Embedding security policies into pervasive computing systems

Pengfei Liu

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux
N° d'ordre : 4735

Intégration de politiques de sécurité dans des systèmes ubiquitaires

THÈSE

soutenue le 17 Janvier 2013

pour l'obtention du

Doctorat de l'Université de Bordeaux 1
(spécialité informatique)

par

Pengfei Liu

Jury

Président :	Laurence Duchien,	Professeur Université Lille 1
Rapporteurs :	Laurence Duchien,	Professeur Université Lille 1
	Pierre-Etienne Moreau,	Professeur Ecole Nationale Supérieure des Mines de Nancy
Encadrants :	Charles Consel,	Professeur l'Institut Polytechnique de Bordeaux
	Hélène Kirchner,	Directrice de la recherche de l'INRIA

ABSTRACT

SPECIFYING AND ENFORCING SECURITY POLICIES VIA HIGH-LEVEL PROGRAMMING FRAMEWORK

When developing pervasive computing applications, it is critical to specify security policies and develop security mechanisms to ensure the confidentiality and integrity of the applications. Numerous policy specification languages only focus on their expressive power. The emerging challenges in pervasive computing systems can not be fulfilled by these approaches. For instance, context awareness is a central aspect of pervasive computing systems. Existing approaches rarely consider context information in their language.

This thesis proposes a generative approach dedicated to specifying and enforcing security policies in pervasive computing applications. To specify a policy, we propose a context-aware policy specification language which helps developers to specify policy rules and required entities (e.g. spatial description, roles, context information). Policies are implemented by term rewriting systems which offers great verification power. To enforce a policy, we propose an architecture that embeds important concepts of security policies (subject, object, security related context) into pervasive computing applications. To apply our approach, we enriched an existing approach which is dedicated to develop pervasive computing applications. Based on the policy specification and the enriched pervasive computing application descriptions, a dedicated programming framework is generated. This framework guides the implementation and raises the level of abstraction which can reduce the workloads of developers.

KEYWORDS: Domain-Specific Language, Generative Programming, Pervasive computing system, security policy

RÉSUMÉ

Lors du développement des applications ubiquitaires, il est essentiel de définir des politiques de sécurité et de développer des mécanismes de sécurité pour assurer la confidentialité et l'intégrité des applications. De nombreux langages de spécification de politiques se concentrent uniquement sur leur puissance d'expression. Les défis émergents dans les systèmes ubiquitaires ne peuvent pas être résolus par ces approches. Par exemple, la sensibilité au contexte est un élément central des systèmes ubiquitaires. Les approches existantes tiennent rarement compte des informations contextuelles dans leurs langages.

Cette thèse propose une approche générative pour spécifier et implanter les politiques de sécurité dans les applications ubiquitaires. Pour définir une politique de sécurité, nous proposons un langage de spécification qui tient compte des informations contextuelles. Il permet aux développeurs de spécifier les règles de la politique et les entités requises (e.g. la description spatiale, les rôles, le contexte). Les politiques sont implémentés par des systèmes de réécriture, ce qui offre une grande puissance de vérification. Pour appliquer une politique, nous proposons une architecture qui intègre les concepts importants des politiques de sécurité (sujet, contexte, objet) dans des applications ubiquitaires. Pour mettre en oeuvre notre approche, nous avons enrichi une approche existante pour le développement des applications ubiquitaires. La spécification de la politique de sécurité et la description de l'application ubiquitaire enrichie sont utilisées pour générer un canevas de programmation qui facilite l'implémentation des mécanismes de sécurité, tout en séparant les aspects sécurités de la logique applicative.

MOTS CLÉS: Architecture Logicielle, Langage Dédié, Systéme ubiquitaire, Politique de sécurité

ACKNOWLEDGMENTS

This thesis could never have been done without the help and support of many people.

First of all, I want to thank my thesis advisers Charles Consel and Hélène Kirchner. Despite the language barrier, they support me unconditionally throughout my thesis. I've never seen anyone like them who gives so much support for students.

The last year of my thesis was particularly difficult for Hélène, as she worked in Paris and I worked in Bordeaux, and we often met on Saturday mornings. She always encouraged me to go further and deeper.

I also want to thank my two reviewers Laurence Duchien and Pierre-Etienne Moreau. Thanks to their wise and careful reviews, I have a new understanding of my work.

I would like to thank Guan HongYu and his wife Guo ZuoSha for all their helps and supports. Especially after my achilles tendon repair surgery, they took care of me and helped me.

Thanks to all members of the Phoenix team, my thesis has been carried out in an ideal environment. Every member has contributed to the perfect team ambiance. I would like to thank Emilie Balland for her hints in writing articles and advice about my work, Damien Martin-Guillerez for giving me good advice about my presentation, Julien Bruneau for all the discussions which helped me to improve my work, Quentin for his friendship, Stephanie for her good humor, Henner for all the German words he taught me and the discussions about my work, Ghislain for explaining me the french jokes that I did not understand, Damien C., Julien M. and Benjamin for answering my questions with patience, Zoé for her sense of humor, Young-Joo for teaching me korean, and the newest members Milan, Camille, Luc and Paul. I want to thank everyone of you for the good time that we had together.

Many of my professors transmit their passions for research to me and introduce me to the world of research. I want to thank Alexis Bès, Ammel Mammar, Marie Duflot-Kremer, Frédéric Gervais and Gaétan Hains for their unconditional support when I was their student.

Finally, I would like to thank all members of my family, and my mother Zhou FengGe particularly who supports and loves me unconditionally.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

In the last ten years, hardware technology has continuously evolved, making the embedded computational devices smaller and cheaper. The advances in communication and sensor technology make pervasive computing applications more and more present in everyday life. The vision of pervasive computing systems was originally proposed by *Mark Weiser*, chief technology officer for Xerox's Palo Alto Research Centre. In his paper [95], he wrote : *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."*

The purpose of pervasive computing applications is to create a heterogeneous and device-rich computing environment which orchestrates all these devices and provides services to the user at anytime and anywhere. To develop such an environment, a great number of challenges need to be resolved [48, 83], such as mobility and heterogeneity of devices, coordination of different communication protocols of devices, interaction between user and devices, device discovery. A lot of approaches have been proposed (e.g. [25, 29, 36]) which make pervasive computing systems more and more mature.

Maturity of pervasive computing systems makes it applicable to an increasing number of areas, such as building automation, health care and assisted living. Some of those areas are critical areas, they have a critical effect on the physical world. For instance, doors need to be opened to evacuate people in case of fire. Rescuers need to be notified in case an elderly fell. Thus, the failure of such system can put people's life in danger.

To guarantee the pervasive computing system behaves as expected, its non-functional properties such as security (e.g. confidentiality, authenticity, etc. [16]), reliability (quality of service [20], fault-tolerance [28]) and efficiency need to be handled before its deployment.

The nature of pervasive computing environments increases the risks of security breaches. For instance, the data which are gathered by movement sensors can reveal the presence of home owners to housebreakers. Stalkers can use a camera in a pervasive computing environment to spy on their victims. To prevent all these scenarios from happening, we need to specify security policies to define who can or can not do what action on which devices, and provide a security-policy enforcement mechanism. This thesis focuses on the specification of security policies and the enforcement of security policies in pervasive computing systems.

In pervasive computing systems, a user interacts with pervasive computing environments through a variety of computing devices (e.g. smart-phone, PDA, tablet, RFID system) in the surrounding environments. In general, the pervasive computing systems consider those devices which represent a user as other common devices. So there is no proper definition for devices which can represent users. As a result, the notion of user needs to be explicit and associated with a range of privileges over resources. Yet, this notion must be pragmatic in that it should not burden the user with strenuous authentication procedures, but instead, leverage existing knowledge.

The set of devices which can provide services are extremely dynamic, since a great number of those devices are mobile. They can connect to or disconnect from the pervasive computing environment at any time. Still, the host pervasive computing environment needs to use a specific notation to encapsulate these devices with a security concept (i.e. object), because security policies may need to monitor these devices.

Context awareness [4, 63] is a central feature of pervasive computing applications that raises interesting challenges for security. For instance, in the context where the event "fire" is detected in the environment, the pervasive computing system has to trigger the fire alarm and doors must be unlocked for evacuation. However security policies in traditional software systems depend only on user privileges and are not context sensitive. In pervasive computing systems, such security policies can not take into account the current situation of the environment and may impede functionalities of pervasive computing system.

In pervasive computing system, security policies critically rely on spacial information to grant access to a given resource. This information needs to be introduced consistently throughout the pervasive computing environment [73]. The spacial information which are provided by sensors (e.g. GPS, infra-red, RFID tag) are not readable by users. To reduce the workload for specifying location-aware security policies, we need to refine these raw data with an abstract location description which is human readable.

Dirk Balfanz et al. [12] wrote : *" The security community has long argued that security must be designed into systems from the ground up; it can not be "bolted on" to an existing system at the last minute"*. A lot of experiences have shown that it is difficult to retrofit security into systems that have not been designed with security concerns in mind. So security concerns of pervasive computing systems need to be considered throughout the design, implementation, verification and test, deployment and maintenance stage [14]. But existing approaches focus on the expressive power of their security policy specification languages. None of them has addressed how to systematically integrate security policies into pervasive computing applications and to provide support throughout the application life-cycle.

The main requirements for developing security policies in pervasive computing systems are:

- Support for embedding the domain-specific notions (e.g. subject, object, security related context) to enforce security policies.
- Support for involving context in the security policies decision.
- Support for designing and enforcing security policies throughout the pervasive computing application life-cycle.

1.1 APPROACH

In this thesis, to facilitate the specification of security policies, we propose a domain specific language for security policies which is based on a rule-based formalism. Our language allows users to specify not only the basic concepts in security policies such as role hierarchy, but also to manipulate abstract location descriptions which are an important concept in pervasive computing systems. It also allows security policies to leverage security relevant context information (e.g. fire, intrusion).

To reduce the workload of the development of security policies in pervasive computing systems, we propose an architecture that embeds important concepts of security policies (e.g. subject, object) into pervasive computing applications. It separates the security policy management from the implementation of pervasive computing applications. Therefore, an administrator can dynamically modify the security policy without changing the implementation of the pervasive computing application or interrupt its services.

To develop our approach, we leverage DiaSuite [25], a tool based development methodology dedicated to the development of pervasive computing systems. The core of DiaSuite is DiaSpec which is a domain-specific design language for specifying pervasive computing applications. A generator named DiaGen is provided by DiaSuite, which can generate programming frameworks based on DiaSpec specifications. The generated programming frameworks guide developers throughout the development life-cycle of pervasive computing applications.

We enriched the description of DiaSpec with specific annotations. These annotations encapsulate basic DiaSpec entities (e.g. sensor, actuator and context) with more abstract notions. These annotations also indicate where to add new functionalities or modify existing ones in the generated programming framework.

To make the policy rules aware of the context related to the physical environment, policy rules must have constraints based on the context information which is provided by pervasive computing systems (e.g. professors can unlock all the doors if fire is detected). So we need a rule-based formalism which can reason over the rules with constraints to implement security policy specifications. In this thesis, the policy specification are compiled into a term rewriting system [10]. A term rewriting system is a collection of rewrite rules which are used to transform terms into equivalent forms. A term rewriting system offers good verification capabilities [19] and tools [13] which can interpret and reason about the term rewrite rules with constraints.

Our approach covers the pervasive computing applications life-cycle: 1) it enriches the description of pervasive computing applications with specific annotations; 2) it enables security policies at the design stage; 3) it guides developers by generating programming framework at the implementation stage; 4) it provides verification and testing tool at the verification stage;

The expressiveness of our language is limited. Our policy language can not express delegation policies or obligation policies. Because, there are already many languages which can express these kinds of policies, our approach focuses on how to facilitate the specification and enforcement of security policies. In this thesis, we only apply our approach on a specific domain of pervasive computing systems (i.e. smart home). However, the solutions which we proposed should be general enough to support other pervasive computing systems.

1.2 THESIS CONTRIBUTION

In this work, we propose a security-policy specification language and a security-policy enforcement mechanism that respectively leverage term rewriting systems [10] and a tool-based development methodology dedicated to pervasive computing systems, named *DiaSuite* [25]. Our contributions can be summarized as followed.

Context awareness. We introduce security-relevant context information in the design of a pervasive computing system by enriching the description of pervasive computing applications. Our policy specification language DiaSecur allows security policy rules to use the security relevant context information as activation condition. These constraints make security policies context sensitive.

Security throughout the application life-cycle. We have identified the requirements to specify and enforce security policies in pervasive computing applications at different stages of the application life-cycle. By enriching the description of pervasive computing applications, the enforcement of security policies has been automatically, systematically and seamlessly integrated into the applications life-cycle.

Security-policy enforcement mechanism. We propose a security-policy enforcement mechanism to fulfil the special requirements of pervasive computing applications. The security-policy enforcement mechanism embeds security policies automatically into all devices in pervasive computing application which improves the scalability. To support the dynamicity of pervasive computing applications, we separate the policy from the implementation of pervasive computing application. As a result, the modification of pervasive computing application do not impact the security policy.

Experiments in the pervasive computing domain. Our approach has been applied to the development of various pervasive computing applications, including a physical access control system of a research facility and a parental control system. These experiments have demonstrated that our approach can effectively guide the development of pervasive computing applications with security concerns addressed during the entire process.

1.3 ROADMAP

This thesis is split into three parts: the first part proposes a state of the art that introduces the context and the challenges of specifying and enforcing security policies in pervasive computing systems; the second part presents our approach that consists of a security-policy specification language and a security-policy enforcement mechanism; and the third part concludes the thesis and outlines future work.

Context

In Chapter 2, we present the related work and the context of my thesis. In Section 2.1, we first introduce the basic terminologies and concepts of pervasive computing systems. Then we present the application domains and requirements of pervasive computing system. In Section 2.2, we present several existing approaches which are dedicated to develop pervasive computing applications. At the end of this section, we highlight an approach, named DiaSpec which we enriched to apply our approach. In Section 2.3, we first give an overview of security policies. Then we present several existing approaches for specifying a security policy. Lastly, we present several approaches for enforcing a security policy in a pervasive computing system.

