LocationServiceAgentURL)
10. }
11. Role Doctor {
12.     Operation CreatePatientReports {
13.         Action PatientDB InvokeSession createReports
14.     }
15.     Operation ApprovePatientDataAccess {
16.         Action NotifyEvent AccessApprovalEvent
17.             Attribute nurseID Value Action Nurse InvokeSession selectMember
18.     }
19. }
20. Role Nurse {
21.     Object CurrentWard RDD (//WardRDD.xml) {
22.         Reaction BindCurrentWard {
23.             When Event UserArrivalEvent
24.             Bind Discover (LOCATION=LocationService.getLocation(
25.                 UserArrivalEvent.getUserName()))
26.         }
27.     }
28.     Object CurrentRoom { // Context-based binding }
29.     Object MyPrinter { // Context-based binding }
```

```
30.     Operation AccessDoctorReports {
31.         Precondition
32.             CurrentWard.isPresent(thisUser) && CurrentWard.isPresent(members(Doctor))
33.             && #Doctor.AccessApprovalEvent(nurseID=thisUser) > 0
34.         Action PatientDB InvokeMethod accessDoctorReport
35.     }
36.     Operation AccessWardPatientInfo {
37.         Action PatientDB InvokeSession accessPatientInformation
38.             AccessConstraint (WardID = CurrentWard.getWardID())
39.     }
40.     Operation Print {
41.         Action MyPrinter InvokeSession print
42.     }
43. }
44.}
```

The *Doctor* role is provided with *CreatePatientReports* operation (lines 12-14) through which a role member can create patient reports. The *ApprovePatientDataAccess* operation (lines 15-18) allows a *Doctor* role member to grant permission to a particular *Nurse* role member to access doctor's reports.

The *Nurse* role is provided with *AccessDoctorReports* operation (lines 30-35). A nurse is allowed to access doctor's reports only if any doctor is present in the ward where the nurse is currently present, and if the doctor has given an approval to that nurse. This condition is specified as the precondition of the *AccessDoctorReports* operation. The operation *AccessWardPatientInfo* (lines 36-39) allows a nurse to access the information about patients who are admitted to the ward where that nurse is currently present. The *AccessConstraint* mechanism is used for this purpose. The object manager for the *PatientDB* object constructs a query string from the specified access constraint and passes it to the database service through the *accessPatientInformation* interface. The service returns only those patient's records that match the nurse's current ward location.

B.4 Context-Aware Distributed Meeting

This is a distributed meeting application spanning multiple rooms [70]. The application supports a chairperson, and number of participants who may be present in different rooms. In this meeting a participant is allowed to perform a *classified presentation* as long as he or she is *co-located* with the chairperson in the same room. The presentation of the classified information by the participant is displayed only on the projector in the room where the participant is present together with the chairperson. If the chairperson leaves the room, or if the participant leaves the room, the classified presentation is automatically ended. The presentation made by the

chairperson is *non-classified* and is displayed on the projectors of every meeting room. In case a particular room becomes empty during the course of the meeting possibly because all the people in that room have left the room, the presentation in that room is automatically ended. In Figure B.3 we present the architecture of this activity.

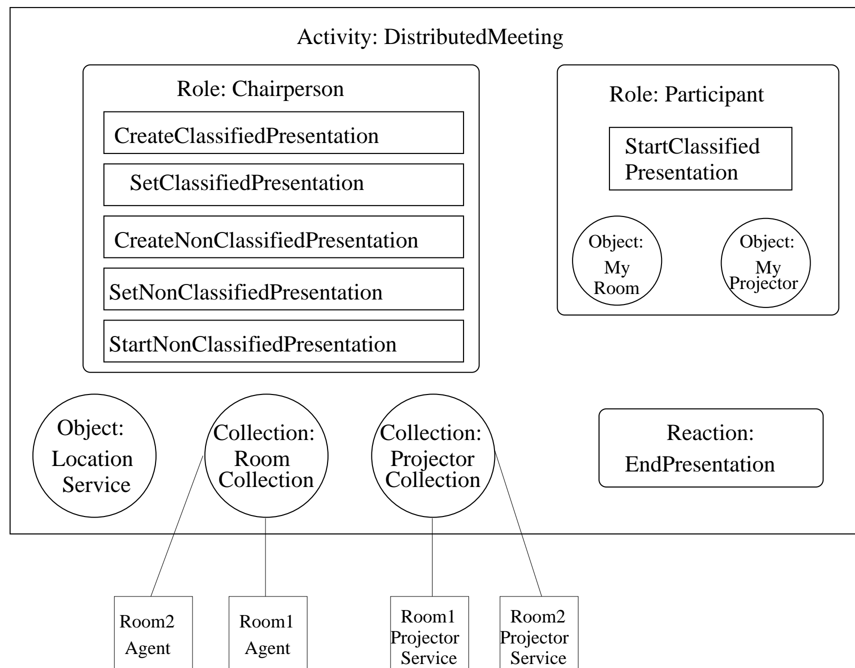


Figure B.3: Context-Aware distributed meeting architecture

B.4.1 Application Components and Services

We define a generic *Presentation* component which can be used for instantiating a *Classified-Presentation* and a *NonClassifiedPresentation*. The interfaces of this component are presented below.

