A Policy Description Language for Context-Based Access Control and Adaptation in Ubiquitous Environment*

Joonseon Ahn[1], Byeong-Mo Chang[2], and Kyung-Goo Doh[3,**]

[1] Hankuk Aviation Universiy, Koyang, 412-791, Korea
jsahn@hau.ac.kr
[2] Sookmyung Women's University, Seoul, 140-742, Korea
chang@sookmyung.ac.kr
[3] Hanyang University, Ansan, 426-791, Korea
doh@hanyang.ac.kr

Abstract. The goal of our research is to provide an advanced programming environment for ubiquitous computing, which facilitates the development of secure and reliable ubiquitous software. The environment consists of a high-level ubiquitous programming framework, a run-time system enhanced with better context adaptation and security, and programming support tools.

In this paper, we focus on a ubiquitous programming framework, which includes a high-level policy description language, a translator to Java and a runtime system. We first present a high-level policy description language for formally specifying context entity relation, as well as context-based access control and adaptation policies. We then describe how a specification in the policy description language can be translated into Java code which makes use of JCAF.

1 Introduction

The vision of ubiquitous computing, where a large number of devices and sensors are embedded into their physical environment, providing contextual services to mobile users and applications, is progressing towards realization. Increases in the performance of hand-held and embedded devices along with improvements in networking technology are aiding this process.

The software technology for ubiquitous services will soon become the strategic core of future ubiquitous computing [6]. Thus the effective and secure ubiquitous computing environment is very important for ubiquitous computing [2].

Several research works have been done to provide software solutions for ubiquitous computing environment, which includes context-aware middleware [4], context-based security [5,3], and programming environment for ubiquitous service [1,8]. Recently, the programming environment for ubiquitous services has become more important for the effective development of ubiquitous software.

* This work was supported by grant No. R01-2006-000-10926-0 from the Basic Research Program of the Korea Science and Engineering Foundation.

** Corresponding author.

X. Zhou et al. (Eds.): EUC Workshops 2006, LNCS 4097, pp. 650–659, 2006.

The goal of our research is to develop an advanced programming environment for ubiquitous computing, which facilitates the development of secure and efficient ubiquitous software. The programming environment consists of three components: a high-level ubiquitous programming framework, a run-time system enhanced with better context adaptation and security, and programming support tools for program analysis and monitoring.

In this paper, we propose a new ubiquitous programming framework, which includes a high-level policy description language, a translator for the language and a runtime system. The ubiquitous programming framework is to support the application developer to easily write secure ubiquitous applications with context-centric ubiquitous services.

Using the policy description language, programmers can describe a high-level specification on context space, context-based security, and context-based adaptation for ubiquitous application programs. A high-level specification consists of spatial context model description, context-based access control rules, and context-based adaptation rules. A spatial context model describes the context world as a tree of nested *entities*, analogous to ambient in the ambient calculus.

Context-based access control is described by access control rules, which specify access privileges of an entity in a context. Context-based adaptation is described by adaptation rules, which specify an action to perform when a condition becomes satisfied in a dynamic context.

We also describe how a specification in the policy description language can be translated into Java code which makes use of JCAF [1]. Our translator provides automatic generation of Java classes for ubiquitous entities. It can relieve programmers from dealing with complex programming related to context space, context adaptation and security. Then we explain our runtime system for supporting context-aware access control in dynamically changing environments.

The rest of the paper is organized as follows. The next section presents the policy description language. Section 3 presents the translation from the policy description language into Java and the runtime system for access control. Section 4 describes some related works and Section 5 concludes this paper.

2 Policy Description Language

A policy specification consists of three parts: entity relation definitions, access control rules, and adaptation rules. Declared first are relations between context entities to be used in the specification, and then access control rules and adaptation rules follow. An example of policy specification is shown in Fig.1.