In Chapter 3, we give a scenario of a pervasive computing application. In Section 3.1, we describe the required devices and the physical environment of this pervasive computing application. In Section 3.2, we present the security requirements of this scenario.

Embedding security policies in pervasive computing systems

The second part presents our domain specific policy specification language (DiaSecur) and our policy enforcement mechanism. We illustrate our approach step by step from the design stage to the deployment stage.

In Chapter 4, we give an overview of our approach. We present the key concepts and requirements which we have identified, and solutions that we propose to address these requirements at each stage of the application life-cycle.

In Chapter 5, we present our policy specification language (DiaSecur) dedicated to specify authorization policies in pervasive computing systems. In Section 5.1, we present the domain analysis which identifies the basic components of our language. In Section 5.2, we present the basic entities of our language.

In Chapter 6, we present the support which we provide at the design stage. In Section 6.1, we present the architecture of our policy enforcement mechanism. In Section 6.2, we present how we use the enriched DiaSpec specification to develop pervasive computing applications with security policies enforcement. In Section 6.3, we use the applications which we introduced in the case study (Chapter 3) to illustrate how to design a concrete secured pervasive computing applications by using our approach.

In Chapter 7, we present how to implement the secured pervasive computing system and the security management. In Section 7.1, we present how we enrich the generated programming framework to embed the security enforcement mechanism into pervasive computing applications. In Section 7.2, we present the basic entities of the security management. In Section 7.3, we present how to implement the applications in the case study based on the design specification which we described in Section 6.3.

In Chapter 8, we first present how to verify and test security policies in a pervasive computing environment. Then, we present how we provide support at deployment and maintenance stage. In Section 8.1, we first present several properties of the security policies which we want to verify. Then we present how to verify these properties. In Section 8.2, we give a brief introduction about the simulator which we use to test our security policies. In Section 8.3, we present how our approach supports the change of secured pervasive computing systems and security policies.

Conclusion

In chapter 9, we present the conclusion of this thesis. In chapter 10, we suggest directions for future work.

Part I

CONTEXT

2 BACKGROUND

The first part of this chapter introduces pervasive computing systems, their application domains and requirements. The second part of this chapter presents the existing approaches dedicated to develop pervasive computing systems. We highlight the DiaSuite approach which is used later on. In the third part of this chapter, we first give an overview of security systems, then we present the existing approaches of security-policy specification language and we highlight a rule-based formalism to implement our policy engine. Lastly, we present some approaches which are dedicated to enforce security policies.

2.1 PERVASIVE COMPUTING SYSTEMS

Pervasive computing

In the article "The computer of 21st century" [95], Mark Weiser shaped the vision of pervasive computing (also called ubiquitous computing) as an omnipresent infrastructure for information and communication technologies. The purpose of pervasive computing applications is to create a device-rich computing environment which orchestrates devices and provides services to users at anytime and anywhere. According to Weiser [96, 97], to meet the claim of "everything, always, everywhere", the following conditions need to be met:

Smart objects. Computational and communication components are integrated into physical objects of any shape and offer services (e.g. air-conditioner, light). The embedded computational and communication components enrich the objects with a new array of digital capacities. Those digital capacities make the physical objects "smart".

Networking. Smart objects are connected to each other and can communicate via wired or wireless connections. The availability of services relies on the communication between devices and applications, not on the devices themselves. This point is what distinguishes pervasive computing from mobile networks.

Transparency. Sensors read signs and gestures of users and collect environment information. Based on the collected information, the pervasive computing system readjusts its behaviour automatically.

Several years after the birth of pervasive computing, a novel feature *context* has been proposed. Context information can be any information (e.g. time, location, temperature, etc.) which describes the physical world that influences the pervasive computing applications.

Ambient intelligence

The term "Ambient intelligence" was originally proposed by the European Commission in 2001. Sadri Fariba [78] describes the vision of ambient intelligence as an environment which is sensitive to the needs of its inhabitants, and capable of anticipating their needs and behavior. The environment is aware of their personal requirements and preferences, and interacts with people in a user-friendly way. Many people [2, 3, 86] consider that the ambient intelligence paradigm is built upon pervasive computing, context awareness and human-centric computer interaction design, and that ambient intelligence is characterized by systems and technologies that are:

Embedded. Many networked devices are integrated into the environment

Context aware. These devices can recognize a user and his situational context

Personalized. They can be tailored to fulfil the needs of inhabitants.

Adaptive. They can be easily reconfigured in response to users desires.

Anticipatory. They can anticipate the desires of users without conscious mediation.

2.1.1 *Application area of pervasive computing systems*

Pervasive computing systems have potential applications in many domains such as smart home, assisted living and health care. Some of them are already realized, and others look more like science fiction. In this thesis, we focus on the domain of smart homes and professional buildings.

2.1.1.1 *Smart home*

Keith Edwards and Rebecca Grinter [37] define smart home (also called aware home) as *"domestic environments in which we are sur-*

rounded by interconnected technologies that are, more or less, responsive to our presence and actions".

Aldrich [8] sees a smart home as*"a residence equipped with computing and information technology, which anticipates and responds to the needs of the occupants, working to promote their comfort, convenience, security, and entertainment through the management of technology within the home and connections to the world beyond"*.

Figure 1 shows a typical example of smart home environment and the common resources which are usually used in smart home environment. The internet gateway plays an important role in this environment. In the first place, it is responsible for connecting all the deployed devices within the smart home environment. Secondly, it is responsible for connecting the smart home environment with the internet. By connecting with the internet, the gateway can integrate web services into applications within the smart home environment (e.g. weather forecasts). The gateway is the ideal candidate to host the runtime environment for applications.

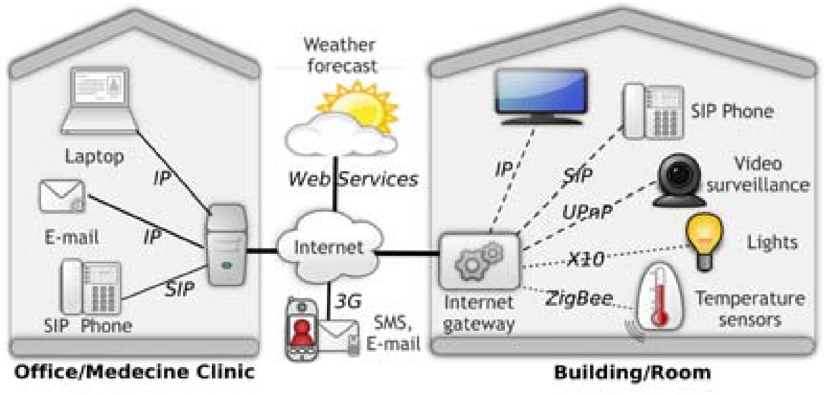


Figure 1: Smart home and its resources

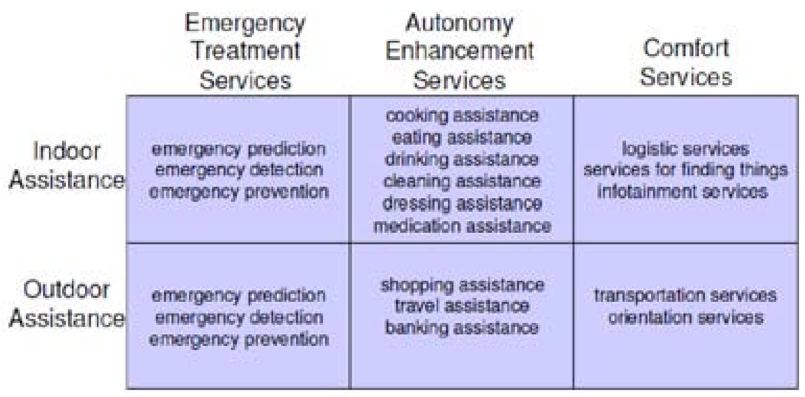
According to Sadri Fariba [78], tasks envisaged in a smart home include the following:

- performing many everyday tasks automatically to reduce the burden of managing the house. For example, open window blinds when inhabitants wake up.
- improving economy of usage of utilities. For instance, save energy by turning off the lights when no one is in the house.
- improving safety and security. For example, activate sprinklers when a fire is detected.
- improving quality of life. For instance, play a specific music based on the mood of inhabitants.

- supporting independent living for people with some cognitive impairment. For example, remind a user to take his pills.

2.1.1.2 *Assisted living*

Due to the increasing number of elderly people, the domain of assisted living is attracting much attention. Assisted living environments provide services ranging from emergency detection to cooking assistance, in order to improve the quality of life of elderly people living independently in their homes. Compared to smart homes, assisted living systems need to provide not only the indoor assistance, but also outdoor assistance when users are doing some outdoor activities. Figure 2 shows assisted-living services classification which is proposed by *Jurgen Nehmer et al.* [65].



	Emergency Treatment Services	Autonomy Enhancement Services	Comfort Services
Indoor Assistance	emergency prediction; emergency detection; emergency prevention	cooking assistance; eating assistance; drinking assistance; cleaning assistance; dressing assistance; medication assistance	logistic services; services for finding things; infotainment services
Outdoor Assistance	emergency prediction; emergency detection; emergency prevention	shopping assistance; travel assistance; banking assistance	transportation services; orientation services

Figure 2: Classification Scheme for the assisted living domain

2.1.1.3 *Health care*

In the health care domain, pervasive computing systems are usually used to gather biological and physiological data of an individual and offers supports to enhance environments for diagnosis, therapy, prevention and early detection of diseases.

Reinhold Haux [46] describes pervasive computing health-enabling technologies as *"technologies include wearable devices, such as microsensors embedded in textiles ans personal computers. These technologies are aimed at making it easier for individuals to monitor and maintain their own health while enjoying lives in normal social settings."*

2.1.2 *Challenges in smart home systems*

The notions of smart home has been proposed since 1970. With the advance of pervasive computing technologies, we could already develop various applications to realize certain functionalities of smart homes. But some of the social and technical problems [37, 48] remain to be addressed.

Based on the work of Edwards et al. [37] and Karen et al. [48], we could divide the various challenges to successfully develop a smart home into two main categories: functional challenges and non-functional challenges.

Functional challenges

The various technologies involved in smart home systems make the design and development of applications complicated.

1. *Dynamicity.*

   Smart objects (computational devices) are the core of smart homes. Computational devices may become available as they get deployed in smart homes. They may become unavailable due to malfunction or network failure. Smart homes need to support the dynamicity of devices which can appear and disappear at any time.

2. *Heterogeneity and Interoperability*

   Smart objects run on specific platforms, feature various interaction models, and provide non-standard interfaces. This raises the question of how a smart home orchestrates all these devices. The devices may use new technologies unsupported by the smart home. This raises the question of how the smart home integrates the new technologies seamlessly.

3. *Computational devices management*

   In a smart home, computational devices need to be managed. Users need to be informed what devices the smart home possesses, what services they provide, what are their status, and how they can be used.

4. *Feedback to user*

   Smart homes collect information of users and their surrounding environment and make decision proactively. Some of those decisions may not serve user needs. The users need to understand how and why the smart homes make such decisions in a given situation.

5. *No system administrator*

   All the new devices which users want to deploy at their smart homes have to be installed and configured in some way. Tasks that were performed by professionals a few years back will now need to be done by the end users.

Non-Functional challenges

When pervasive computing systems are applied to the domain of health care or assisted living for elderly people, the well functioning of pervasive computing systems is critical. To guarantee the well functioning of pervasive computing system after deployment, the non-functional properties such as dependability and security need to be addressed.

DEPENDABILITY *Avizienis et al.* [9] define dependability of a system as *"the ability to avoid service failures that are more frequent and more severe than is acceptable"*. They believe that the dependability of a system consists of the following attributes:

- *availability:* the services which are provided by pervasive computing systems need to be ready when users need them.
- *reliability:* the services in pervasive computing systems need to continue to work even when there is a device failure.
- *safety:* services which are provided by pervasive computing systems will never lead to catastrophic consequences on the users and the environment.
- *integrity:* absence of improper system alteration.
- *maintainability:* ability for a process to undergo modification and repairs.

SECURITY The goal of security is also to guarantee the well functioning of pervasive computing systems. But form another point of view, security ensures the availability, integrity and confidentiality of pervasive computing systems in the presence of possible threats. Compared to the dependability of systems, security requires more restrictions. For instance, in security the availability means that the services which are provided by systems are only available for authorized users, the integrity of data means that only authorized users can modify the data. In Section 2.3, we will discuss security in details.

2.2 DEVELOPING PERVASIVE COMPUTING APPLICATIONS

We have seen the various challenges to develop pervasive computing applications. Many approaches have been proposed to overcome these challenges. As pervasive computing systems orchestrate various devices, interconnecting and running applications across a set of devices is the main challenge. Many of the early works in pervasive computing focused on interconnecting and managing a large set of devices. Many middleware [45] approaches have been proposed such as iROS (Interactive Room Operating System) [51] and Gaia [76].

As pervasive computing systems require the transparency of devices, research efforts have been devoted to simplify the interaction between users and smart environments. Tools such as the *Context Toolkit* [79] have been proposed to provide contextual information about the environment surrounding users.

But people start to realize that those approaches focus on only few aspects of pervasive computing systems. For instance, the middleware approaches address only the networking aspect, Approaches like Context Toolkit address only the transparency aspect. We need more sophisticated approach which can cover all the aspects of pervasive computing systems. In this section, we present the existing approaches which help developers to address the functionalities of pervasive computing applications.

2.2.1 *Vigil*

In 2001, Vigil [55, 56] was proposed by a research group of university of Maryland. *Lalana Kagal et al.* use an agent-oriented paradigm to model the interactions between computational entities. Vigil is developed based on the work of Centaurus [53, 54], which is an infrastructure and communication protocol for interconnecting heterogeneous computational entities.

