Generative Programming Approach for Building Pervasive Computing Applications

Devdatta Kulkarni and Anand Tripathi*
Department of Computer Science
University of Minnesota Twin Cities
MN 55455, USA
Email: (dkulk,tripathi)@cs.umn.edu

Abstract

In this paper we present a generative programming approach for building context-aware applications. In this approach, a context-aware application is programmed using high-level specification constructs provided in our programming framework. The runtime environment of the application is generated from this specification by a middleware. We demonstrate the utility of this approach by presenting an example case-study.

1 Introduction

Developing context-aware applications is a challenging task because of several reasons. Such applications require dynamic integration of application components, resources, and services based on the context. Applications may span across multiple physical spaces. Application requirements may evolve over time. Modifying an application's behavior may require re-programming the entire application.

To address above challenges we have developed a programming framework to support rapid construction of context-aware applications from their high-level specifications. Our approach for building context-aware applications follows the generative programming paradigm [3]. The central idea in our approach is to construct the runtime environment of a context-aware application from its high-level design specification. The runtime environment of the application is generated by dynamically integrating policies that are derived from the application's specification with a set of middleware components and environmental services.

There are several advantages of this approach. First, the task of developing context-aware applications is simplified. Development efforts are limited to developing the application components and developing the specification of various context-aware requirements for the application. Developers need not worry about integrating the application with environmental services, such as context services and resource discovery services. The middleware automatically performs these tasks as part of generating the application's runtime environment from its specification. Second, this approach allows rapid construction of a context-aware application in which the design requirements including any context-based requirements can be modified easily. Third, it is possible to perform various kinds of static analyses [1] of the application's specification to ensure that the application requirements are consistent and do not violate any security requirements.

*This work was supported by NSF grant 0411961.

2 Programming Framework Overview

Generative programming paradigm corresponds to automated generation of a family of applications belonging to a particular domain from high-level specification of the application's requirements [3]. Our approach for building context-aware applications follows this generative programming paradigm as shown in Figure 1. The conceptual foundations of our programming framework for context-aware applications has its roots in the specification based programming model for distributed collaborative applications that we had developed earlier [6]. In this programming model, an *activity* abstraction defines a shared application workspace containing *shared objects* for a group of collaborating *role members* to perform application related tasks. An activity defines a namespace of such objects which is accessible to all the roles. Additionally, each role may also define a namespace of objects, which is maintained separately for each member of that role. An object in a namespace may be bound to different resources under different context conditions. Mechanisms are provided to specify conditions under which such bindings should change. These mechanisms support integration of context information in

performing resource discovery functions. Furthermore, at the time of binding a name to a resource, certain specified initialization actions may be performed on the resource.

Roles are provided with *role operations* through which role members may perform application tasks by invoking methods on objects in the activity's or a role's namespace. User's privileges to execute role operations are based on application's internal context conditions such as task execution histories within the activity, user's membership in roles, and also on external context conditions, such as user's presence in a particular place or co-location of a user with other users. Such dynamic access control policies for role operation executions are specified through *preconditions* associated with the role operations.

A middleware provides generic manager components for managing activities, roles, and objects. As part of the application generation process, policies are derived from the application's specification and integrated with the generic managers [6]. These policies include access control policies for object managers, event subscription/notification policies for role managers and object managers, role management policies, such as role admission policies and role activation policies, for role managers, and binding policies for the activity manager and role managers. Application specific managers for activity, roles, and objects are derived from the corresponding generic managers by integrating these policies. Every role member is provided a *User Coordination Interface* (UCI) component through which he/she can execute application tasks.

3 Case Study: Developing a Context-Aware Application

We illustrate here programming of a context-aware application using this framework. We consider a music player application for a mobile user. A mobile user plays music on his/her personal computing device. The user may like to listen the music either on his/her personal headphones or on the public speakers in the room. We consider the following context-aware requirement for this application. When the user enters a room the application automatically discovers and binds to the room's public speakers. We consider implementation of two alternate designs for this application.
Design 1: Automatically start streaming the music to the room's public speakers if no one else is present in the room.
Design 2: User should explicitly initiate streaming of music to the public speakers. This action should be allowed only if no one else is present in the room.

In order to program these requirements, the application's design needs to contain the following: First, objects need to be defined to represent user's headphones and the public speakers in the room. The binding of the headphone will be permanent while the binding of the public speakers will change as the user moves from one room to another room. This binding action will have to *discover* the appropriate public speakers in the room in which the user is present. Second, the application needs to be notified of context events, such as arrival of a user in a room. Similarly, the application should be able to query the state of the environment, such as whether a particular user is present in the room, and the number of users present in the room. Correspondingly, objects need to be defined to represent the environmental context agents, for example location service agent that monitors user location, and agents representing physical space abstractions such as the rooms. The binding of location service will be permanent while the binding of the agent representing the user's current location will change as the user enters or leaves a room. Finally, the application needs to provide operations through which a user may initiate music streaming on headphones and public speakers.