2.1 Entity Relation Definitions

A *context entity* in ubiquitous environment is either a physical or logical space, a fixed object, or a moving object. For example, consider the context of a software company: "building", "lobby", "floor", "room", and "lounge" are space entities; "printer" is a fixed-object entity; and "PDA" is a moving-object entity. Each entity of the real world corresponds to an instance of an *entity class* in programs.

```
-- Entity Relation Definitions
Building:ubisoft[Floor[Room[Printer]+Lounge[Printer]]+Lobby[Printer]],
Pda!IsIn(Room), Pda!IsIn(Lobby), Pda!IsIn(Lounge),
Pda!Employed(Building), Pda!Owns(Room), Pda!Hosts(Pda),
Pda!Friends(Pda)

-- Access Control Rules
(Building:ubisoft/$Lobby/$Pda,$Lobby/$Printer.print, true, CALL);
($Pda, $Room/$Printer.print, $Pda!Owns($Room), CALL);
($Pda,$Lounge/$Printer.print,$Pda!Employed(Building:ubisoft),CALL);
($Pda_1,$Room/$Printer.print,$Pda_2!Hosts($Pda_1)^$Pda_2!Owns($Room),CALL);
($Pda_2, $Pda_2!Hosts($Pda_1), $Pda_2!Friends($Pda_1),MODIFY)

-- Adaptation Rules
$Pda_1!IsIn($Room) ^ $Pda_2!IsIn($Room)
   => $Pda_2!Hosts($Pda_1) if $Pda_2!Owns($Room);
$Pda!IsIn(Building:ubisoft/.../$Room)
   => $Pda.registerPrinter($Room/$Printer);
$Pda!IsIn(Building:ubisoft/$Lobby)
   => $Pda.registerPrinter($Lobby/$Printer);
$Pda!IsIn(Building:ubisoft/$Lounge)
   => $Pda.registerPrinter($Lounge/$Printer);
```

Fig. 1. An Example of a Policy Specification

For example, floors of a building can be represented as a class named `Floor`, and each floor corresponds to an instance of the class.

The type of a relation between entities in policy specification must be defined before their use. The general form is a triple, $id_1!id_2(id_3)$, where id_2 is the name of a relation, and id_1 and id_3 are the names of entity classes (types). For example, `Pda!IsIn(Room)` defines that `IsIn` is a relation between a `Pda` entity and a `Room` entity. Fig.1 shows some examples of entity relation definitions.

A physical space and fixed object can be represented as a nesting hierarchical structure since it has a structure inside which other spaces and objects are nested. We describe the nested hierarchical structure as a tree which naturally defines spatial containment relation among spaces and fixed objects. For example, the tree representation in the second line of Fig.1 indicates that: (1) A `Building` instance, `ubisoft`, is a root having children nodes, `Floor` and `Lobby`; (2) `Floor` has two children nodes, `Room` and `Lounge`; and (3) each of `Room`, `Lounge` and `Lobby` have as a child, `Printer`. The meaning of this tree is that (1) there are lobbies in the `ubisoft` building; (2) there are rooms and lounges in each floor of building; and (3) there are printers in every lobby, room and lounge in the building. In fact, we can think of this representation as a syntactic sugar of seven `Contains` relations (or `IsIn` relations when its arguments are in reverse order) as follows:

```
Building!Contains(Floor), Building!Contains(Lobby),
Floor!Contains(Room), Floor!Contains(Lounge),
```

```
Room!Contains(Printer), Lounge!Contains(Printer),
Lobby!Contains(Printer)
```

The specific name of an instance can be specified in the definition along with its class name when statically determinable and necessary. Note that an instance name must always be preceded by a class name and a colon for clarity.

The formal syntax for entity relation definition is defined as follows:

$$
\begin{aligned}
c &\in \text{Context-Relation} ::= id_1!id_2(id_3) \mid s \mid c_1, c_2 \\
s &\in \text{Space-Relation} ::= id \mid id_1 : id_2 \mid id[s] \mid id_1 : id_2[s] \mid s_1 + s_2 \mid \epsilon \\
id &\in \text{Identifier}
\end{aligned}
$$