A Vigil system consists of six functional components within the Vigil architecture.:

- *Communication manager.* It provides a communication gateway between a *client* and a *Service Manager*.
- *Capability manager.* It maintains a static capability matrix for system entities and responds to initial requests for access control.
- *Service manager.* It finds out the matching services which users request.

- *Certificate manager.* It generates X.509 digital certificates for each system entity and responds to certificate validation queries.
- *Security agent.* It interprets and enforces the security policy to provide access control services.
- *Role Assignment Manager.* It is responsible for the assignment of roles to entities.

Vigil is one of the few approaches who has a strong focus on security. The *certificate controller*, *security agent* and *role assignment manager* are specially designed to address security concerns. But how these components collaborate together is not clear. In fact, Vigil was built from scratch to easily integrate security concerns. The authors do not focus on how to support developers for designing and implementing pervasive computing applications.

2.2.2 *Aura*

In 2002, the aura approach was proposed. The goal of Aura project [43, 89] is to maximize the use of available resources, minimize user distraction and drain users attention. To achieve these goals, the concept of *personal aura* has been proposed. The personal aura acts as a proxy for the user which the Aura represents. When a user enters a new environment, his personal aura discovers appropriate resources to support the user's tasks.

To enable the functionalities of such a personal Aura. An architectural framework has been proposed. Figure 3 shows a bird's-eye view of the architectural framework of Aura.

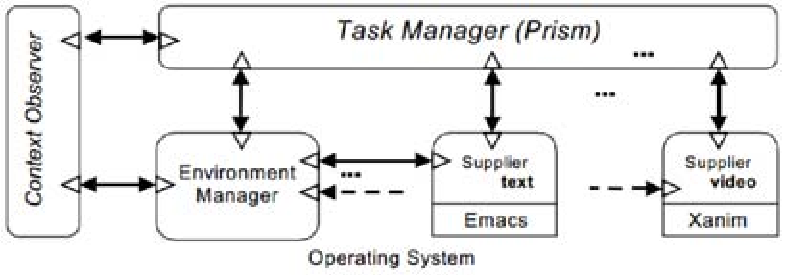


Figure 3: The Aura bird's-eye view

Aura consists of four basic component: task manager (called Prism), context observer, environment manager and suppliers.

- *Task manager.* It implements the concept of personal aura. It can coordinate the information related to the user task and negotiates the task support with the new environment. Task manager monitors quality of service of *suppliers*. For instance, if a supplier fails, the task manager will find an alternative supplier.

- *Context observer.* It provides information on the physical world and reports the useful information to the *Task manager*.
- *Environment manager.* It embodies the gateway to the environment which can marshal various *suppliers*.
- *Suppliers.* They provide the abstract services which tasks are composed of. These abstract services can be implemented by concrete application such as Emacs or Notepad.

The personal aura concept facilitates the designing of user-centric applications, because the personal aura manages which actual services will be used at each pervasive computing environment. But, this raises the problem of mapping the actual services with the required services of personal aura. To accomplish the mapping, all actual services need a wrapper and an abstract service description. Aura does not provide enough support for implementing services. These increase the workload of upgrading and maintaining the system.

2.2.3 *MavHome*

In 2004, the MavHome (Managing An Intelligent Versatile Home) approach was proposed. The objective of MavHome project is to create a home full of devices which can acquire and apply information about the inhabitants to provide comfort and efficiency [32, 100].

The MavHome system consists of a group of agents; each agent contains fours layers: physical layer, communication layer, information layer and decision layer. All these layers are realized through a set of concrete functional components. Figure 4 shows the functional components of each layer [32].

- *Physical layer.* The physical layer contains all physical hardware such as devices and network equipments. This layer collects the raw data which are provided by the physical devices (e.g. camera, temperature sensor, etc.).
- *Communication layer.* The communication layer can not only communicate with all layers within the agent, but also with external layers of other agents. The communication layer discovers services from physical layers and transfers data to information layers. In the communication layer, the Common Object Request Broker Architecture (CORBA) [22] is used to perform the communications.

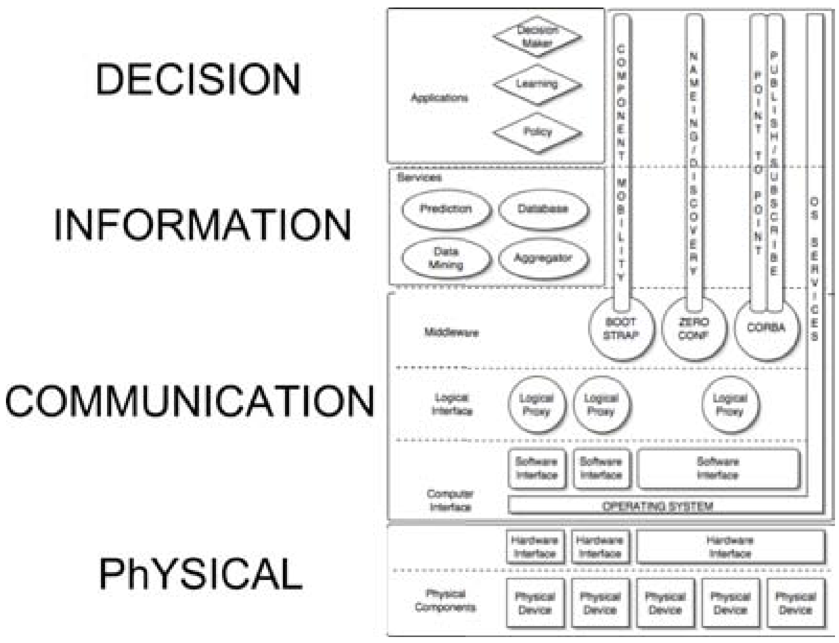


Figure 4: The MavHome agent architecture

- *Information layer.* The information layer gathers and stores the data collected by the physical layers. Based on these data, the information layer generates knowledge useful[1] for decision making.
- *Decision layer.* The decision layer takes in useful knowledge, learns from stored knowledge, makes decisions on actions to automate the everyday tasks in the environment, and develops policies while checking for safety and security.

The MavHome system provides a clear architecture to develop pervasive computing applications. But it does not provide any support at the design stage. It makes the applications hard to extend and maintain.

2.2.4 *Olympus*

Roy Campbell et al. propose a new concept of *Active Spaces* [72]. The authors define active spaces as spaces with computing and communication devices that provide support to users. They have developed a middleware, named Gaia [76] for developing active spaces.

Gaia [29] provides an authentication service which includes various protocols (e.g. Kerberos [2]) and hardware technologies (e.g. fingerprint). Sampemane et al. [80] propose an access control

1 We could consider those useful knowledge as contextual information
2 http://web.mit.edu/kerberos/

model dedicated to *active spaces* [72]. They use an access list to specify access rights for each service in the active space. But, Gaia offers no support for creating coherent security policies which are dedicated to the deployed applications and resources. As a result, it is error prone for enforcing the security policies.

To provide more guidance to developers, they proposed Olympus [72] in 2005. The main feature of Olympus is that developers can specify computational entities, services, locations and users at an abstract level. The high-level programming model allows developers to develop active spaces without worrying about how space operations are implemented and which concrete devices will be used. To realize such functionalities, their model provides an entity discovery component. The entity discovery component can map the abstract entities which are specified by developers with the concrete entities (e.g. light, air-conditioner, etc.) based on the current context, space configuration and user preferences.

The Olympus offers an abstract way to describe active space operations. But it does not provide enough support at the design stage. It is hard to extend the space operations.

2.2.5 *PervML*

In 2009, PervML [85] was developed by a research group of the Technical University of Valencia. To facilitate the development of a pervasive computing system, they propose a model-driven development method, which covers the entire development lifecycle. PervML is a domain-specific modelling language that allows developers to represent pervasive computing systems in an abstract model (PervML model).

PervML specifies a context-aware pervasive computing system in a way that is independent from a platform and a technology. To do so, PervML uses conceptual primitives such as *users*, *services*, *interactions* and *component structure*. PervML isolates these conceptual primitives and represents each primitive with a dedicated model. For instance, a UML-like class diagram is used to specify services and a UML package diagram is used to describe the different areas where the user can move or where services can be located.

PervML provides tools that can generate the dedicated framework in Java and OWL (Ontology Web Language [94]) based on the PervML models. Then the generated applications can be deployed by using OSGi (Open Service Gateway initiative [34]).

PervML has no clear notion of security-policy. But it provides a *User Profile Model* which allows developers to specify the types of user (profiles). The administrators can associate a list of authorized operations to each profile. According to the authors, this model *"provides support for the privacy, the security and the views of the system, since users can see and execute only system actions that they are authorized to use"*.

PervML represents each primitive with a dedicated model. For instance, an application requires six different models: 1) a UML-like class diagram, 2) a UML package diagram to describe the different areas, 3) a graphical state transition diagram, 4) a UML interaction diagram, 5) a OCL description, 6) a user profile model. As a result, developers need to master these technologies. All these models make the PervML hard to use and master, compared to other approaches.

2.2.6 *DiaSuite*

In this section, we highlight DiaSuite [25], a tool based development methodology dedicated to the development of pervasive computing systems. In this thesis, we use this approach to illustrate how to embed security policy into pervasive computing systems, which justifies why we introduce this approach with more details.

The core of DiaSuite is DiaSpec which is a domain-specific design language for specifying pervasive computing applications. A generator named DiaGen is provided by DiaSuite, which can generate programming frameworks based on DiaSpec specifications. The generated programming frameworks guide developers throughout the development life-cycle of pervasive computing applications.

The sense-compute-control pattern

The DiaSpec language is inspired by an architecture pattern, named sense-compute-control (SCC) [90]. The SCC pattern is often applied to the applications which interact with an environment. The environment could contain physical devices, services which are provided by software (e.g. web services, data base, etc.). The SCC pattern embodies four layers: sensing layer, refinement layer, control layer, action layer.

The DiaSpec language uses three kinds of components (i. e. devices, contexts and controllers) to cover the SCC pattern. Figure 5 gives a graphic representation of the DiaSpec architecture which represents the SCC pattern. *Devices* sense the information about pervasive computing environment and send the raw data to context. Devices also provide actions that can be executed by controllers. *Contexts* collect the raw data from devices and refine them to produce more abstract data. *Controllers* make decisions to activate devices based on the information received from contexts.

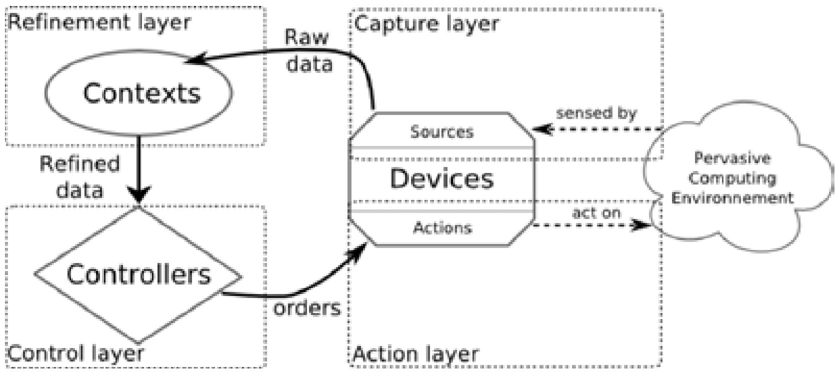


Figure 5: The DiaSpec architecture with SCC pattern

Overview of the DiaSuite approach

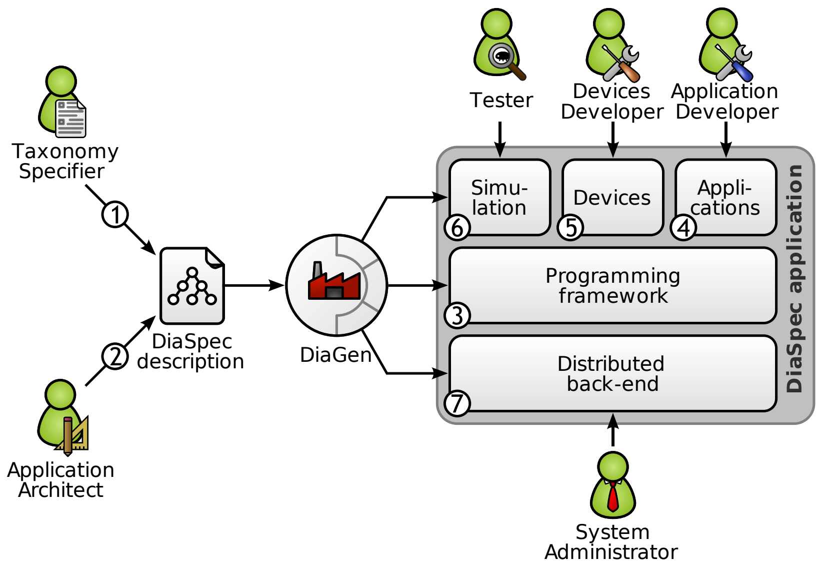


Figure 6: The DiaSuite development cycle

The DiaSuite approach proposes a development process underlying DiaSpec. This process revolves around key stages and roles depicted in Figure 6. At the *design stage*, a domain expert defines a taxonomy of devices which are available in the pervasive computing environment and actions that the devices provide. An architect describes the architecture of pervasive computing application based on the taxonomy definition. At *development stage*, DiaSuite provides a compiler, named DiaGen. DiaGen can generate a customized programming framework based on a taxonomy and the architecture description of an application. Then the application developers use the generated framework to implement the application. Device developers instantiate the abstract entities with actual devices. At the *test stage*, DiaSuite provides a 2D-simulator, named DiaSim [23]. It produces a hybrid simulation environment which contains simulated devices and actual devices. At the *deployment stage*, the system administrator chooses a distributed systems technology such as Web Services [15], RMI [71], CORBA [68] and SIP [77] to deploy the pervasive computing system.

