2007 International Conference on Convergence Information Technology

A Type System for Checking Consistencies of a Policy Specification Used in Ubiquitous Programming Environment*

Ki-Hwan Choi, Hye-Ryeong Jeong, and Kyung-Goo Doh†
Hanyang University
Ansan, 426-791, Korea
{*khchoi, hrjeong, doh*}*@pllab.hanyang.ac.kr*

Joonseon Ahn
Hankuk Aviation University
Koyang, 412-791, Korea
jsahn@hau.ac.kr

Byeong-Mo Chang
Sookmyung Women's University
Seoul, 140-742, Korea
chang@sookmyung.ac.kr

Abstract

The high-level policy description language used for ubiquitous programming framework specifies context entity relations, as well as context-based access control and adaptation rules. Then the specification in the policy description language is translated into the code in a general-purpose language, which is to be used in ubiquitous environment. However, the inconsistencies and errors in the policy specification are all passed into the translated code, potentially resulting disastrous malfunction. This paper introduces a type system that checks the consistency of a policy specification so that the specification is free from type-related errors and inconsistencies.

1. Introduction

Along with the development of hardware for ubiquitous computing, the software technology for the effective and secure ubiquitous programming environment should also be advanced [6].

Several research works have been done to provide software solutions for ubiquitous programming environment, which includes context-aware middleware [3] and programming environment for ubiquitous service [2, 8]. However, they lack an effective medium to specify policies regarding access control and adaptation rules. To fill up this efficiency, a Policy Description Language[1] is developed to specify access control and adaptation policies for ubiquitous computing environment.

The high-level policy description language used for ubiquitous programming framework specifies context entity relations, as well as context-based access control and adaptation rules. Access control rules specify the access privileges of an entity in a given context. Adaptation rules specify an action to perform when a certain condition is met in a given dynamic context. Using the policy description language, programmers can describe a high-level policy specification for ubiquitous application programs. Then the specification in the policy description language is translated into the code in a general-purpose language, which is to be used in ubiquitous environment. However, the inconsistencies and errors in the policy specification are all passed into the translated code, potentially resulting disastrous malfunction. This paper introduces a type system that checks the consistency of a policy specification so that the specification is free from type-related errors and inconsistencies, which facilitates the development of secure and reliable ubiquitous software.

The rest of the paper is organized as follows. Section 2 presents the policy description language. Section 3 discusses entity types and Section 4 explains our typing algorithm. Section 5 describes some related works. Section 6 concludes this paper and outlines future works.

2. Policy Description Language

A policy specification consists of three parts: entity relation definitions, access control rules, and adaptation rules. Defined first are relations between context entities to be used in the specification, and then access control rules and

*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.

0-7695-3038-9/07 $25.00 © 2007 IEEE
DOI 10.1109/ICCIT.2007.260

adaptation rules follow.

2.1. Entity Relation Definition

A context entity in ubiquitous environment is either a physical or logical space, a fixed object, or a moving object. The locations of space entities and fixed entities are not changed and thus static, while those of moving objects may be changed and thus dynamic. Each entity of the real world corresponds to an instance of an entity class in a program.

The type of a relation between entities in policy specification must be defined before its use. The general form of entity relation definition is a quadruple of form $id_1(id_2, id_3, id_4)$, where id_1 is the kind of a relation, id_3 is the name of a relation, and id_2, id_4 are the names of entity classes. For example, `Location(Pda,IsIn,PCRoom)` defines that `IsIn` is the name of a relation of `Location` kind between `Pda` entity and `PCRoom` entity.

Among entity relations, space entities may have inclusion relations (such as the `IsIn` relation above). Since relations on space entities are transitive, i.e., space entities have other space entities and fixed objects nested inside, it is convenient to represent them alternatively as a tree-like structure. For example, the following definition represents the hierarchical inclusion relations among space entities in `School`.

```
School[Floor[PCRoom[Printer+Projector]
             +Lounge[]
             +Laboratory[Desktop+Printer]
             +MeetingRoom[Projector]
             +Toilet]
       +Lobby[Printer]]
```

This representation can be viewed pictorially as in Figure 1, which indicates that: (1) a `School` entity can have `Floor` and `Lobby` entities inside; (2) a `Floor` entity can have `PCRoom`, `Lounge`, `Laboratory`, `MeetingRoom`, and `Toilet` entities inside; (3) a `PCRoom` entity can have `Printer` and `Projector` entities inside; (4) a `Laboratory` entity can have `Desktop` and `Printer` entities inside; (5) a `MeetingRoom` can have `Projector` entities inside; and (6) a `Lobby` entity can have `Printer` object inside. This tree-like representation helps check the transitivity in space inclusion relations with ease.