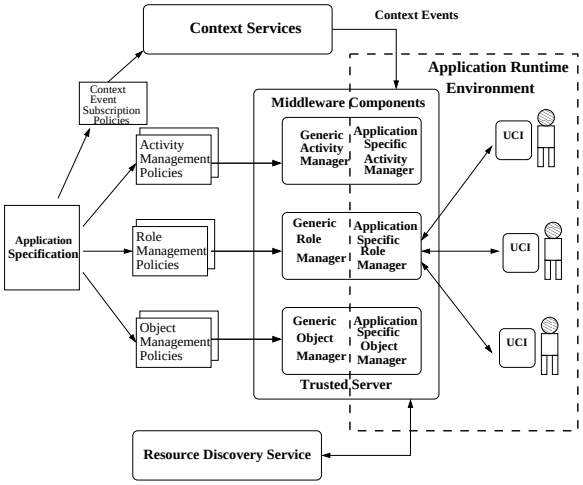

Figure 1. Context-Aware Application Generation Process

Below we show the specification of such a music player application modeled as an activity[1].

```
1.Activity MusicPlayer
2.    Object UserLocationService
3.        Import_Event UserArrivalEvent
4.    Bind UserLocationService Direct (//UserLocationAgentURN)
5.    Role User
6.        Object CurrentRoom RDD (//RoomRDD.xml)
7.        Bind CurrentRoom When UserArrivalEvent
8.                Discover (LOCATION=
9.                    UserArrivalEvent.getLocation(thisUser))
10.       Operation CreateAudioPlayer
11.           Action AudioPlayer = New (//codeBase/AudioPlayer)
12.       Operation InitializeAudioPlayer
13.           Action AudioPlayer.setMediaLocation(//MediaURL)
```

```
14.   Object HeadPhone
15.   Bind HeadPhone Direct (//LocalAudioReceiverURL)
16.   Object PublicSpeaker RDD (//SpeakerRDD.xml)
17.   Bind PublicSpeaker When UserArrivalEvent
18.     Discover (LOCATION=
19.       UserArrivalEvent.getLocation(thisUser))
20.     Init_Action
21.       Precondition CurrentRoom.isPresent(thisUser)
22.       && CurrentRoom.presentUserCount() == 1
23.       Action
24.       PublicSpeaker.setSender(AudioPlayer.getAddress())
25.       AudioPlayer.addTarget(PublicSpeaker.getAddress())
26.   Operation PlayMusicOnHeadPhones
27.     Action HeadPhone.setSender(AudioPlayer.getAddress())
28.     AudioPlayer.addTarget(HeadPhone.getAddress())
```

Environmental Context Agents: In the activity specification, an object *UserLocationService* is defined in the activity's namespace (line 2). The *UserArrivalEvent* is specified to be imported from this object (line 3). This object is bound permanently with the user location tracking agent (*UserLocationAgent*) (line 4). In our programming framework, such agents are programmed using an agent based distributed event monitoring framework that we have developed [7]. The agents representing physical spaces are registered with the discovery service to allow them to be discovered by applications.

A *User* role is defined to represent the application user (line 5). This role defines an object *CurrentRoom* in its namespace to represent the agent of the room in which the user is currently present (line 6). We define two role operations, *CreateAudioPlayer* (lines 10-11) and *InitializeAudioPlayer* (lines 12-13), through which the user may create the audio player and initialize it with the required music file.

Context-based Resource Binding: Two objects, *HeadPhone* and *PublicSpeaker*, are defined in the *User* role's namespace. The *HeadPhone* object is *directly* bound with the audio receiver on the user's device (line 15). The *PublicSpeaker* object represents the public speakers in the room. The binding of this object is shown in lines 17-19. The binding action is triggered by a *UserArrivalEvent* that is generated whenever the user enters a room. Binding is performed by *discovering* the audio receiver in the room in which the user has just entered.

Discovering a resource in the environment involves the following. The object that would be bound through discovery needs to be associated with a description which would be used in searching for the required resource. For the *PublicSpeaker* object, such a description is given through SpeakerRDD.xml (line 16). The description specifies the attributes and the interfaces of the audio receiver that would be bound with this object. In this description some of the attributes may be filled based on the context information. For the *PublicSpeaker* object, the attribute LOCATION is filled based on the user's current location. The current location of the user is extracted from the *UserArrivalEvent* by invoking *getLocation* method on the event. A special variable *thisUser* defined in the programming framework is passed as a parameter to this method. This variable represents the role member for whom the binding is triggered. This location context information is plugged in the SpeakerRDD.xml and is used by the discovery service to find a matching resource. *PublicSpeaker* object is bound to the discovered resource if the discovery is successful. Binding of the *CurrentRoom* object is similar (lines 7-9). Role member namespace is maintained separately for each role member. Hence this same specification can be used by multiple users.