In the next two paragraphs, we focus on the design stage and development stage of the development process.

Designing a pervasive computing application with DiaSpec

Figure 7 shows an architecture of a fire emergency management application. The arrows indicate the flow of data. To detect a fire, the application uses two kinds of devices: a temperature sensor and a smoke sensor. When a fire is detected, the alarm and sprinkler will be turned on. This application is a typical context-aware application in a smart home environment. Based on the context information, the application makes decisions for users autonomously.

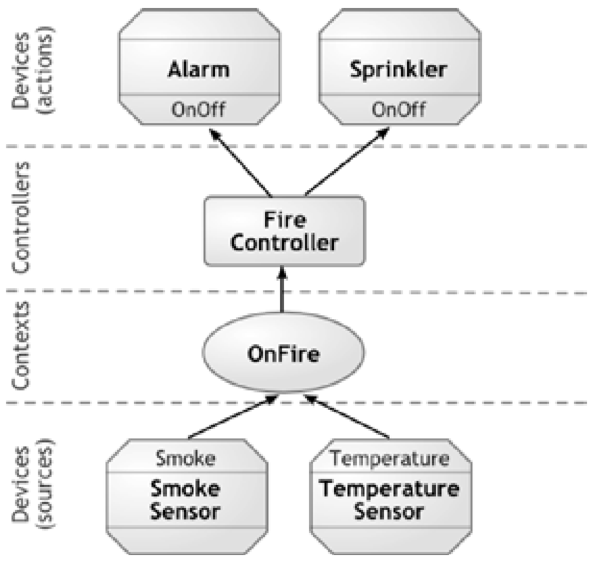


Figure 7: The graphical representation of the fire management architecture

To describe the available resources (software or hardware) of a pervasive computing environment, DiaSuite provides a domain specific language DiaSpec. The domain expert uses DiaSpec to write a taxonomy. A taxonomy is a collection of device declarations, each of which characterizes a set of possible device implementations sharing common functionalities. Device functionalities consist of data sources and actions. A data source specifies values sensed by a device. An action declares a set of operations supported by a device. To distinguish each instance of a class of devices, the device declarations may have one or more attributes such as identity and location. Device declarations are organized hierarchically, allowing devices to inherit attributes, sources and actions from other devices.

Listing 1 shows a fragment of the taxonomy of the fire emergency management application. The entity description starts with keyword **device**. The first device description defines a root device (lines 2 to 4 of Listing 1), which encapsulates the basic attributes of an entity such as identity and location. These attributes serve mainly as filters for entity discovery in the pervasive computing environment.

DiaSpec uses the keywords **source** and **action** to define the capabilities of a device. The keyword **source** defines the sensing capabilities of an entity. The keyword **action** defines the actuating capabilities of an entity. For instance, line 7 of listing 1 defines that the device `TemperatureSensor` provides a float value which describes the temperature of the pervasive computing environment. Line 15 of listing 1 defines that the device `sprinkler` provides the action `OnOff` which is defined further in lines 19 to 20.

```
1  device Devices{
2     attribute identity as String;
3     attribute location as Location;
4  }
5  
6  device TemperatureSensor extends Devices{
7     source temperature as Float;
8  }
9  
10 device SmokeSensor extends Devices{
11    source smoke as Boolean;
12 }
13 
14 device Sprinkler extends Devices{
15    action OnOff;
16 }
17 
18 device Alarm extends Devices{
19    action OnOff;
20 }
21 /* Description of the supported actions*/
22 action OnOff {
23     On();
24     Off();
25 }
```

Listing 1: Extract of the taxonomy of the fire detection application

To design the architecture of the pervasive computing applications, the DiaSpec language provides two building blocks: context and controller. The context can refine the raw data collected from deployed devices and provide more abstract context information. The controller can make a decision to execute actions based on the context information. Listing 2 shows a fragment of the architecture description of the fire detection application.

Lines 1 to 3 of listing 2 defines a context. It starts with keyword **context**, and followed by the name (i.e. OnFire) of this context. The keyword **as** followed by the type boolean indicates the type of the output value. The keyword **source** followed by `temperature` indicates the name of the input data (line 2). The keyword **from** followed by `TemperatureSensor` indicates which kind of devices provides the input data (line 2). For instance, the context `OnFire` collects temperature from temperature sensors and smoke from smoke sensors (lines 1 to 3).

Lines 6 to 9 of listing 2 defines a controller. It starts with keyword **controller**, and followed by a controller name (e.g. FireController). The keyword **context** followed by a context name indicates the source of input context data (line 7). The keyword **action** indicates which specific actuators can be invoked by the controller. For instance, the controller `FireController` receives contextual information fire from context `OnFire`, and can turn on and off `Alarm` and `Sprinkler` (lines 7 to 9).

```
1  context OnFire as boolean{
2  source temperature from TemperatureSensor;
3  source smoke from SmokeSensor;
4  }
5
6  controller FireController{
7  context OnFire;
8  action OnOff on Alarm;
9  action OnOff on Sprinkler;
10 }
```

Listing 2: Extract of the architecture description of the fire detection application

Implementing a pervasive computing application

To complete the development, DiaSuite provides a compiler, named DiaGen. DiaGen is implemented by using the ANTLR parser generator [70]. Based on the taxonomy and architecture description, DiaGen generates a dedicated Java programming framework and a distributed backend which encapsulates various communication technologies, and manages the component registration and discovery. The backend abstracts over the communication layer, allowing to transparently deploy the application implementation.

Given a DiaSpec design declaration, the DiaGen compiler generates dedicated code to implement each basic building blocks. For instance, each action defined in a taxonomy generates a Java interface. Listing 3 shows an example of the generated Java interface for the action `OnOff` which was declared in Listing 1.

```java
public interface OnOff {
  void On(RmiRemoteServiceInfo source) throws RemoteException;
  void Off(RmiRemoteServiceInfo source) throws RemoteException;
}
```

Listing 3: The generated Java interface for the OnOff action which was declared in Listing 1

DiaGen produces an abstract class for each device, context and controller declaration, providing methods to support the development (discovery and interactions). It also generates abstract method declarations to allow the developer to write the code that implements the expected behaviour of the application (e.g. a controller activates the fire alarm when a fire is detected). Implementing a DiaSpec-declared entity is done by subclassing the corresponding generated abstract class. In doing so, the developer is required to implement each abstract method.

Let us use an example to illustrate the implementation of the devices. Listing 4 shows the generated abstract class of the device declaration `Alarm` in Listing 1. The abstract class has two abstract methods `On` and `Off` that the developers need to implement.

```java
public abstract class AbstractAlarm{
    protected void updateIdentity(String identity) {...}
    protected void updateLocation(Location location) {...}
    public abstract void on();
    public abstract void off();
    ...
}
```

Listing 4: Extract of the abstract class of Alarm

Listing 5 shows a code fragment that wraps a alarm device that is controlled by using X10 [1], commonly used in smart home platform.

```java
public class Alarm_X10 extends AbstractAlarm{

   ... // defines x10Ctrl and x10Addr

@Override
public void on() {
   x10Ctrl.addCommand(new Command(x10Addr, Command.ON));
   }
@Override
public void off() {
   x10Ctrl.addCommand(new Command(x10Addr, Command.OFF));
   }
}
```

Listing 5: A developer-supplied implementation of an alarm device using the X10 protocol in Java

Based on the `OnFire` context declaration of Listing 2, DiaGen generates an abstract class to support developers. The abstract class allows distributed devices to be selected through service discovery. To implement the context processing logic (i.e. how the context refines the raw data which are provided by the sensors), we need to extend the generated abstract class of the context `OnFire`. The code fragment in Listing 6 presents the implementation of the `OnFire` context declaration in Listing 2. This is done by extending the corresponding generated abstract class `AbstractOnFire` (We do not show the code of the abstract class here, it is similar to the one in Listing 4).

The `OnFire` context is declared as taking a `smoke` input source from smoke sensors (line 13 of Listing 2). We highlight the `onSomkeFromSmokeSensor` method in Listing 6. This method is triggered when the smoke sensors sense fresh data. In this implementation, we first check the smoke value (line 20 of Listing 6), then we check the temperature of the place where the smoke is detected (lines 22 to 27). If the temperature is higher than fifty degrees celsius, the context OnFire publishes a new boolean value to indicate the presence of fire at this specific location (line 30).

```java
1
2  public class OnFireContext extends AbstractOnFire {
3        @Override
4     public void postInitialize() {
5         for (SmokeSensorProxy smokeSensors :
              this.discoverSmokeSensorForSubscribe
6                 .all()) {
7             smokeSensors.subscribeSmoke();
8         }
9
10    }
11    @Override
12    public FireContextIndexedValuePublishable
          onSmokeFromSmokeSensor(
13            SmokeFromSmokeSensor smoke,
14            GetContextForSmokeFromSmokeSensor getContext,
15            DiscoverForSmokeFromSmokeSensor discover) {
16      //Tests whether smoke is detected
17     if(smoke.value==true){
18        //Gets the location of where the smoke is detected
19         String location = smoke.indices().location();
20         TempSensorCompositeForSmokeFromSmokeSensor temp =
         discover.tempSensors().whereLocation(location);
          Boolean fire;
          //Checks whether the temperature is abnormal
           if(temp.anyOne().getTemperature()>50)
              fire=true;
           else fire=false;
        return new FireContextIndexedValuePublishable(fire,
            location, true);
      }
     else return null;
    }

     }
```

Listing 6: A developer-supplied implementation of the OnFire context in Java

To realize the functionality of the application, the application logic needs to be implemented in the generated abstract classes, which represent context and controller declarations. Listing 7 shows a code fragment that implements the abstract class of `FireController`. The abstract method `onFireContext` has three parameters: `FireContextValue` provides the data which is computed by the context `OnFire`. `GetContextForFireContext` contains a set of proxies of other context components which connects to this controller. `DiscoverForFireContext` contains a set of proxies that allows the `FireController` to discover the instances of the deployed devices. In this example, we use `DiscoverForFireContext` to discover alarms and sprinklers (lines 14 to 16).

To implement the application logic, first, we retrieve the boolean value which indicates if a fire has been detected (line 10). Then we get the location of fire (line 11). Lastly, if there is a fire, we activate all the alarms (line 14) and the sprinklers located at the fire zone (lines 16 to 20).

```
1  public class MyFireController extends AbstractFireController {
2
3      public MyFireController(ServiceConfiguration
           serviceConfiguration) {
4          super(serviceConfiguration);
5      }
6
7    @Override
8    public void onFireContext(FireContextValue fireContext,
9          GetContextForFireContext getContext,
              DiscoverForFireContext discover) {
10     Boolean fire = fireContext.value();
11     String location = fireContext.indices().location();
12     if (fire) {
13         //Discover all alarms and turn them on
14         discover.alarms().all().on();
15         /*Discover all sprinklers and turn the sprinklers which
              located at the fire zone on*/
```

```
        SprinklerCompositeForFireContext sprinklers=
            discover.sprinklers().all();
        for (SprinklerProxyForFireContext sprinkler : sprinklers) {
            if (sprinkler.location().equals(location)) {
                sprinkler.on();
                }
            }}
    else { /*If there is no fire detected, do nothing.*/ }
            }
        }}
    }
```

Listing 7: A developer-supplied implementation of the FireController

We choose DiaSpec to apply our approach, because it provides strong support at the design stage. It allows us to address security concerns at the design stage. For instance, the taxonomy in DiaSpec which describes the available devices facilitates the enforcement of security policies. The architecture description provides context information which may be used by security policy. The generated framework allows us to embed security mechanism automatically.

2.3 SECURITY OF PERVASIVE COMPUTING SYSTEM

The National Security Telecommunications and Information Systems Security Committee (NSTISSC) of United States defines computer security as *"Measures and controls that ensure confidentiality, integrity, and availability of the information processed and stored by a computer"*. A lot of similar definitions for computer security have been proposed by other scientists [17]. Confidentiality of information refers to preventing the information leakage. Integrity of information refers to preventing the modification of information. Availability refers to preventing the denial of access of information.

Authentication, access control and access audit are the basic technologies to ensure confidentiality, integrity and availability. Authentication helps access control system to distinguish legitimate users from malicious users. An access control system restricts the accesses to system resources to legitimate users. An access control system consists of a security policy and a security mechanism to enforce this policy. Access audit verifies and guarantees the well functioning of access control system.

2.3.1 *Access control models*

Access control has been intensively studied for decades. Several access control models have been identified. Security policies were originally divided into discretionary policies and mandatory policies, based on who is allowed to define the access of users to the information. With the development of computer technology, some systems need to allow owners of the objects to modify the access rights, while the access rights of some specific objects can only be modified by the administrator. As a result, the discretionary and mandatory policies can no longer fulfil the new requirements. Therefore, new security policy models have been proposed. Let us examine the main models.

2.3.1.1 *Discretionary policy*

Discretionary Access Control (DAC) [67] is defined by the United States Department of Defence. In DAC, subjects are capable to pass the permission on the objects which they own, to other subjects. This feature of discretionary model makes the notion of delegation of access rights essential in any discretionary policy. The problem with DAC is that it is hard to enforce a system-wide policy, because the owner of an object can delegate the access rights of the object to whoever he wants.