2.2 Access Control Rules

An access control rule specifies that the given subject entity has the given access mode to the given object when the given condition is met. We describe the rule to be a quadruple consisting of a subject, an object, a condition, and an access mode as in the following syntax:

$$
\begin{aligned}
x &\in \text{Access-Rule} ::= (p,\ o,\ r,\ m) \mid x_1\ ;\ x_2 \\
p &\in \text{Entity-Expression} ::= id_1 : id_2 \mid \$id \mid \$id\_n \mid * \mid p_1/p_2 \mid \ldots/p \\
o &\in \text{Object} ::= p.id \mid p_1!id(p_2) \\
r &\in \text{Relation-Expression} ::= p_1!id(p_2) \mid \sim r \mid r_1 \wedge r_2 \\
m &\in \text{Mode} ::= \text{READ} \mid \text{MODIFY} \mid \text{CALL} \\
n &\in \text{Number}
\end{aligned}
$$

The subject is an entity instance in context and is described as an entity expression. An entity expression, $id_1 : id_2$, represents an entity instance where id_2 is the name of the instance and id_1 is the name of its class, e.g., `Pda:pdaBob`. For space entities, we write an entire path (sequence of entity names) describing a route through the space entity hierarchy from the root, which is thus called a *path expression*. For example, a path expression, `Building:ubisoft/Floor:fl1/Room :rmBob`, represents a `Room` entity named `rmBob` in the floor named `fl1` inside the building named `ubisoft`. $\$id$ is a placeholder representing an instance of a class named *id*. The placeholder may be numbered when two or more different ones of the same class are needed. Entity expressions employ regular expression-like notation to effectively name some entity. For example, a path expression, `Building:ubisoft/.../$Room`, indicates a `Room` instance in the `ubisoft` instance of `Building`, and `Building:ubisoft/Floor:fl1/*` represents any entity instance in the `fl1` floor inside the `ubisoft` building.

A context relation expression describes a relation between entities in context, and is represented as $p_1!id(p_2)$, where p_1 and p_2 are entity expressions, and *id* is a relation name. The triple means that an entity represented by p_1 has an *id* relation with the one by p_2, which is always interpreted as either true or false. For example, a relation "a PDA is in Bob's room" can be expressed as: `$Pda!IsIn(Room:rmBob)`. Technically, when this relation expression evaluates to true, the placeholder `$Pda` represents a `Pda` instance in the Bob's room. We

employ a logical "not" operator, $\sim$, to express the negation of a relation, and a logical "and" operator, $\wedge$, to express the conjunction of two relations.

An object is either an entity's method name, $p.id$, or a context relation expression, $p_1!id(p_2)$. A condition is a dynamic context relation which needs to be met in order to grant the given access mode. The READ/MODIFY mode indicates the right to read/modify a dynamic relation in the context, and the CALL mode the right to call a method.

Let us examine some examples of access control rules in Fig. 1. The first rule says that any PDA located in any lobby in the `ubisoft` building has the permission to use a printer at the lobby. The second rule permits that only the owner of a room can use the printer in her room. The third rule lets all employers of the `ubisoft` building use printers in lounges. The fourth rule restricts that anyone who is the guest of a room's owner can use the printer in the owner's room. The last rule lets a PDA set a `Hosts` relation with a guest PDA only when they have a `Friends` relation.

2.3 Adaptation Rules

An event occurs when the change of a relation in context takes place. For example, "Bob's PDA enters Bob's room" is an event that sets `Pda:pdaBob!IsIn (Room :rmBob)` to true. An adaptation rule specifies how to respond when an event occurs in a given context. The syntax of adaptation rules is as follows:

$$\begin{aligned} d \in \text{Adaptation-Rule} &::= r \Rightarrow a \text{ if } b \mid d_1 \; ; \; d_2 \\ a \in \text{Action} &::= p_1.id(p_2) \mid p_1!id(p_2) \mid a_1 \; ; \; a_2 \end{aligned}$$

where b represents either a Java conditional expression or a context relation expression.

Let's look through the examples in Fig.1. The first rule specifies that when a guest PDA enters a room where the owner PDA is in the room, the owner sets the `Hosts` relation with the guest PDA to true. The second rule says that when a PDA enters a room equipped with a printer in the Ubisoft building, the PDA registers the printer in the room. The third(last) rule describes that when a PDA enters the lobby(lounge) of the Ubisoft building, the PDA registers the printer in the lobby.

3 Implementation of Policy Files

Given a policy specification file, we generate Java classes for context entities described in the file. Adaptation rules are translated into Java methods which execute adaptation actions when specified context-change events take place. Access-control rules are implemented as the runtime access controller which maintains the rules and controls the access. Our implementation makes use of JCAF [1].