```
public class Presentation {
    public void setPresentation(String URL)
    public void setNumberOfSlides(int totalSlides)
    public String getCurrentSlide()
    public String getNextSlide()
    public String getPreviousSlide()
    public String getLastSlide()
}
```

The interfaces *setPresentation* and *setNumberOfSlides* are used to initialize the presentation component with the particular presentation. The interfaces *getCurrentSlide*, *getNextSlide*, *getPreviousSlide*, and *getLastSlide* are used to obtain the current, next, previous, and the last slide of a presentation, respectively.

We define the *Projector* service to represent a projector in a room. This service provides a blank Java Swing Panel on which a slide may be displayed. This service supports the following interfaces.

```
public class Projector {
    public void display(String slide)
    public void next(String slide)
    public void previous(String slide)
}
```

The interfaces *display*, *next*, and *previous* are used to display respectively, a specific slide, next slide, and previous slide. The next and previous slides are relative to the currently displayed slide.

B.4.2 Specification

Below we present the specification of the distributed meeting application which is designed for meetings spanning two rooms.

```
1.Activity DistributedMeeting {
2.  Object LocationService {
3.      ImportEvent UserArrivalEvent FilterCondition members(Participant)
4.      Reaction BindLocationService
5.          When ActivityCreationEvent Action Bind Direct (//LocationServiceURL)
6.  }
7.  Collection RoomCollection {
8.      ImportEvent RoomStatusChangeEvent
9.      Reaction BindRoomAgents {
10.         When ActivityCreationEvent
11.             Action Bind Direct (//Room1AgentURN)
12.             Action Bind Direct (//Room2AgentURN) }
13. }
14. Collection ProjectorCollection {
15.     Reaction BindProjectors {
16.         When ActivityCreationEvent
17.             Action Bind Direct (//Projector1URL)
18.             Action Bind Direct (//Projector2URL) }
19. }
20. Role Chairperson {
21.     Operation CreateClassifiedPresentation Action ClassifiedPresentation
```

```
22.         NewObject (//Presentation/codeBase)
23.     Operation SetClassifiedPresentation Action ClassifiedPresentation
24.         InvokeSession setPresentation setNumberOfSlides
25.     Operation CreateNonClassifiedPresentation Action NonClassifiedPresentation
26.         NewObject (//Presentation/codeBase)
27.     Operation SetNonClassifiedPresentation Action NonClassifiedPresentation
28.         InvokeSession setPresentation setNumberOfSlides
29.     Operation StartNonClassifiedPresentation {
30.         Action ProjectorCollection InvokeSession display next previous
31.             MethodParams {
32.                 Method display Param NonClassifiedPresentation.getCurrentSlide()
33.                 Method next Param NonClassifiedPresentation.getNextSlide()
34.                 Method previous Param NonClassifiedPresentation.getPreviousSlide()
35.             }
36. } // end of Chairperson Role
37. Role Participant {
38.     Object MyRoom RDD (//RoomRDD.xml) {
39.         Reaction {
40.             When Event UserArrivalEvent
41.             Action Bind Discover
42.                 (LOCATION=LocationService.getLocation(UserArrivalEvent.getUserName()))
43.         }
44.     }
45.     Object MyProjector RDD (//ProjectorRDD.xml) {
46.         Reaction {
47.             When Event UserArrivalEvent
48.             Action Bind Discover
49.                 (LOCATION=LocationService.getLocation(UserArrivalEvent.getUserName())) }
50.         }
51.     }
52.     Operation StartClassifiedPresentation {
53.         Precondition MyRoom.isPresent(thisUser) && MyRoom.isPresent(members(Chairperson))
54.         Action MyProjector InvokeSession display next previous
55.             MethodParams {
56.                 Method display Param ClassifiedPresentation.getCurrentSlide()
57.                 Method next Param ClassifiedPresentation.getNextSlide()
58.                 Method previous Param ClassifiedPresentation.getPreviousSlide()
59.             }
60.     }
61.     ContextGuard {
62.         When Event RoomCollection.RoomStatusChangeEvent
63.         GuardCondition MyRoom.isPresent(thisUser) && MyRoom.isPresent(members(Chairperson))
64.     }
65.     OnException ContextInvalidationException {
66.         Action MyProjector InvokeMethod display
67.             MethodParams Method display Param ClassifiedPresentation.getLastSlide()
```