2.3.1.2 *Mandatory policy*

Mandatory Access Control (MAC) Policy is often used in military systems. In MAC, the access control is determined entirely on the basis of the security classification of subjects and objects. The Bell-LaPadula lattice-based model [61] is the most representative of mandatory security policy models. The Bell-LaPadula model identifies a set of objects (O) and a set of subjects (S) with: S is a subset of O, and defines the following actions that subjects can perform on objects:

- Execute (no observation, no alteration)
- Read (observation, no alteration)
- Append (no observation, alteration)
- Write (observation, alteration)

The model defines a set of totally ordered classifications *C* (Top-secret, Secret, Classified, Unclassified), and a set of categories *K* (e.g. France, China, Japan, etc.). A level is defined as: $L = C \times K$. Each object O is assigned exactly one level. L(O) is called the classification of the object. A Subject is assigned with two levels: L(S) and maxL(S). maxL(S) is called *clearance* and L(S) is called *security level*. The clearance of the subject is static, and $L(S) \propto maxL(S)$ must be true. Here, $\propto$ means "dominate". A security level (c_1, k_1) dominates (c_2, k_2) iff $c_1 \geqslant c_2$ and $k_2 \subseteq k_1$, where $c_1, c_2 \in C, \quad k_1, k_2 \in K$

The model defines the following properties, which constitute the mandatory access rules of the model:

- The Simple Security Property - a subject with a given clearance may not read an object at a higher security level (no read-up), i.e. for a subject S to be able to read object O, the following must be true: $L(O) \propto maxL(S)$.
- The Star-Property - a subject at a given security level must not write to any object at a lower security level (no write-down), i.e. for subject S to be able to write object O, the following must be true: $L(S) \propto L(O)$.

Subjects can never change the categories of objects, so the ownership between subject and object has disappeared. In MAC model, individual users can not alter access rights. This feature makes the MAC model an nightmare for general account administration in a dynamic and evolving system.

2.3.1.3 *Non-Discretionary policy*

Non-Discretionary Access Control (NDAC) policy [5] is appropriate for the situations in which some authorized users can modify the security policy, but the modification must be under control. A non-discretionary policy not only contains the access control policies, but also administrative policies to explicitly define which subject has rights to delegate rights to access an object.

The NDAC model can compensate the deficiencies of the MAC model and the DAC model.

2.3.1.4 *Role based policy*

Role Based Access Control (RBAC) policy [39, 81] is one of the most popular access control models. The key concept is to associate a *Role* for every subject. Privileges are no longer assigned to individual subjects, but to roles. So RBAC can considerably simplify authorisation management.

The disadvantage of RBAC is that it does not provide flexibility. The access right of a certain entity is bound to the role. It is rare that very large groups of entities all need exactly same access rights.

2.3.1.5 *Other security models*

Some models are proposed to address specific problems. The *Clark-Wilson model* [30] is designed to ensure the integrity of data. The policies describe how the data should be manipulated to preserve the integrity.

Chinese wall policy [21] is designed to avoid conflict of interest. The main idea is that a subject can only access information which is not in conflict with any other information that it already has.

2.3.2 *Policy specification languages*

Matt Bishop defines a security policy as *"a specific statement of what is and is not allowed"*. To specify security policies, we need to use policy specification languages. Many policy specification approaches have been proposed. Some of those languages are close to human language. Some of them are close to logic expression.

2.3.2.1 *ASL*

In 1997, *ASL* (Authorization specification language) was proposed by Jajodia et al. [50]. ASL uses first-order logic to express authorization policies. ASL uses predicate *cando* to specify authorization policy. The following formula is an example of authorization rule.

cando(file-a, user, +write) $\leftarrow$ in(user, professor)

This rule defines that all users who belong to the group professor are allowed to write on file-a. ASL provides two predicates *do* and *done* to specify history based authorization rules. However, ASL is not able to specify authorisation rules for groups of target objects that have the same type. ASL has been extended with new predicates [49] to evaluate the hierarchical relationships between users role.

2.3.2.2 *RDL*

In 1998, *RDL* (Role definition language) was proposed by Hayton et al. [47]. RDL defines a service as a set of proof rules (Horn clauses) that specify who can use it and in what way. RDL allows developers to specify authorization policies and delegations. In RDL, there are rules which define under which conditions a user can obtain a role. New roles can be assigned to users based on their actual roles. For instance, if a user has role Physicist and SeniorProfessor, this user can have a new role SeniorPhysicist.

The following example shows the RDL rule which defines the new role assignment.

$$\mathsf{SeniorPhysicist}(s) \leftarrow \mathsf{Physicist}(s) \wedge \mathsf{SeniorPorfessor}(s)$$

However, RDL can only allow the delegation of roles. It does not support the delegation of individual access rights.

2.3.2.3 *Ponder*

Ponder is a declarative, object oriented language which was proposed in 2001. Ponder allows developers to specify authorization policies and obligations [33]. Ponder can express delegation by means of specific delegation rules. Based on the delegation rules and authorization rules, we could determine which access rights can be delegated. The Ponder language is designed to specify security policies which can map onto various domains, such as firewalls, operating systems. Ponder also provides support to analyze conflicts and consistency.

Ponder is a declarative language, easy to read and write. But it is not domain specific. As a result, important concepts of pervasive computing applications (e.g. context) are not considered.

2.3.2.4 *Rei*

The policy language *Rei* was proposed in 2003; it is based on deontic concepts [52]. Rei is designed for pervasive computing applications. Rei provides constructs for expressing rights (permissions), prohibitions, obligations, and dispensations. Rei uses a meta policy to resolve conflicts between policies. Rei allows users to delegate not only rights to access services, but also to delegate rights to other users to allow them to delegate rights to access services. On the one hand, this feature allows end users to delegate privileges to whoever they want. On the other hand, it increases the workload of end-users and makes security concern no longer transparent for end user. A subset of the Rei language (authorization, delegation and revocation) has been applied in the Vigil [52] pervasive environment.

Although Rei can express authorization, obligation and delegation policies, it does not support any event triggered actions. We are not aware of any validation tool or approach for Rei.

2.3.2.5 *XACML*

XACML (eXtensible Access Control Markup Language) is also a declarative policy language which is based on an extension of XML [38]. XACML can express conditional authorisations and obligations. The first version of XACML was proposed in 2003. Then the XACML 2.0 was proposed in 2005. In the newest version (XACML 3.0), it includes an owner-centric delegation mechanism which is very similar to the delegation mechanism of Rei. XACML can be used to describe context constraints such as time and environment constraints.

Due to the fact that XACML is built upon XML, it is hard to read for humans. Even though there are approaches to retrofit XACML with appropriate validation tools such as Margrave [40] and RW [101], validation has never been a concern during the design of XACML.

2.3.2.6 *Access Nets*

The *Access Nets* approach was proposed in 2011, it uses an extension of Petri nets [74] to model the access control in physical places, such as offices or buildings [42]. Figure 8 shows an example of the access nets model for a museum. They use *tokens* to represent different kinds of users (e.g. visitor, guard, administrator). They use *places* to represent the physical places (e.g. lobby, gallery and archive). In *transition*, they can specify access control policies. For instance the transition *main(in)* (see Figure 8) defines that the visitors may only be in the museum between 9:00 and 17:00.

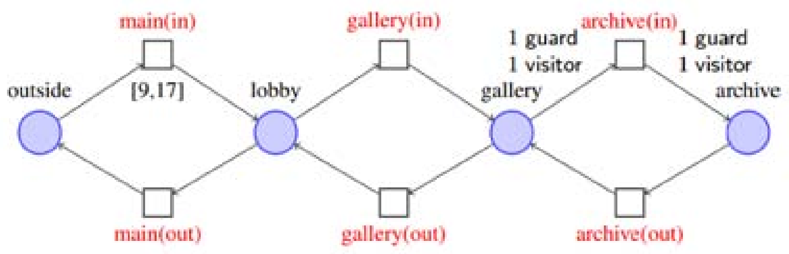


Figure 8: The ACCESS NET model for securing a museum

Access Nets models only the physical access control. The authors of Access Nets did not explain how to extend their approach in other domains (e.g. firewalls, pervasive computing systems, etc.).

2.3.3 *Access control policies based on term rewriting system*

This section presents formal approaches of access control policy based on term rewriting systems [10]. A term rewriting system is a Turing complete computational model. In this thesis, we use this approach to implement the security policy engine. The choice of a term rewriting approach is motivated by its expressivity, by the numerous available proof techniques, and by the efficiency of the various available implementations of rewrite engines. To be self-contained, we briefly recall several basic notions used in term rewriting system.

SIGNATURES A many-sorted signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ is given by a set of sorts $\mathcal{S}$, a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$.

A function symbol f with arity $\mathfrak{s}_1, \dots, \mathfrak{s}_n \in \mathcal{S}$ and co-arity $\mathfrak{s}$ has type $f{:}\mathfrak{s}_1, \dots, \mathfrak{s}_n \mapsto \mathfrak{s}$. A predicate symbol p with arity $\mathfrak{s}_1, \dots, \mathfrak{s}_n \in \mathcal{S}$ has type $p{:}\mathfrak{s}_1, \dots, \mathfrak{s}_n$.

Equality and inequality are the two predicates which we used in this thesis.

TERMS $\mathcal{T}(\Sigma, \mathcal{X})$ is the set of well-sorted terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. Variables are also sorted, and $x : \mathfrak{s}$ means that the variable x in $\mathcal{X}$ has sort $\mathfrak{s}$. The set $\mathcal{X}_\mathfrak{s}$ denotes a set of variables of sort $\mathfrak{s}$, and $\mathcal{X} = \bigcup_{\mathfrak{s} \in \mathcal{S}} \mathcal{X}_\mathfrak{s}$. We may omit sort names when they are clear from the context.

The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* (we may also call it a *closed term*). $\mathcal{T}(\Sigma)$ is the set of all ground terms.

De nition 1. *A substitution σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\Sigma, \mathcal{X})$, with a finite domain $\{x_1, ..., x_k\}$. It is denoted $\sigma = \{x_1 \mapsto t_1, ..., x_k \mapsto t_k\}$. For all $i \in \{1, ..., k\}$, x_i, t_i are of the same sort. The application of a substitution σ to a term t is denoted $\sigma(t)$.*

De nition 2. *(Rewrite rule). Given a signature Σ, a rewrite rule is an oriented pair of terms denoted $l \to r$ where $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$. We call l and r respectively left-hand side and right-hand side of the rule. A rewrite system is a set of rewrite rules.*

De nition 3. *(Rewrite Relation). Given a signature Σ and a rewrite system R over $\mathcal{T}(\Sigma, \mathcal{X})$, the rewrite relation associated to R is denoted $\to_R$ and is defined as follows: $t \to_R s$ if there exists a position p in t, a rewrite rule $l \to r$ in R and a substitution σ such that $t\,|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$.*

A term t is said irreducible by R, if there is no term s such that $t \to_R s$

A constrained rewrite rule over a signature Σ is a 3-tuple $(l, \varphi, r) \in \mathcal{T}_{\Sigma,\mathcal{X}} \times \mathcal{F}or_{\Sigma,\mathcal{X}} \times \mathcal{T}_{\Sigma,\mathcal{X}}$, denoted by $l \xrightarrow{\varphi} r$, such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{FV}ar(\varphi)$, where $\mathcal{FV}ar(\varphi)$ is the set of free variables in φ. A constrained term rewrite system (CTRS) $\mathcal{R}$ is a set of constrained rewrite rules. The symbol $\to_R$ denotes the rewrite relation with a CTRS that generalize the rewrite relation $\rightarrow_R$ on terms.

In this case, a constrained rewrite rule ($l \rightarrow r$ if φ) applies in a Σ-model $\mathcal{M}$ if the formula $\sigma(\varphi)$ holds in $\mathcal{M}$ [19].

STRATEGIC REWRITING The notion of strategy is fundamental in rewriting to express control on the rules application. Here, we only give a general presentation of the main ideas. The formal definition and the related properties can be found in [35].

In a term rewrite system R, if a term can be rewritten by two or more than two rewrite rules at the same time, we need to control how we apply these rewrite rules. The strategy allows controlling the application of rules on a term.

For instance, the strategy *universal* means that all rules are applied in any order. Suppose we have two rewrite rules $a \rightarrow b, a \rightarrow c$ and a term a. If we apply the strategy universal on a, then we get three results $\{a, b, c\}$. The strategy *first-applicable* selects in the ordered set of rules the first one that applies to the term. With the list of rules ($a \rightarrow b, a \rightarrow c$) and the strategy first applicable, the term a rewrites to b.

SECURITY POLICIES In access control policies based on term rewriting systems, policies are defined by a set of rewrite rules, governed by a rewrite strategy. The access requests and the environment where policies are enforced are represented as algebraic terms. When applied to an access request the rewrite rules induce an evaluation process which eventually returns an authorization decision.

Anderson et al. [35] give a formalization of access control policies based on term rewriting system. They define a security policy, $\wp$, as a five-tuple(Σ, D, R, Q, ζ) where:

1. Σ is a many-sorted signature;
2. D is a non-empty set of ground terms called decisions: $D \subseteq \mathcal{T}(\Sigma)$;
3. R is a set of constrained rewrite rules over $\mathcal{T}(\Sigma, \mathcal{X})$;
4. Q is a set of terms from $\mathcal{T}(\Sigma, \mathcal{X})$, called access requests;
5. ζ is a rewrite strategy for the set of rewrite rules R.

Anderson gives an example of a ticket system of the public transportation to illustrate their approach. The magnetic ticket can store the number of remaining trips and the time and data of the last validation. Users can take buses as many times as they want within 60 minutes. This is expressed by the following policy:

- The signature Σ contains :

  ticket: Nature $\times$ Time $\rightarrow$ Decision
  q: Ticket $\times$ Time $\rightarrow$ Decision
  deny: $\rightarrow$ Decision

- The set of decision D={$\mathit{deny}, \mathtt{ticket}(t_0, t_1) \mid t_0, t_1 \in \mathcal{T}(\Sigma)$}

- The set of rewrite rules R contains:

  q(ticket(n,time), current time)$\rightarrow$ ticket(n-1, current time)
  if (current time $>$ time + 60)$\rightarrow$ true
  q(ticket(n,time), current time)$\rightarrow$ ticket(n, time)
  if (current time $\leqslant$ time + 60)$\rightarrow$ true
  q(ticket(0,time), current time)$\rightarrow$ deny

- The access request q is a term from $\mathcal{T}(\Sigma, \mathcal{X})$, for instance, $\mathtt{q}(\mathtt{ticket}(10, t_0), t_1)$ is an access request, where t_0, t_1 are of sort Time.