3.1 JCAF

In [1], Bardram presents Java Context-Awareness Framework(JCAF) which is a Java-based context-awareness infrastructure and API for creating context-aware applications. The core modelling interfaces provided by JCAF are `Entity`,

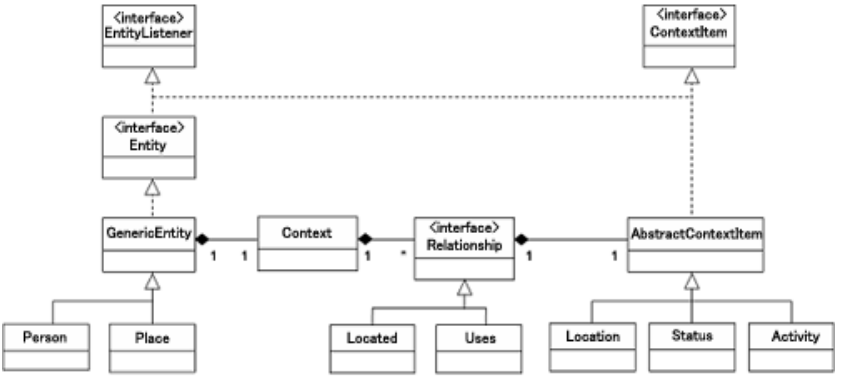

Fig. 2. UML model of an Entity with a Context in JCAF[1]

`Context`, `Relationship`, and `ContextItem` as illustrated in Fig.2. JCAF is also equipped with default implementations of these core interfaces. For example, the `GenericEntity` class implements the `Entity` interface and can be used to create concrete entities through specialization. `Person` and `Place` are some examples of entities. A Hospital Context and an Office Context, each knowing specific aspects about a hospital and an office, respectively, are examples of context. Physical location and the status of an operation are examples of abstract context items. Examples of relations are `Located` or `Uses`. Hence, we can model that a *PDA* is *located* in a *lobby*, where a *PDA* is an Entity, *located* is a relation, and a *lobby* is a context item.

The central processing part of an Entity is its `contextChanged()` method in the `EntityListener` interface. This method is guaranteed to be called by the entity container whenever this entity's context is changed. This is a very powerful way of handling context changes effectively. That is, for each possible event of context changes, an appropriate action to be taken can be defined for users of applications. For example, suppose that we want to turn a TV's power switch on when a person approaches to the TV within a viewable distance. Then we can add the following `contextChanged()` method to the TV class:

```
public void contextChanged(ContextEvent event) {
  // If someone approaches to a TV within a viewable distance,
  // then turn the TV's power switch on.
}
```

The `contextChanged()` method is called by the JCAF runtime system as soon as the TV's context is changed. Then the `contextChanged()` method examines the event and executes an appropriate adaptation action accordingly.

3.2 Translation of Adaptation Rules

Adaptation rules are implemented using the `contextChanged` method. An adaptation rule $p_1!id_1(p_1') \wedge p_2!id_2(p_2') \wedge \ldots \wedge p_n!id_n(p_n') => a$ `if` b is applied when

```
public void contextChanged (ContextEvent e) {
   Entity x1, x2, x3; Entity[] t; int i;
   if (e.getRelationship instanceof IsIn) {
      x1 = e.getItem();
      if (x1 instanceof Room) {
         t = getAllEntitiesByType(Class.forName("Pda"));
            for (i=0; i<t.length; x2 = t[i++])
               if (x2.getContext().getContextItem(new IsIn()) == x1)
                  if (x2.getContext().getContextItem(new Owns()) == x1)
                     addContextItem(x2.getId(), new Hosts(), this);
   }}
   if (e.getRelationship instanceof IsIn) {
       x1 = e.getItem();
       if (x1 instanceof Room) {
          t = getAllEntitiesByType(Class.forName("Pda"));
          for (i=0; i<t.length; x2 = t[i++])
             if (x2.getContext().getContextItem(new IsIn()) == x1)
                if (this.getContext().getContextItem(new Owns()) == x1)
                   addContextItem(this.getId(), new Hosts(), x2);
   }}
   // ... if statements for dealing with other context change events
}
```

Fig. 3. A part of the `contextChanged` method of `Pda` class

some context value of entity p_i is changed and the adaptation condition is satisfied as a result. Therefore, the rule is used to generate those `contextChanged` methods which belong to the entity classes represented by $p_1, \ldots, p_n$.