When statically determinable and necessary, the specific instance name of a space entity class can be declared along with its class name as follows:

```
School:ubischool
       [Floor[PCRoom[Printer+Projector]
              +Lounge[]
              +Laboratory[Desktop+Printer]
              +MeetingRoom[Projector]
              +Toilet]
        +Lobby[Printer]]
```

which declares that the space inclusion relation defined is for a `School` instance named `ubischool`.

The formal syntax for entity relation definition is as follows:

$id \in$ Identifier
$c \in$ Context-Relation $::= id_1(id_2, id_3, id_4) \mid c_1, c_2 \mid s$
$s \in$ Space-Relation $::= id \mid id_1 : id_2 \mid id[s] \mid id_1 : id_2[s] \mid s_1 + s_2 \mid s_1, s_2 \mid \epsilon$

2.2. Entity Expression

An entity expression describes one or more entity instances of an entity class in context. An entity expression, $id_1 : id_2$, represents an entity instance id_2 of class id_1. $\$id$ represents the set of all instances of a class id. For space entities, the complete route through the space entity hierarchy from the root must be shown in an entity expression, which is represented as the sequence of space entity expressions starting from the root separated by '/'. We call the route a *path expression*. For example, a path expression,

```
School:ubischool/Floor:f2/PCRoom:pcrm203
```

represents a `PCRoom` entity named `pcrm203` in the `f2` floor in the `ubischool` building.

Entity expressions employ regular expression-like notation to effectively name some space entities. For example, a path expression,

```
School:ubischool/Floor:f1/*
```

represents the set of all instances in the `f1` floor in the `ubischool` school.

```
School:ubischool/.../$PCRoom
```

indicates the set of all `PCRoom` instances in `ubischool`. That is, `p1/.../p2` is the convenient form of expressing multiple paths in a single path expression.

The formal syntax for path expression is defined as follows:

$id \in$ Identifier
$n \in$ Number
$p \in$ Entity-Expression $::= q/p \mid q$
$q \in$ Factor $::= id_1 : id_2 \mid \$id \mid \$id\_n \mid * \mid .../q$

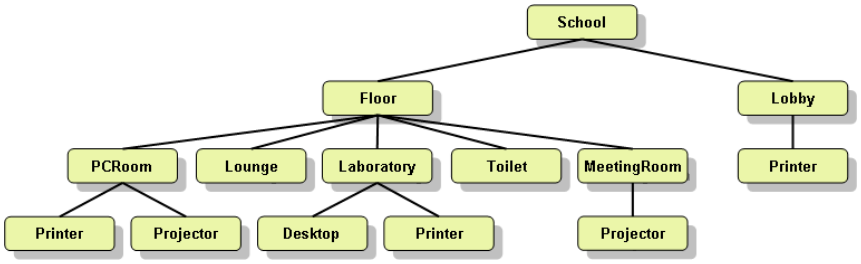

Figure 1. A Tree Representation

2.3. Access Control Rules

An access control rule specifies that the given set of entities has the given right to the given object when the given condition is met. The syntax of access control rules is as follows:

$x \in$ Access-Rule $::= (p_1, p_2.id, r) \mid x_1 \ ; \ x_2$
$r \in$ Relation-Expression $::= id_1(p_1, id_2, p_2) \mid \sim r \mid r_1 \wedge r_2 \mid true$

We describe the rule to be a triple consisting of the subject, the object, and the condition. The subject is the set of entities; the object is either an entity's method name or a relation name; and the condition is a dynamic context relation which needs to be met in order for the access to be granted. For example, an access control rule,

```
(School:ubischool/$Lobby/$Pda,
          $Lobby/$Printer.print,true)
```

represents any PDA located in any lobby in the `ubischool` building has the permission to use a printer at the lobby.

A context relation expression, r, describes relations between entities in context. A relation expression is interpreted as either true or false. For example, a relation "a PDA is in PCRoom 103" can be expressed as:

```
Location($Pda,IsIn,PCRoom:pcrm103)
```

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.