- The strategy ζ = universal(R)

We give more details about the properties (e.g. consistency, termination) of the security policies based on term rewriting system in Section 8.1.1.3.

Tony Bourdier et al. [19] focus not only on the security policy, but also the system on which the security policies apply. They consider that a secured system *"consists of two parts: on the one hand, the set of rules describing the way the decisions are taken and on the other hand, the information used by the rules and the way these evolve in the system. We call the former the policy rules and the latter the security system."*

In this approach, a security system is *" a transition system that describes the way security information evolves. The states of the system are defined intensionally by a set of kernel information (that can be modified by the transition rules of the system) and an immutable set of closure rules used to compute the complete security information."*

They use transition rules to formalize the evolution of the system. A transition is triggered by an event. An event contains two terms: authorization query and decision. With the help of transition rules, they formalize the impact of a security policy on the system which it controls.

2.3.4 *Policy management architecture*

We have seen how to specify a security policy with policy specification languages. To realize those policies, we need to enforce them in the actual computing systems. In this section, we present three architectures which are dedicated to the management of security policies.

2.3.4.1 *IETF security policy management architecture*

Figure 9 shows the architecture which the IETF has proposed for the enforcement of policies. In the IETF architecture, the

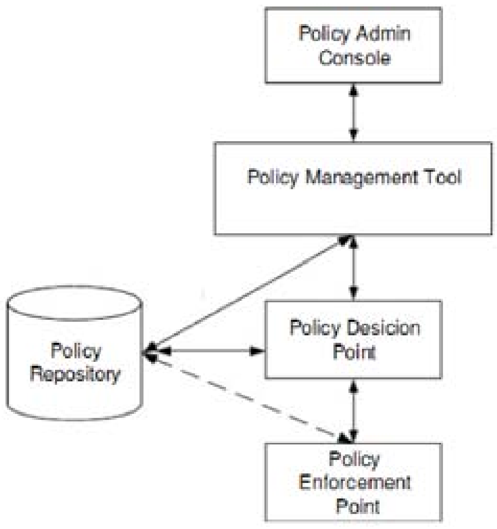


Figure 9: The IETF policy enforcement architecture

policy management tool is expected to provide a graphical user interface. The GUI needs to allow an administrator to specify security policies and store policies in the *policy repository*. The *policy enforcement point* (PEP) is responsible for enforcing policies with respect to authentication of subject, authorization to access and services. The *Policy Decision Point* (PDP) is responsible for managing one or more policy enforcement points. The PDP receives policy decision requests from PEPs and returns policy decisions to them. Policy enforcement involves the PEP applying actions according to the PDP's decision.

Verma [93] presents a detailed description of the IETF framework in his book. He describes the concepts and the implementation of the IETF architecture including policy validation, policy distribution, policy enforcement point and translation algorithms.

This architecture is considered the best approach for policy management. It has been used by many approaches. The architecture which we propose in this thesis is inspired by this architecture.

2.3.4.2 *Strongman security policy management architecture*

Keromytis et al. propose the *Strongman* security policy management architecture [57]. Figure 10 illustrates the main concepts of the Strongman architecture. Their main idea is to support various global high-level policy specification written in different languages. They provide a common lower-level policy interoperability layer which is implemented by using KeyNote. The interoperability layer is responsible for enforcing high-level policies on various network devices.

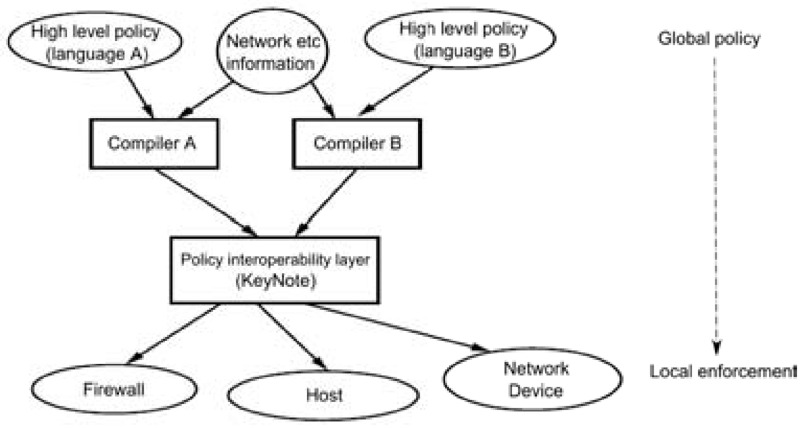

High level policy (language A)
Network etc information
High level policy (language B)
Global policy
Compiler A
Compiler B
Policy interoperability layer (KeyNote)
Firewall
Host
Network Device
Local enforcement

Figure 10: The strongman security policy management architecture

This architecture can enforce policies which are written in different languages in a system. But the authors do not explain how the high-level policies are managed by administrator. We do not have a clear idea of where the high-level policies are stored, how they interact with the network information.

2.3.4.3 *Self-Managed Cell architecture*

The *SMC* (Self-Managed Cell) architecture is proposed by Sloman et al [88]. Figure 11 [88] shows a graphical presentation of the SMC architecture.

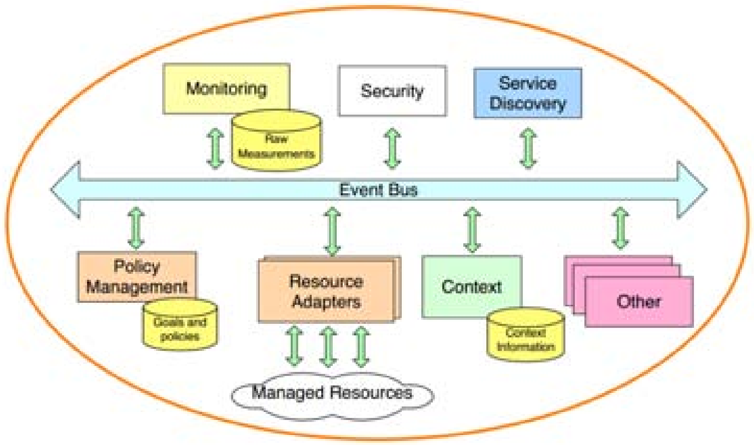


Figure 11: The Self-Managed Cell architecture

The *service discovery* component discovers nearby components capable of becoming members of the SMC (e.g. intelligent sensors), and other SMCs when they come into communication range. A *policy management* component manages the policies specifying the behaviour of the cell as well as policies supplied to new members to control their behaviour and how they interact with other SMC members. A *security* component performs authentication, confidentiality support and anomaly detection if needed. A *context* component determines and disseminates context information. A *event bus* is used for interaction between components and disseminating events which trigger policies

The SMC architecture was proposed to structure pervasive computing systems. This architecture gives us a first idea how the security policies interact with other components of the pervasive computing systems. However, the authors do not explain how the security policies are managed by the administrators.

2.4 SUMMARY

In this chapter, we first presented the pervasive computing system and its application domains. Then we presented the existing approaches which help programmers develop pervasive computing systems. We noticed that there are only few approaches that addressed the security concerns. But, most of them neglect the security concerns completely at design time. Few of them provide support for enforcing a security policy. None of them guides developers throughout the pervasive computing application lifecycle. We introduced the traditional approaches that addressed security policies. We noticed that there are some concepts in a security policy that are not present in pervasive computing system and vice versa. To specify appropriate security policy for pervasive computing system, we need to adapt the missing concepts, such as context information, to security policy.

3 CASE STUDY

In this chapter, we give a scenario of three typical pervasive computing applications. We first describe the required devices and the physical environment of the pervasive computing applications. Then we present the security requirements of this scenario. This scenario will be used throughout this thesis.

3.1 SCENARIO

As we mentioned before, our approach focuses on the domain of smart home. The scenario which we present in this section includes three typical pervasive computing applications of the smart home domain. The anti-intrusion application prevents unauthorized access to a building. The fire-detection application detects a fire in a building. The remote control application allows a user to control certain devices. Figure 12 shows the required entities of these applications.

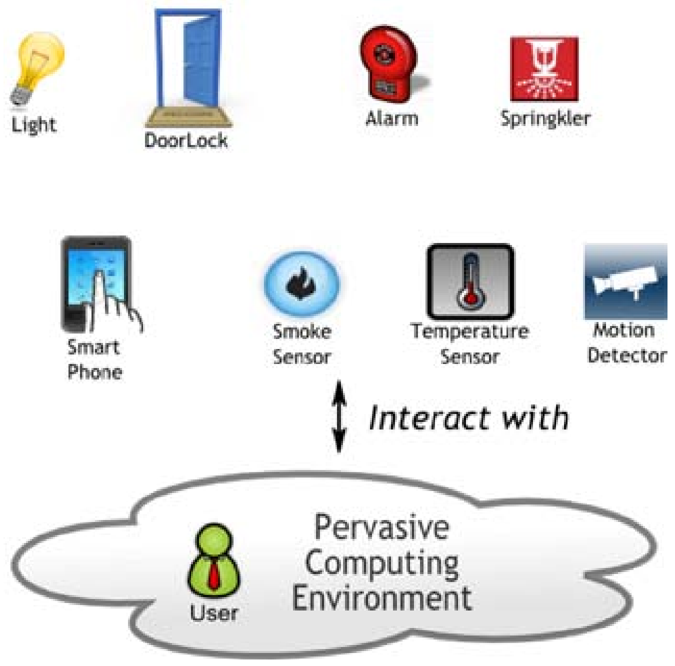


Figure 12: A pervasive computing application scenario

The anti-intrusion application uses motion detectors to detect unauthorized entrance in the building. If it detects an intrusion, it will trigger the alarm. The fire-detection application uses smoke sensors and temperature sensors to detect a fire. If it detects a fire, it activates sprinklers and sound alarms. The remote control application uses smart phones to receive control orders from authorized users. Authorized users can turn on or turn off lights and lock or unlock doors.

We suppose these applications are deployed in a building named *A29*. Figure 13 shows the location description of the research facility. We noticed that there are four rooms in the hallway East which are located on the second floor of building A29. We named the second floor *F2*.

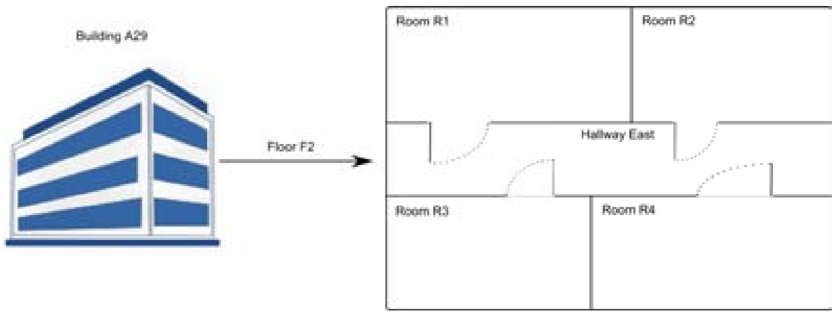


Figure 13: The location description of the research facility

These three pervasive computing applications can effectively illustrate our approach. The fire-detection application and anti-intrusion application can produce context information which may influence the decision of the security policies. The remote control application can effectively illustrate how we introduce the required concept to enforce security policies.

3.2 SECURITY REQUIREMENTS

We suppose there are five kinds of users in this research facility: professor, intern, phd-student, student and janitor. Interns and phd-students are the students who have particular privileges. We suppose that room R1 is the office of a professor named Alice, room R2 is the office of a professor named Bob, room R3 is the office of phd-students and interns, room R4 is the meeting room.

We suppose that only authorized users can access the research facility and use the devices. All unauthorized users' request will be denied. It means:

1. Only professor Alice can access to his office (i.e. room R1) and use the device in the office .
2. Only professor Bob can access to his office (i.e. room R2) and use the device in the office.
3. Phd-students can access to room R3 and turn on or off the light.

4. Students of the research facility can access room R4.
5. In case of the fire emergency, all members of the research facility can unlock all doors.
6. In case of the intrusion, only janitor can unlock doors.
7. In case of none of the above rules can be applied to a request, this request will be denied.

Part II

DEVELOPING SECURITY POLICIES IN PERVASIVE COMPUTING SYSTEM

4 OVERVIEW OF OUR APPROACH

In this chapter, we give an overview on our approach. We present the key concepts and requirements which we have identified, and solutions that we propose to address these requirements at each stage of the application life-cycle.

4.1 PRESENTATION OF OUR APPROACH

A successful deployment of security policy in a pervasive computing system requires two basic entities: a specification of security policy and a security-policy enforcement mechanism. Specifying and enforcing a security policy involves different stages of the application life-cycle. Due to the special requirements of pervasive computing systems (e.g. dynamicity, scalability, etc.), enforcing a security policy is tricky and time-consuming. To facilitate the task of specifying and enforcing a security policy, we propose a systematic approach to embed security policies into pervasive computing applications and illustrate it on a specific application.

Our starting point is the description of a pervasive computing application and the description of a security policy by rules expressing permissions and prohibitions.

The only assumptions that are made on these descriptions are as follow:

- The pervasive computing system has sensors and actuators, that are the two facets of entities corresponding to devices, whether hardware or software, deployed in an environment. In addition, a pervasive computing system may have ways to collect and transmit information.
- The security policy is expressed by constrained rules in first-order logic where constraints have variables taking values in finite or denumerable domains, and satisfiability of constraints is decidable.