```
68.     }
69. } // end of Participant Role
70. Reaction EndPresentation {
71.     When Event RoomCollection.RoomStatusChangeEvent
72.         Var Room {
73.             Select Set RoomCollection Condition Equal RoomStatusChangeEvent.getLoc()
74.                             RoomCollection.getLoc()
75.         }
76.         Var Projector {
77.             Select Set ProjectorCollection Condition Equal RoomStatusChangeEvent.getLoc()
78.                             ProjectorCollection.getLoc()
79.         }
80.     Precondition Room.presentUserCount() == 0
81.     Action Projector InvokeMethod display
82.         MethodParams Method display Param NonClassifiedPresentation.getLastSlide()
83. } // end of the Reaction
84.} // end of DistributedMeeting activity
```

We define the *RoomCollection* (lines 7-13) to represent the agents of the various rooms in which a meeting is being organized. We define the *ProjectorCollection* (lines 14-19) to represent the projector services in the various rooms. We use the *Collection* primitive defined in our programming framework for defining these collections. A collection is similar to an object, the only difference being that multiple services can be bound to a collection. In the middleware, a collection is managed by an *ObjectCollectionManager*. A collection manager supports interfaces for executing a particular method on all the services that are bound to a collection, and for selecting a particular service from the collection based on a *selector* predicate. For example, a service's identifier can be used as a selector predicate.

We define the *Chairperson* role (lines 20-36) and provide it the following operations *CreateClassifiedPresentation*, *SetClassifiedPresentation*, *CreateNonClassifiedPresentation*, *Set-NonClassifiedPresentation*, *StartNonClassifiedPresentation*. The *CreateClassifiedPresentation*, and *CreateNonClassifiedPresentation* operations allow the chairperson to create the classified and the non-classified presentations, respectively. The codebase for the *Presentation* component is used for creating both these presentations. The *SetClassifiedPresentation*, and *SetNonClassifiedPresentation* operations can be used by the chairperson to set the classified and the non-classified presentations, respectively. Through the *StartNonClassifiedPresentation* operation the chairperson can initiate the non-classified presentation. This operation allows the chairperson to execute the following methods, *display*, *next*, and *previous*, on the *ProjectorCollection*. The parameter values for these methods are obtained by invoking methods *getCurrentSlide*, *getNextSlide*, and *getPreviousSlide* on the *NonClassifiedPresentation* object (lines 31-34).

We define the *Participant* role (lines 37-69) and define the *MyRoom* object and the

MyProjector object in it. The *MyRoom* object refers to the agent in the room where a particular participant is present, and the *MyProjector* object refers to the projector in that room. The operation *StartClassifiedPresentation* (lines 52-60) allows a participant to start the classified presentation. The precondition for this operation ensures that a participant will be able to execute this operation only if that participant is co-located with the chairperson in the same meeting room. If the precondition is true, the operation allows the participant to execute the following methods, *display*, *next*, and *previous*, on the *MyProjector* object. The parameter values for these methods are obtained from the *ClassifiedPresentation* object (lines 55-59). We define a context guard and *ContextInvalidationException* handler for this operation. The context guard is triggered by the *RoomStatusChangeEvent* which may be generated by any of room agents. If the context guard fails to hold, the participant's session is terminated and the *ContextInvalidationException* is raised. As part of handling this exception the presentation in that room is closed by displaying the last slide of the presentation on the projector service.

We define the *EndPresentation* reaction (lines 70-83). The purpose of this reaction is to end the presentation in a room if the room is empty. The execution of this reaction is triggered by the *RoomStatusChangeEvent*. Once triggered, the reaction has to do the following. The reaction has to first find out whether the room corresponding to which the event was generated is empty. If it is empty, then the reaction has to end the presentation on the projector in that room. This requirement is programmed as follows. First, we create two temporary variables *Room*, and *Projector*, by selecting the appropriate room agent and the projector from the *RoomCollection* and the *ProjectorCollection*, respectively. We use the *getLoc* as the selector function, which returns the location attribute of a service. The visibility of these temporary variables is restricted to the scope of the reaction. As part of the reaction's precondition we check whether the selected room is empty. This is done by invoking the *presentUserCount* method on the *Room* variable. If the precondition is true, the non-classified presentation in that room is ended by displaying the last slide of the presentation on that room's projector (whose reference is available in the variable *Projector*).