2.4. Adaptation Rules

An adaptation rule specifies how to respond when an event occurs in a given context. The syntax of adaptation rules is as follows:

$d \in$ Adaptation-Rule $::= r \Rightarrow a \mid d_1 \ ; \ d_2$
$a \in$ Action $::= p_1.id(p_2) \mid id_1(p_1, id_2, p_2) \mid a_1 \ ; \ a_2$

For example, consider that if a PDA enters a PCroom, set the printer in the PCroom as a main printer of the PDA. In this example, an event, "a Pda enters a PCroom", will set the relation `Location($Pda,IsIn,$PCRoom)` to true.

```
Location($Pda,IsIn,$PCRoom)
^ Location($Printer,IsIn,$PCRoom)
    => $Pda.setMainPrinter($Printer)
```

3. Entity Types

An entity type consists of the entity's class name and its location. Thus the entity type must contains the path from the root in order to indicate the entity's location. For example, consider the following path expression:

`School:ubischool/Floor:f2/PCRoom:pcrm201`

Its class name is `PCRoom`, and its location information, `School/Floor/PCRoom`, can be derived by listing the class names attached throughout the path. Since a path expression can represent the set of entity instances, its type must include its type as well as the collection of locations showing where each entity instance is. In this paper, the type of a path expression is described as the set of all possible paths from the root to its entity class. For example, for the following expression:

```
School:ubischool/.../$Projector
```

its type according to the context in Figure 1 is:

```
{School/Floor/PCRoom/Projector,
 School/Floor/MeetingRoom/Projector}
```

Let's look at some more examples of path expressions with their corresponding types:

```
School:ubischool : {School}
   $Floor        : {School/Floor}
School:ubischool/$Floor/* :
          {School/Floor/PCRoom,
           School/Floor/Lounge,
           School/Floor/Laboratory,
           School/Floor/Toilet,
           School/Floor/MeetingRoom}
```

4. Typing Algorithm

4.1. Environments

The environments needed to check type consistency in a policy specification are a class tree (e.g., Figure 2) and an instance tree (e.g., Figure 3). From the entity relation

```
School[Floor[PCRoom[Printer+Projector]
             +Lounge[]
             +Laboratory[Desktop+Printer]
             +MeetingRoom[Projector]
             +Toilet]
       +Lobby[Printer]]
```

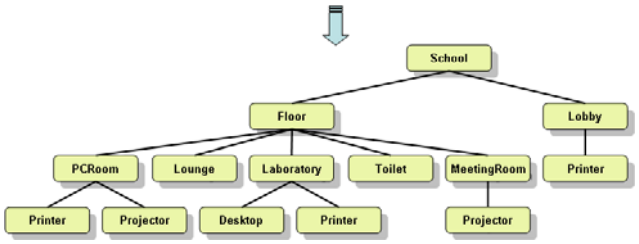

Figure 2. Class Tree

definition of a policy specification, a class tree can be built straightforwardly as in Figure 2. When a new instance variable is encountered, an instance tree grows with the variable attached to the tree. When an existing instance variable is encountered, the type consistency is checked.

```
School:ubischool[Floor:f2[PCRoom:pcrm201[]
                 +PCRoom:pcrm202[]]]
```

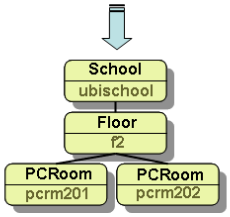

Figure 3. Instance Tree

4.2. Typing Entity Expression

Given the set of class trees and instance trees, the type of an entity expression is determined. First, the class tree is traversed to see whether or not the path expression conforms the inclusion relation. Second, the instance tree is traversed to check the type consistency of the instance variable. If the instance variable is found in the instance tree, then check the type consistency of the variable. Otherwise, the proper branch is attached to the tree. One that makes the typing complicated is the ellipsis in the middle of a path expression. Since we need to collect all the paths matched in the tree, we use a tag, *Ellipsis* or *Full*, to indicate that the path information in the middle is missing. The typing rules for an entity expression is shown in Figure 4 and Figure 5.

$$\begin{array}{rcl} \tau & \in & \mathcal{P}(\text{ClassTree}) \\ \eta & \in & \mathcal{P}(\text{InstanceTree}) \\ \pi & \in & \mathcal{P}(\text{Path-Type}) \\ t & \in & \text{Tag} = \{\text{Full, Ellipsis}\} \end{array}$$

$$\begin{array}{l} \mathcal{F} : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \times \text{Entity-Expression} \\ \quad \rightarrow \mathcal{P}(\text{String list}) \times \mathcal{P}(\text{InstanceTree}) \end{array}$$

$$\frac{\tau, \eta \vdash \mathcal{F}(\mathrm{p}, \tau, \eta) : \{\pi^t \mid \pi^t \in P\}, \eta'}{\tau, \eta \vdash \mathbf{p} : \{\pi\}, \eta'} (\text{MainFun})$$