For the sake of clarity, we use two concrete languages (i.e. DiaSpec [25] and DiaSecur[1]) to illustrate our approach. Figure 14 shows how to develop a secured pervasive computing application with our approach. In the following sections, with the idea that security should be integrated all along the life-cycle of pervasive

1 A domain specific policy specification language defined in Chapter 5

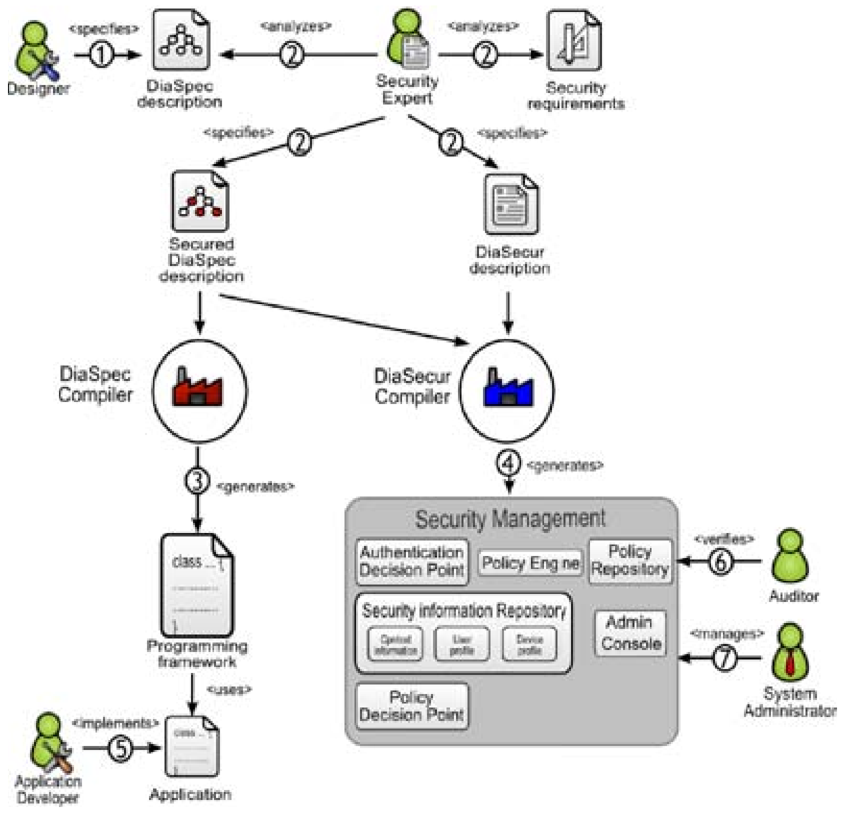


Figure 14: Enforcing security policy with our approach

computing applications, we show how security policies are defined, enforced, developed, tested, deployed, and reconfigured if security requirements change.

4.2 DESIGN

To facilitate the specification of security policies, we propose a security policy specification language named DiaSecur. To enforce the security policy, we propose an approach to manage the security policies and the interface between pervasive computing systems and security policy management mechanisms.

At the design stage, the application designer describes basic components (e.g. devices, context, controller) and the design of pervasive computing applications (stage ①). To secure this pervasive computing system, a security expert analyzes the security requirements and the description of pervasive computing applications. After the analysis, he identifies the components which need to be secured and tag them with specific security annotations (stage ②). Lastly, the security expert specifies security policies to fulfil security requirements (stage ②).

4.3 IMPLEMENTATION STAGE

At the implementation stage, the secured pervasive computing application descriptions are passed to two compilers. The DiaSpec compiler generates a programming framework (stage ③). This framework contains an abstract class for each entity (e.g. devices, contexts and controllers). The abstract classes contain abstract methods which can facilitate the implementation of the pervasive computing application. Based on the security annotations, by adding new methods or modifying existing ones in the generated abstract class, the enforcement of security policies is seamless. Based on the secured pervasive computing description and a DiaSecur description, the DiaSecur compiler generates a security manager which manages the security policies and security related information (stage ④). The security manager treats the authentication and authorization requests based on the security policies and context information. Lastly, the application developer implements each entity by extending the generated abstract class (stage ⑤).

4.4 VERIFICATION AND TEST STAGE

After the design and implementation stage, and before the deployment stage, the security policies need to be verified. The crucial properties such as absence of conflicts and termination of request evaluation need to be ensured to prevent potential security breaches. We propose a tool which allows an auditor to verify certain properties of security policy (e.g. conflicts) (stage ⑥), and provides help to resolve conflicts and incompleteness.

To test the behavior of security policy in pervasive computing applications prior to deploying a real environment, we use a 2D simulator, named DiaSim [23]. Security policy auditor can test the impact of security policy in the simulated pervasive computing environment (stage ⑥).

4.5 DEPLOYMENT AND MAINTENANCE

To deploy the security policy, some security relevant information such as user profiles and device profiles need to be initialized. The class of entities (e.g. sensors, actuators) which the application developer implemented need to be instantiated. After deployment, the maintenance of the secured pervasive computing system may require that the system administrator adds or deletes user profiles or reconfigures policy rules.

We propose a graphical user interface which allows the system administrator to manage the security policy rules and other security relevant information (stage ⑦).

5 SECURITY POLICY SPECIFICATION LANGUAGE

In this chapter, we present our policy specification language DiaSecur dedicated to specify authorization policies in pervasive computing systems. We first present the domain analysis which identifies the basic components of our language. Then we present the basic entities of our language. Lastly, we present how to use term rewriting systems to implement security policies in our approach.

5.1 DOMAIN ANALYSES

Many computer languages are domain specific rather than general purpose such as Microsoft-Excel, SQL. Domain-specific languages (DSLs) focus on a particular application domain. By using specific notations and constructs, domain-specific languages offer substantial gains in expressiveness and ease of application development. Dave Thomas wrote in his paper [91] *"Domain-specific languages lift the platform s level, reduce the underlying APIs surface area, and let knowledgeable end users live in their data without complex software centric models and the API field of dreams."*. Martin Fowler said *"Domain specific languages (DSLs) are very good at taking certain narrow parts of programming and making them easier to understand and therefore quicker to write, quicker to modify and less likely to breed bugs"* [41].

To identify the terms and concepts to be used in our language, we need to analyze our problem domain. Domain analysis can give us a clear understanding of desired system features and the required functionalities to implement those features. In DSL development, domain analysis is essential to define and exploit commonalities and variabilities in a given problem domain. The table 1 shows us some definitions of the basic terms in domain analysis.

Most of DSLs are developed with informal domain analysis. Because it is much easier to realize. Formal domain analysis is difficult to accomplish, but the output is richer. In this thesis, we use a formal analysis method, named FAST [98, 99].

Term	Definition
Context:	The circumstances, situation, or environment in which a particular system exists.
Domain:	A set of current and future applications which share a set of common capabilities and data.
Domain analysis:	The process of identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain. Domain engineering: An encompassing process which includes domain analysis and the subsequent construction of components, methods, and tools that address the problems of system/-subsystem development through the application of the domain analysis products.
Domain model:	A definition of the functions, objects, data, and relationships in a domain.

Table 1: Definitions of the basic terms in domain analysis

5.1.1 *Domain analysis with FAST*

FAST approach uses SCV analysis [98] to identity, formalize and document commonalities and variabilities in a given problem domain. SCV stands for Scope, commonality, and variability. There are five main steps in SCV analysis as followed:

1. Establish the scope: the collection of objects under consideration.
2. Identify the commonalities and variabilities.
3. Bound the variabilities by placing specific limits such as maximum values on each variability.
4. Exploit the commonalities.
5. Accommodate the variabilities

5.1.1.1 *Establish the scope*

At this step, we identify and define the domain which we work on. As I mentioned before, our problem domain consists of security policy and pervasive computing system.

A security policy is a set of objectives, rules of behaviors for users and administrators, and requirements for system configuration and management that collectively are designed to ensure Security of computer systems in an organization. In this domain, our objective is to define security policies which can limit the activities of a user. A security policy is a group of *policy rules* which define and govern the right behaviors of a system. A policy rule defines *actions* that *subject* can perform on an *object* when a set of *conditions* are satisfied. Actions represent the operations which subjects want to perform. Subjects and objects are the basic entities in the system. For example, a subject could be a user or a process, an object could be a door or an alarm. The condition defines in which situation the policy rule can be applied. *Decision* is given when system send a *request* to enquiry the security policy. We can identify **policy rule, action, subject, object, condition, request, decision** as basic terms.

A pervasive computing system consists of intelligent computational entities (software or hardware) which can collect, communicate and reason about data of the environment and providing services to users. SOUPA [27] defines a standard ontology for pervasive computing systems which has been used by many approaches (e.g. [62], [26], [66]). It identifies basic entities such as **device, person, event, action and location**.

5.1.1.2 *Commonalities and variabilities*

David.M Weiss [98] proposed several mechanisms(Procedures, Inheritance, etc.) for exploiting commonalities and variabilities in FAST approach. We choose *Parametric polymorphism* mechanisms for our domain analysis, because it is the one most widely used. In this mechanism, we implement the same operation with different existing approaches. Then we observe the identical parts and different parts.

We introduce a little scenario which describes security requirements. Then we use different approaches (e.g. Rei, Ponder, XACML, SPL) to write security policy rules to fulfil these requirements. The scenario and policy speifications can be found in appendix A.

In the policy specifications of Rei, Ponder, XACML, the concepts such as subject, object, action, condition and decision are explicit. In SPL, author(subject), target(object), action are represented as attributes of events. The concept of condition and decision are hidden in the acceptability of the event (Listing 38, line 2). The decision of policy rule is not explicit, it depends on the evaluation of the logical expression. However we are convinced that **subject, object, action, condition and decision** are commonalities.

Variabilities in a domain are the elements which are optional or alternative. We can imagine that the characteristic of subject and object could be different in different scenario. For instance, in an on-line DVD rental shop, we want to set a security policy which can prevent the teenager to rent a DVD for adult. So the age is indispensable for the subject. Nevertheless if we want to set a security policy for printer access in a school, the age is no longer needed. So the attributes which we use to characterize subjects and objects are variabilities.

5.1.2 *Identified Concepts and Terminology*

We have identified the key concepts of pervasive computing systems and the security policies. To develop a security policy dedicated to pervasive computing systems, the security policy needs to adapt some concepts of pervasive computing system. To embed a security policy into pervasive computing systems, the pervasive computing system needs to adapt some concepts of security policy.

Figure 15 shows how we combine the concepts of pervasive computing systems and security policies. We have three pairs of "common concept". Person in pervasive computing system describes the profile of a user which is capable of executing an action on the devices. Subject is the entity which wants to gain access to an Object. The concept of person maps into the concept of subject, the concept of device maps into the concept of object, and the concept of action exists both in pervasive computing system and security policy. For the sake of clarity, we named the three pairs of "common concept" as subject, object and action. To enforce security policies, these three basic concepts are essential.

A concept such as location which describes the ambient information of pervasive computing system is essential for context aware security policies. The security policy needs to adapt these concepts to fulfil the requirements of pervasive computing systems. The pervasive computing system needs to form requests to apply security policy.

5.2 DIASECUR

In this section, we present the basic entities which we identified in the domain analysis. These entities are the basic building blocks of our security policy specification language dedicated to specify authorization policies in pervasive computing systems.

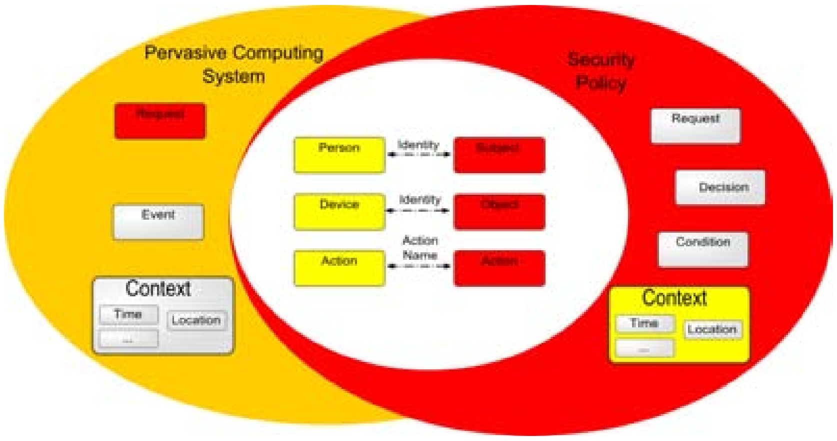


Figure 15: The merger of the concepts of pervasive computing system and security policy

5.2.1 *Basic entities*

5.2.1.1 *Subject*

We identified three key attributes to characterize subject: *identity*, *role*, *location*. We use the attribute identity to distinguish one user from another. We use the attribute role to group the users who have the same privileges. It prevents duplicating the policy rules for different subjects which have the same role. The attribute location is also a very important characteristic for subjects, because in pervasive computing system, the privilege of a subject could be different in different location.

5.2.1.2 *Object*

We use three key attributes to characterize object: *identity*, *objectType*, *location*. We use the attribute identity to distinguish one object from another. We use the attribute objectType to group the objects which are the same kind. It can avoid assigning privileges to objects one by one, when they possess the same properties. We use the attribute location to group objects which are located at the same place.

5.2.1.3 *Action*

Actions are the activities which can be executed by actuators. The action definition in DiaSpec starts with keyword **action**, followed by the name of action. An action could contain one or several methods. List 8 is an example of action definition. We borrow this language fragment from DiaSpec to define actions.

```
action AlarmAction{
  On();
  Off();
}
```

Listing 8: An example of action definition in DiaSpec

5.2.1.4 *Role*

In contrast to the access control systems without roles, we can associate privileges to a set of users instead of only one user.

A role system definition starts with keyword **Roles**, followed by a set of role definitions. The role definition consists of two parts. The first part declares the name of the role. It starts with keyword **Role**, followed by the name of the role. The second part describes the role hierarchy. It starts with keyword **inherit**, followed by a list role names. The role hierarchy can not only express the relationship of roles, but also delegate rights to senior role. List 9 shows an example of role definition. Role phd_student inherits all rights from role student (Line 5).

```
1 Roles {
2        Role professor;
3        Role student;
4        Role intern inherit student;
5        Role phd_student inherit student;
6        Role janitor;}
```

Listing 9: An example of role definition in DiaSecur

5.2.1.5 *Location*

In pervasive computing, the location given by sensors (e.g. GPS, RFID tag, etc.) are not readable by human users. So, in our language we use abstract locations.