Requirement 1: Automatic music streaming: For this requirement we want that once the *PublicSpeaker* object has been bound and if there is no one else present in the room, the application should start streaming the music to the public speakers. This is programmed as part of the initialization action of the *PublicSpeaker* object (lines 20-25). The initialization action has two parts: a precondition specification and a list of action specification. The precondition checks whether the user is present in the *CurrentRoom* and whether the number of users present in the *CurrentRoom* is equal to one. This is achieved by executing the query methods (*isPresent* and *presentUserCount*) on the *CurrentRoom* object. The actions are performed only if the precondition is true. As part of actions, first the *PublicSpeaker* is initialized with the audio player's session address (line 24) and then it is added as a target of the *AudioPlayer* (line 25).

When the user moves from one room to another room, we want to stop streaming the music to public speakers of the previous room. This is programmed as part of the *PublicSpeaker's setSender* interface. This interface will invoke the *AudioPlayer's removeTarget* method by passing to it the previous address associated with the *PublicSpeaker*. This is not required for the *HeadPhone* object.

Requirement 2: User initiated music streaming: For this requirement we want the user to initiate streaming of the music on the *PublicSpeaker*. This can be programmed by providing a *PlayMusicOnPublicSpeakers* operation to the *User* role. The precondition and action parts of this operation will be exactly similar to the corresponding parts of the *Init_Action* in the binding specification of *PublicSpeaker* object. We do not show that specification here. The operation *PlayMusicOnHeadPhones* allows streaming the music to the user's *HeadPhone*.

We observe that there is a potential race condition between the binding of *CurrentRoom* (lines 7-9) and *PublicSpeaker* (lines 17-19) objects. Both the bindings are trig-

[1]We have developed an XML schema for activity specifications. Here we use a pseudo notation for writing activity specifications. In this notation, terms in **boldface** represent XML tags. Complete specification is available at http://www.cs.umn.edu/Ajanta/MusicPlayer/MusicPlayer.txt

gered by the *UserArrivalEvent*. It may happen that the *PublicSpeaker* binding is triggered *before* the binding of *CurrentRoom*. In that case the precondition (lines 21-22) evaluation in the binding fails because *CurrentRoom* is unbound. It is possible to program the application such that the set of rooms in which the user may move are pre-bound as a *collection* object in the activity. The precondition evaluation would then be modified to first select from the collection the room in which the user is present and then query the state information from the selected room. Due to space limitations we do not show that specification here.

4 Related Work

Other research groups that have followed a similar approach include Gaia [5], and RCSM [8]. In Gaia, active spaces are programmed using the Olympus programming environment which is implemented as a Gaia middleware service. The RCSM system [8] provides a context-aware interface definition language for programming context-awareness in applications' components. Gaia shields context-awareness behind its high-level operators whereas RCSM restricts context usage only to component interface definitions. In our programming framework, context-usage is much more explicit. Constructs for resource binding and task execution explicitly support integration of context information. This allows us to provide finer control over context-aware requirements of the application. Several adaptive middlewares for context-aware applications [2, 4] have been developed by other researchers. The goal of both PCOM [2] and Chisel [4] is to automatically support adaptation of a context-aware application at runtime. PCOM performs such adaptations through discovery and integration of components satisfying a particular contract at runtime. Application adaptations are guided by component contracts. Our approach is similar to this but differs in the way intra-component requirements are specified. In our approach, policies *derived* from an application's specification capture the intra-component requirements. In a sense, policies are latent in the application's specification. Such policies are integrated with application components to generate a particular runtime environment. These policies are similar to the PCOM's intra-component contracts. Chisel focuses on adaptation of the non-functional requirements of an application, specified as an application's metatypes, driven by a high-level policy language. For certain applications, it may not be possible to satisfy the adaptation requirements by performing such a functional/non-functional separation. For example, in the music player application, the context-triggered binding of an application's namespace is a functional requirement but would have to be considered as non-functional in order to implement the application in the Chisel framework. Our approach does not make such a distinction. On the contrary it helps in modeling the complete design of a context-aware application under different context conditions.

5 Discussion

In our approach, the entire operational configuration of a context-aware application, under different contextual conditions, is laid out in the form of its specification. The generated runtime environment supports dynamic integration of environmental resources and context-dependent task executions. One limitation of this approach is that the user interactions are highly structured and cannot be extended in an ad-hoc manner at runtime. The advantage is that it provides a concise representation of the application's context-aware behavior. Such a high-level specification is also amenable to static analyses [1] for ensuring properties of consistency, correctness, and secure information flow for the application.

References

[1] T. Ahmed and A. R. Tripathi. Static Verification of Security Requirements in Role Based CSCW Systems. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 196–203, New York, June 2003. ACM.

[2] C. Becker, M. Handte, G. Schiele, and K. Rothermel. PCOM - A Component System for Pervasive Computing. In *Proc. Second IEEE International Conference on Pervasive Computing and Communications*, pages 67–76, March 14-17 2004.

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

[4] J. Keeney and V. Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–14, Washington, DC, USA, 2003. IEEE Computer Society.

[5] 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*, pages 7–16, 2005.

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

[7] 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 and Experience*, Accepted in July 2006 for publication. Available at URL `http://www.cs.umn.edu/Ajanta/publications.html`.

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