Figure 4. Main Rule for an Entity Expression

The typing rules in Figure 5 are mostly self-explanatory. The function 'declared' checks the given class name is defined in the class tree. The function 'build' takes a class and instance name pair, checks if the instance name is in the instance tree, and returns the updated instance tree if it is not in the tree, or checks the consistency otherwise. The function 'parentOf' takes two entity names and checks if one is the parent of the other in the class tree, and the function 'ancestorOf' takes two entity names and checks if one is the ancestor of the other in the class tree. The function 'connectPath' takes two paths and returns all complete paths that connect the two.

4.3. Typing Relation Expressions

Typing relation expressions is merely to check the type of entity expressions in the relation expressions, and thus is straightforward. The type rules are shown in Figure 6. The function 'lookup' checks if the relation is defined.

$$\begin{array}{rcl} \tau & \in & \mathcal{P}(\text{ClassTree}) \\ \eta & \in & \mathcal{P}(\text{InstanceTree}) \\ \sigma & \in & \mathcal{P}(\text{Context}) \end{array}$$

$$\begin{array}{l} \_, \_, \_ \vdash_r \_ : \_ : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \\ \qquad \times \mathcal{P}(\text{Context}) \times \text{Relation-Expression} \times \text{Unit} \end{array}$$

$$\frac{\begin{array}{c}\tau, \eta \vdash \mathrm{p}_1 : P_1, \eta_1 \quad \tau, \eta_1 \vdash \mathrm{p}_2 : P_2, \eta_2 \\ \text{lookup}(\mathrm{id}_1, \text{last}(\pi_1), \mathrm{id}_2, \text{last}(\pi_2), \sigma) = true\end{array}}{\begin{array}{c}\tau, \eta, \sigma \vdash_r \mathbf{id_1}(\mathbf{p_1}, \mathbf{id_2}, \mathbf{p_2}) : () \\ \text{where } \pi_1 \in P_1,\ \pi_2 \in P_2\end{array}} (\text{Relation})$$

$$\frac{\tau, \eta, \sigma \vdash_r \mathrm{r} : ()}{\tau, \eta, \sigma \vdash_r \sim \mathbf{r} : ()} (\text{Neg})$$

$$\frac{\tau, \eta, \sigma \vdash_r \mathrm{r}_1 : () \quad \tau, \eta, \sigma \vdash_r \mathrm{r}_2 : ()}{\tau, \eta, \sigma \vdash_r \mathbf{r_1} \wedge \mathbf{r_2} : ()} (\text{And})$$

$$\tau, \eta, \sigma \vdash_r \mathbf{true} : () (\text{True})$$

Figure 6. Type Rules of Relation Expression

$$\begin{array}{rcl} \tau & \in & \mathcal{P}(\text{ClassTree}) \\ \eta & \in & \mathcal{P}(\text{InstanceTree}) \\ \pi & \in & \mathcal{P}(\text{Path-Type}) \\ t & \in & \text{Tag} \end{array}$$

$$\_,\_ \vdash \_ : \_,\_ : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \times \text{Entity-Expression} \times \mathcal{P}(\text{Path-Type}) \times \mathcal{P}(\text{InstanceTree})$$

$$\frac{\tau,\eta \vdash \mathrm{q} : Q,\eta' \quad \tau,\eta' \vdash \mathrm{p} : P,\eta''}{\tau,\eta \vdash \mathbf{q}/\mathbf{p} : \{\texttt{case}\ \pi_2^{t_2}\ \texttt{of}} \text{(PathEntity)}$$

$$\begin{array}{rcl} \pi_2^{\text{Full}} & \Rightarrow & \texttt{if}\ \text{parentOf}(\text{last}(\pi_1),\text{first}(\pi_2),\tau) \\ & & \texttt{then}\ (\pi_1 @\ \pi_2)^{\text{Full}} \\ \pi_2^{\text{Ellipsis}} & \Rightarrow & \texttt{if}\ \text{ancestorOf}(\text{last}(\pi_1),\text{first}(\pi_2),\tau) \\ & & \texttt{then}\ \text{connectPath}(\pi_1,\pi_2,\tau)^{\text{Full}} \\ & & \quad \mid \pi_1^{t_1} \in Q,\ \pi_2^{t_2} \in P\},\eta'' \end{array}$$