Fig.3 shows two `if` statements of the `contextChanged` method of `Pda` entity class. They are generated from the following adaptation rule from Fig. 1.

```
$Pda_1!IsIn($Room) ^ $Pda_2!IsIn($Room)
     => $Pda_2!Hosts($Pda_1) if $Pda_2!Owns($Room)
```

The `Pda` object whose `contextChanged` method has been called can be either `$Pda_1` or `$Pda_2` where the former case is dealt with by the first `if`-statement and the latter is dealt with by the second. The first `if`-statement deals with the case that a PDA enters a room while the owner of the room is in the room. The second `if`-statement is for the case that the PDA which is the owner of a room enters its room while other PDA's are in the room.

In the first `if`-statement, `x1` is bound to `$Room` entity and `this` and `x2` is bound to `$Pda_1` and `$Pda_2` entity, respectively. In the `if`-statement, we bind related entities and examine the types of related entities and other conditions for the adaptation action. Because there can be multiple PDA's in the current context, the statements for binding `$Pda_2` is included in the `for`-statement which examines each `Pda` entity. `(x2.getContext().getContextItem(new IsIn()) == x1)` examines whether `x2` and `x1` is in `IsIn` relation, where `getContext()` returns the current context of an entity and `getContextItem(`<*relation*>`)` returns

the entity which is in the <*relation*> relation. `addContextItem(x2.getId(), new Hosts(), this)` replaces the `Hosts` relation value of `x2` with `this`. For the second `if`-statement, `x2` and `this` is bound to `$Pda_1` and `$Pda_2` entity of the event, respectively, and the statements are generated analogously.

An expression $p_1!id(p_2)$ included in the adaptation rule $r \Rightarrow a$ `if` b has different meaning based on where it is used. If it occurs in the r or b part, the translated code examines whether the *id* relation value of p_1 is p_2 using the `getItem()` method of JCAF. If it occurs in the a part, the generated statement replaces the *id* relation value of p_1 with the entity represented by p_2 using the method `addContextItem`.

3.3 Access Control Implementation

Access control policies are managed and forced by Context-aware Access Control Manager(CACM) with the help of `ContextService` runtime system. Before executing a method, or reading or updating a context value, the entity object consults CACM whether or not the request can be accepted.

For those methods under access control, we insert a method call

`checkMethodAccess(`<*method name*>`)`

Given the call, CACM decides whether the request should be accepted considering the method name and contexts of related entity objects which can be identified by inspecting stack frames. If the request is permitted, the method returns normally. Otherwise, CACM disallows the method call by raising an exception.

Access control for context relation value is implemented using the call

`checkRelationAccess(`<*entity*$_1$>`,`<*entity*$_2$>`,`<*relationship*>`,`<*mode*>`)`

which consults CACM whether the requesting entity can access <*entity*$_1$>'s <*relationship*> context value, where <*mode*> is either `READ` or `MODIFY`.<*entity*$_2$> is used for the modification case as a new context value and is ignored for the read access. The method calls are inserted at the beginning of the `addContextItem` method and the `getContextItem` method which updates and reads context relation value, respectively.

CACM maintains associations between entities and their applicable permissions which change under dynamic context. To this end, CACM manages a hash table which maps a 4(or 3)-tuple of related entities and a relation(or method) name to a list of sufficient context conditions for the method call or context value access. Because, there can be multiple sufficient conditions for an access, each condition becomes an element of the list.

Given a query for access control, CACM maps the tuple of related entity classes and a method or relation name to a list of representations of sufficient conditions. Then, CACM examines whether there exists any condition which is satisfied under the current dynamic context. If CACM finds such one, it allows the requested access. Otherwise, if no condition is satisfied, CACM refuses the access by rasing an exception.

4 Related Works

Several research works have been done to provide software solutions for ubiquitous computing environment, which includes programming framework, context-aware middleware, and context-based security for ubiquitous service [7,1,5,4,8].