$$\tau,\eta \vdash \mathbf{q} : \{\pi^t \mid \pi^t \in Q\},\eta' \text{(Factor)} \qquad \tau,\eta \vdash \mathbf{id_1} : \mathbf{id_2} : \{[\,\mathrm{id}_1\,]^{\text{Full}}\}, \text{build}(\mathrm{id}_1,\mathrm{id}_2,\eta) \ \text{ if declared?}(\mathrm{id}_1,\tau)\text{(EntInstance)}$$

$$\tau,\eta \vdash \$\mathbf{id} : \{[\,\mathrm{id}\,]^{\text{Full}}\},\eta \ \text{ if declared?(id,}\tau\text{)(EntSet)}$$

$$\tau,\eta \vdash \$\mathbf{id\_n} : \{[\,\mathrm{id}\,]^{\text{Full}}\},\eta \ \text{ if declared?(id,}\tau\text{)(EntWithNum)}$$

$$\tau,\eta \vdash * : \{entity^{\text{Full}} \mid entity \in \tau\},\eta\text{(Star)} \qquad \frac{\tau,\eta \vdash \mathrm{q} : Q,\eta'}{\tau,\eta \vdash \ldots/\mathbf{q} : \{\pi^{\text{Ellipsis}} \mid \pi^t \in Q\},\eta'}\text{(PathEntAll)}$$

Figure 5. Type Rules of Entity Expression

4.4. Typing Access Control and Adaptation Rules

The typing rules for access control and adaptation rules also merely checks if the entity expressions and the relation expressions are typable. They are shown in Figure 7 and in Figure 8, and are self-explanatory. The function 'checkMethod' is to check if the give class has the given method.

$$\begin{array}{rcl} \tau & \in & \mathcal{P}(\text{ClassTree}) \\ \eta & \in & \mathcal{P}(\text{InstanceTree}) \\ \sigma & \in & \mathcal{P}(\text{Context}) \end{array}$$

$$\_,\_,\_ \vdash \_ : \_ : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \times \mathcal{P}(\text{Context}) \times \text{Access-Rule} \times \text{Unit}$$

$$\frac{\tau,\eta \vdash \mathrm{p}_1 : P_1,\eta_1 \quad \tau,\eta_1 \vdash \mathrm{p}_2 : P_2,\eta_2 \quad \text{checkMethod}(\text{last}(\pi_2),\mathrm{id}) = true \quad \tau,\eta_2,\sigma \vdash \mathrm{r} : ()}{\tau,\eta,\sigma \vdash (\mathbf{p_1},\mathbf{p_2}.\mathbf{id},\mathbf{r}) : ()} \text{(AccRule)}$$

where $\pi_2 \in P_2$

$$\frac{\tau,\eta,\sigma \vdash \mathrm{x}_1 : () \quad \tau,\eta,\sigma \vdash \mathrm{x}_2 : ()}{\tau,\eta,\sigma \vdash \mathbf{x_1};\mathbf{x_2} : ()} \text{(AccRuleSeq)}$$

Figure 7. Type Rules of Access Control Rule

$$\begin{array}{rcl} \tau & \in & \mathcal{P}(\text{ClassTree}) \\ \eta & \in & \mathcal{P}(\text{InstanceTree}) \\ \sigma & \in & \mathcal{P}(\text{Context}) \end{array}$$

$$\_,\_,\_ \vdash \_ : \_ : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \times \mathcal{P}(\text{Context}) \times \text{Adaption-Rule} \times \text{Unit}$$

$$\_,\_,\_ \vdash_a \_ : \_ : \mathcal{P}(\text{ClassTree}) \times \mathcal{P}(\text{InstanceTree}) \times \mathcal{P}(\text{Context}) \times \text{Action} \times \text{Unit}$$

$$\frac{\tau,\eta,\sigma \vdash \mathrm{r} : () \quad \tau,\eta,\sigma \vdash_a \mathrm{a} : ()}{\tau,\eta,\sigma \vdash \mathbf{r} \Rightarrow \mathbf{a} : ()} \text{(AdapRule)}$$

$$\frac{\tau,\eta,\sigma \vdash \mathrm{d}_1 : () \quad \tau,\eta,\sigma \vdash \mathrm{d}_2 : ()}{\tau,\eta,\sigma \vdash \mathbf{d_1};\mathbf{d_2} : ()} \text{(AdapRuleSeq)}$$

$$\frac{\tau,\eta \vdash \mathrm{p}_1 : P_1,\eta_1 \quad \tau,\eta_1 \vdash \mathrm{p}_2 : P_2,\eta_2 \quad \text{checkMethod}(\text{last}(\pi_1),\mathrm{id}) = true}{\tau,\eta,\sigma \vdash_a \mathbf{p_1}.\mathbf{id}(\mathbf{p_2}) : ()} \text{(MethodCall)}$$

where $\pi_1 \in P_1$

$$\frac{\tau,\eta \vdash \mathrm{p}_1 : P_1,\eta_1 \quad \tau,\eta_1 \vdash \mathrm{p}_2 : P_2,\eta_2 \quad \text{lookup}(\mathrm{id}_1,\text{last}(\pi_1),\mathrm{id}_2,\text{last}(\pi_2),\sigma) = true}{\tau,\eta,\sigma \vdash_a \mathbf{id_1}(\mathbf{p_1},\mathbf{id_2},\mathbf{p_2}) : ()} \text{(Relation)}$$

where $\pi_1 \in P_1,\ \pi_2 \in P_2$

$$\frac{\tau,\eta,\sigma \vdash_a \mathrm{a}_1 : () \quad \tau,\eta,\sigma \vdash_a \mathrm{a}_2 : ()}{\tau,\eta,\sigma \vdash_a \mathbf{a_1};\mathbf{a_2} : ()} \text{(ActionSeq)}$$

Figure 8. Type Rules of Adaptation Rule

5. Related Works

There are some research works done in developing programming environment for ubiquitous computing.

Cho and Lee described a security policy description model based on an ubiquitous language called PLUE [4], and a static checker to extract the rules that will be possibly fired under a given credential and a policy.

A. Ranganathan and Roy H. Campbell developed a context-aware system called Gaia [7]. This system gathers various context information according to kind of relation and this information is analyzed and disposed using first-order logic. This system reacts to context circumstance with context defined in configuration file. However, in this configuration file, only instance values can be used.

Scott proposed the approach of using new application level security policy languages in combination to protect vulnerable applications [9]. Policies are abstracted from main application code, facilitating both analysis and future maintenance. They proposed three new application-level policy description languages for preventing application-level vulnerabilities.

Kagal, Finin, and Joshi suggests a middle-ware which enables access control policy description to define users' view [5]. But they elaborated only on the data model without consideration of application development.

However, no work has been done for checking the consistency of a policy specification, which enables programmers to make sure that their policy specification is consistently described and error free.

6. Conclusion and Future Works

This paper proposes a type system that automatically checks the consistency of types and space inclusion relations defined in a policy specification for ubiquitous programming environment. The system relieves programmers from battling with inconsistent policy specification by providing a compile-time checker.

Future works include the design and implementation of program analysis and monitoring tools for ubiquitous programming environment, as well as the enhancement of our policy specification and its type system employing time elements.

References

[1] J. Ahn, B. Chang, and K. Doh. A Policy Description Language for Context-based Access Control and Adaptation in Ubiquitous Environment. *Lecture Notes in Computer Science*, Vol.4097:pp. 650–659, August 2006.

[2] J. E. Bardram. The Java Context Awareness Framework-A Service Infrastructure and Programming Framework for Context-Aware Applications. Munich, Germany. Third International Conference, Pervasive 2005.

[3] P. Bellavista, A. Corradi, and R. Montanari. Context-Aware Middleware for Resource Management in the Wireless Internet. *IEEE Transactions on Software Engineering*, Vol. 29, No. 12, December 2003.

[4] E. Cho and K. Lee. Security Checks in Programming Languages for Ubiquitous Environments. *Proceedings of 2004 Workshop on Pervasive, Security, Privacy and Trust*, August 2004.

[5] L. Kagal, T. Finin, and A. Joshi. Moving from Security to Distributed Trust in Ubiquitous Computing Environments. *IEEE Computer*, December 2001.

[6] T. Kindberg and A. Fox. System Software for Ubiquitous Computing. *IEEE Pervasive computing*, pp. 70-81, January-March 2002.

[7] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. Springer-Verlag London Limited 2003, November 2003.

[8] M. Roman, C. Hess, R. Cerqueira, A. Ranganat, R. Campbell, and K. Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, pp.74-83, 2002.

[9] D. J. Scott. Abstracting application-level security policy for ubiquitous computing. Technical Report UCAM-CL-TR-613, University of Cambridge, Computer Laboratory, January 2005.