Bellavista et al. developed a middleware for context-aware resource management, called CARMEN, capable of supporting the automatic reconfiguration of wireless Internet services in response to context changes without any intervention on the service logic [4]. CARMEN determines the context on the basis of metadata, which include declarative management policies and profiles for user preferences, terminal capabilities, and resource characteristics.

Roman et al. also proposed a middleware infrastructure called Gaia to provide support for mobile user-centric active space applications [7]. It manages the resources and services of an active space. It provides services for location, context, and events, and repositories with information about the active space.

Corradi et al. introduced an access control model built upon the concept of context as the first-class design principle to rule access to resources [5]. This model associates access control permissions with contexts where users operate and users acquire/lose their permissions when entering/leaving a specific context, and exploits the user context to fully determine the set of available permissions. This model allows to express context-based access control policies at a high level of abstraction cleanly separate from service logic implementation.

Scott proposed a spatial model for describing mobil agents with respect to physical location of objects as well as virtual location of mobile code. The model comes with a mobility restriction policy language which can control the mobility of agent programs.

In [3], Cho and Lee described a security policy description model based on an ubiquitous language called PLUE, and a static checker to extract the rules that will be possibly fired under a given credential and a policy.

5 Conclusion and Future Works

We have designed an unified high-level policy description language for formally specifying space context relation, context-based security policies and context-based adaption. In our policy description language, programmers can specify a hierarchical spatial context easily, which includes physical and logical context space. Also, programmers can describe various dynamic relations among entities located in the spatial context. Based on them, programmers can develop a context-based access control and adaptation specification in a unified way. Our approach of the unified application of spatial context model and flexible relation specification to dynamic access control and context adaptation provides a simple and strong way to specify ubiquitous systems.

The final goal of our research is to develop an advanced programming environment for ubiquitous computing which facilitates the development of secure and adaptive ubiquitous software. We will design a type system for the specification language and check consistency of the policy specification automatically

based on the type system. To enforce the policy specification in the program execution, we will enhance the existing context-aware run-time system so as to provide context adaptation and access control based on the spatial model and rules in the specification.

We will also develop ubiquitous programming support tools for program analysis and monitoring. The program analysis tools help programmers develop secure and efficient programs based on static analysis such as context-flow analysis, access control analysis, secure information-flow analysis and exception analysis. The program monitoring tools allow programmers to examine the execution traces of programs such as context change, exception handling and access control during execution.

Acknowledgement

We would like to thank the anonymous reviewers for thoughtful comments and corrections.

References

1. J. E. Bardram, The Java Context Awareness Framework-A Service Infrastructure and Programming Framework for Context-Aware Applications, Third International Conference, Pervasive 2005, Munich, Germany, May, 2005.
2. V. Cahill et. al, Using Trust for Secure Collaboration in Uncertain Environment, Pervasive computing, July-September 2003, pp52-61.
3. E. Cho and K. Lee, Security Checks in Programming Languages for Ubiquitous Environments, *Proceedings of 2004 Workshop on Pervasive, Security, Privacy and Trust*, Aug. 2004.
4. P. Bellavista, A. Corradi, R. Montanari, Context-Aware Middleware for Resource Management in the Wireless Internet, *IEEE Transactions on Software Engineering* , Vol. 29, No. 12, December 2003.
5. Antonio Corradi, Rebecca Montanari, Daniela Tibaldi, Context-based Access Control for Ubiquitous Service Provisioning, *Proceedings of the 28th International Computer Software and Applications Conference(COMPSAC'04)*, 2004.
6. T. Kindberg, A. Fox, System Software for Ubiquitous Computing, Pervasive computing, January-March 2003, pp. 70-81.
7. M. Roman, C.K. Hess, R. Cerqueira, A. Ranganat, R.H. Campbell, K. Nahrstedt, Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive Computing, pp. 74-83, 2002
8. D. J. Scott, Abstracting application-level security policy for ubiquitous computing, University of Cambridge, Computer Laboratory, Technical Report UCAM-CL-TR-613, January 2005.
9. D. Wichadakul, X. Gu and K. Nahrstedt, A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications, *Proceedings of Multimedia'02*, December, 2002, Juan-les-Pins, France.