Such a smart space could be a single room, an entire building or a set of buildings. We define a location with the keyword **Locations** followed by a set of smart space definitions. A smart space definition could contain one or more buildings. A building contains one or several floors, a floor contains one or several hallways, a hallway contains one or several rooms.

When we define a location, we have several constraints. We can not have two rooms which have the same name in the same floor, or two floors with the same name in a building. In other words, location names must be unique. Listing 10 shows an example of location definition.

```
Locations{ Building A29 {
              Floor F1{
                Hallway West {
                     Room R1;
                     Room R2;}
                      }
              Floor F2{
                Hallway East {
                     Room R3;
                     Room R4; }
                      }
                        }
         }
```

Listing 10: An example of location definition in DiaSecur

5.2.2 *Authorization rules*

Authorization rules define which subject can do or can not do which action on which object under which condition. In our case, we declare the authorization rules at the design stage. We do not have the concrete information of the device and user instances. As a result, we can only assign privilege of a group of objects to a role. To be more precise, we define which kind of subjects can do or can not do which action on which kind of objects. As a result, we can define a positive authorization rule as in line 1 of Listing 11. It starts with the keyword **Permission**, followed by the role of the subject, the action and the type of the object. We can define a negative authorization rule as in line 2 of Listing 11. It starts with the keyword **Prohibition**, followed by the role of the subject, the action and the type of the object. We use the keyword **if** to declare the conditions.

```
1 Permission(professor, open, door) if condition.
2 Prohibition(professor, close, door) if condition.
```

Listing 11: Authorization rules

The role of the subject can be concrete roles (e.g. professor, student, etc.). It also can be a variable, denoted by `subRole`. The action and the type of object also can be variables, denoted by `action` and `objType`. For instance, we can write a rule Prohibition(subRole, action, objType). As we give no constraints to these variables, this rule means "anyone is prohibited to do any action on any object".

CONDITION In DiaSecur, a condition is a decidable quantifier-free logic expression, built as a set of comparisons which are separated by boolean operators: 'or' and 'and'. In a comparison, values can be compared by: '==' (equality) and '!=' (inequality). We can divide the comparisons into two categories: *context comparisons* and *attribute comparisons*.

A context comparison starts with a context variable, followed by an operator (i.e. '==', '!='), followed by a value which has the same type as the context variable. For instance, $X_{\mathtt{OnFire}} == \mathtt{true}$ is a context comparison. A context variable is built based on the name of a context. The context name and the type of the context value are both provided by the pervasive computing application description.

An attribute comparison starts with an attribute variable, followed by an operator (i.e. '==', '!='), followed by a attribute value. For instance, `subjectId!=alice` is an attribute comparison. An attribute variable is built based on the name of an attribute. In this work, we fix that a subject only has three attributes (subjectId, subjectRole, subjectLocation), an object only has three attributes (objectId, objectType, targetLocation).

At design time, in the pervasive computing system, the name of contexts and their types are already defined. In the security management, to be aware of the context value, We need to store the context names and the type of these contexts. The context names are used as the keys to access the context values.

At run time, the context values are provided by the pervasive computing environment. Then this context value is stored in the security management. For instance, if the pervasive computing system produce a context value true of the context `OnFire`, in the security management, the value associated to the context name `OnFire` is replaced by the value true. When a policy rule needs to evaluate his condition such as $X_{\mathtt{OnFire}} == \mathtt{true}$, the context variable $X_{\mathtt{OnFire}}$ is replaced by the value associated to the context name `OnFire`. In this example, the context variable is replaced by the value true. In Section 6.2.2, we give more details of how we define security relevant contexts, and how they provide context values. The attribute values are provided by the profile of subjects and objects.

5.2.3 *Generated term rewriting system*

To enforce the security policies, the policy specification is compiled into a term rewriting system which is defined with a signature and a set of rewrite rules.

5.2.3.1 *Signature of the generated term rewriting system*

In the signature, we use functions, sorts and constants to represent subject, object, action and their attributes (e.g. location, roles and types of objects). Listing 12 is an example of a generated signature where sorts are in blue.

LOCATION A term of sort *location* consists of four subterms of sorts: *building, oor, hallway and room* (lines 4 to 8). Building, floor, hallway and room are declared as sorts.

SUBJECT A term of sort *subject* consists of three subterms of sorts: subjectId, subjectRole and location (line 12). *SubjectId* is declared as a sort which has a primitive type (e.g. String, int). *SubjectRole* is declared as a sort (line 2).

OBJECT A term of sort *object* consists of three subterms of sorts: objectId, objectType and location (line 11). ObjectId is declared as a sort which has a primitive type. *ObjectType* is declared as a sort (line 3).

```
1  Signature {
2  janitor, professor, student, intern: -> subjectRole
3  doorlock : -> objectType
4  loc : building * floor * hallway * room -> location
5  A29 : -> building
6  F1 : -> floor
7  West, East : -> hallway
8  R1, R2 , R3, R4: -> room
9  DoorAction_close, DoorAction_open : -> action
10 req : subject * action * object -> request
11 obj : objectId * objectType * location -> object
12 sub : subjectId * subjectRole * location -> subject
13 deny, permit : -> decision
14  }
```

Listing 12: An example of a generated signature of term rewriting systems

REQUEST AND DECISION Based on the domain analysis in Section 5.1, we decide that an authorization request is built with the function symbol **req**, followed by three subterms of sort subject, action and object. The decision consists of two constants: `permit` and `deny` (see listing 12).

POLICY RULES The policy rules of DiaSecur generate a term rewriting system. Listing 13 shows an example of the generated rewrite rules. The first rule defines that the student who has the identity *chuck* is allowed to unlock the door located at building

A29, floor F1, hallway east and room R2 (lines 2 to 4). The left-hand side of the rewrite rule is a term built with the function symbol **req** and three subterms of sort subject, action and object. The term of sort subject is built with the function symbol **sub** and three parameters. These parameters can be constants or variables. The constants are declared in the previous signature. To express a more abstract subject, variables of sorts *subjectId*, *subjectRole* and *subjectLocation* may be used. For instance, the term subject contains two variables `subId` and `subLocation` and a constant `student` of sort *subjectRole* the role of the subject. These variables can be used in the conditions. The terms of sorts *action* and *object* in the rules are built the same way as we build the subject.

```
1  Rewrite rule {
2  req(sub(subjectId, student, subjectLocation),
3      DoorAction_open,
4      obj(objectId, doorlock, loc(A29,F1,East,R2))) -> permit
5      if subjectId=="chuck" ;
6
7  req(sub(subjectId, professor, subjectLocation),
8      DoorAction_open,
9      obj(objectId, doorlock, loc(A29,F1,West,R2))) -> deny
10     if subjectLocation!=loc(A29,F1,West,R2) ;
11  }
```

Listing 13: An example of a generated rewrite rules of term rewriting systems

5.2.3.2 *Request evaluation*

Request evaluation in our approach is based on term evaluation by rewriting. When applied to an authorization request (a term of sort *request*) the rewrite rules induce an evaluation process which eventually returns a decision. The evaluation process using term rewriting systems is precisely defined in [10, 35].

Here, we only give a general presentation of the main ideas. Suppose we have a set of policy rules $R = \{r_1, ..., r_n\}$, a context environment Con, a strategy first-applicable and a set of decisions $D = \{permit, deny\}$. A request q is a term of the form $req < sub, act, obj >$, where sub, act and obj are terms which are built by constants. For instance, sub can be expressed as *sub(alice, student, loc(A29,F1,West,R2))*, where 'alice' is a constant of sort *subjectId*, student is a constant of sort *subjectRole*, loc(A29,F1,West,R2) is a term of sort *location*. Since a policy rule in the term rewriting systems has the form $req(subject, action, object) \rightarrow d \;\; \mathtt{if} \;\; c, where \;\; d \in D$ and c is a condition, matching policy rules to the requests is easy and amounts to check a system of simple equations. If there is a solution σ which matches the left-hand side of the policy rule against the request, and if $\sigma(c)$ is true in context environment Con (provided by the pervasive computing environment), then the right-hand side of this rewrite rule is the decision of the request. If there is no solution σ, then we continue to apply the next policy rule. If no rule of the policy can be applied, then the policy is incomplete. We will explain how to resolve this problem in Section 8.1.3.

CONDITION EVALUATION We have defined a context environment Con as a set of contexts. At run time, a context has the form <contextName, contextValue>. As described in Section 5.2.2, the condition is a set of equality comparisons, the condition evaluation amounts to prove a set of equalities. For instance, suppose we have a context condition $X_{\mathtt{OnFire}}$==true in a policy rule that is selected for application, and a context <OnFire, contextValue> in *Con*. At run time, the contextValue is provided by the pervasive computing environment (e.g. false). The context condition evaluation amounts to instantiated the variable $X_{\mathtt{OnFire}}$ by the context value false and compare the equality of two constants.

The evaluation of an attribute condition is the same as the context condition. Indeed the condition evaluation also involves the disjunction and conjunction of the results of the context and attribute comparison evaluations.

5.3 SUMMARY

In this chapter, we presented our policy specification language DiaSecur dedicated to specify authorization policies in pervasive computing systems. Based on the domain analysis, we presented the basic entities of our language. We also presented how to use the term rewriting system to implement policies in our language. Compared to other approaches, our language takes care of context information (e.g. location description) in order to make the policies context aware. In this work, we fix the attributes in subject and object. However, new attributes could be added in subject and object definitions to support other pervasive computing systems. For example, we could add a new attribute *age* in the subject for the assisted living applications.

6 DESIGNING THE SECURED PERVASIVE COMPUTING APPLICATIONS

In this chapter, we first present how our approach enforces and manages security policy in a pervasive computing system. Then we use enriched DiaSpec specifications to illustrate how to apply our approach in DiaSuite. Lastly, we use the applications which we introduced in the case study (Chapter 3) to illustrate how to design a concrete secured pervasive computing applications by using our approach.

6.1 DESIGN OF SECURITY MECHANISM

A security policy needs to be enforced to fulfil the security requirement. To do that, we need a security mechanism to enforce security policy. Figure 16 shows the design of our security mechanism.

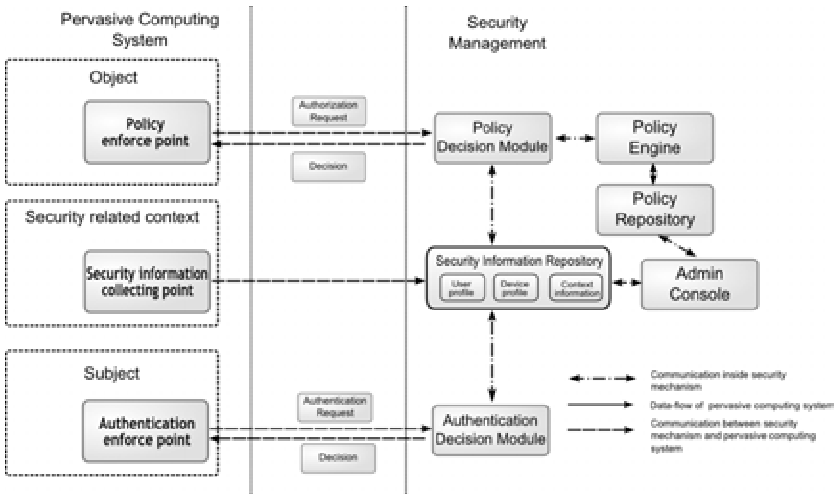

Pervasive Computing System
Security Management
Object
Policy enforce point
Authorization Request
Decision
Policy Decision Module
Policy Engine
Policy Repository
Security related context
Security information collecting point
Security Information Repository
User profile
Device profile
Context information
Admin Console
Subject
Authentication enforce point
Authentication Request
Decision
Authentication Decision Module
Communication inside security mechanism
Data-flow of pervasive computing system
Communication between security mechanism and pervasive computing system

Figure 16: Architecture of the security mechanism

Our security mechanism can be divided into two parts. To support the dynamicity of the pervasive computing system, we isolate the components which are independent of the pervasive computing system and call them as *security management*. To enforce security policies in pervasive computing applications, we introduce three key concepts (subjects, objects and security related context). We introduce *security enforcement points* that enrich the existing entities of the pervasive computing system and establish interface between the pervasive computing system and the security management.

6.1.1 *Security management*

Verma [93] describes that a *policy management tool* is expected to provide a graphical user interface which allows an administrator to specify security policies and store policies, it also can receive requests and return decisions based on the provided policies with respect to authentication of subject.

Based on the above description, we decide that a security management consists of a security information repository (SIR_{sm}), an authentication decision module (ADP_{sm}), a policy engine (PEP_{sm}), a policy decision module (PDP_{sm}), a policy repository (PR_{sm}) and an Administration Console (AC_{sm}).

- The security information repository stores security information such as user profiles, device profiles, context information (e.g. fire, intrusion).
- The authentication decision module is responsible for handling authentication requests and making decision based on requests and user profiles.
- The policy decision module is responsible for handling authorization requests and sends decisions back to the pervasive computing system.
- The policy engine processes the authorization requests based on the policy rules and context information.
- The policy repository is responsible for storing the security policies.
- The administration console provides a graphical user interface to help users manage the security policy and security data such as user profiles and device profiles.

6.1.2 *Interface between pervasive computing system and security policy*

Developing the interface between pervasive computing systems and security mechanisms requires the mapping of the key concepts of the two domains. Based on the domain analysis in Section 5.1, we have identified three key concepts: subjects, objects and security related contexts.

The concept of context in pervasive computing systems does not exist in traditional security policies. In pervasive computing systems, the security policies which can not take into account the context information may impede functionalities of pervasive computing systems. To effectively support pervasive computing, we encapsulate context with the concept of security related context and import this concept in security policy. Our security management mechanism is capable to receive and store the context information which is provided by pervasive computing environment. Our policy specification language is based on rule based formalism [19]. Policy rules are constrained rewriting rules. The constraints allow us to handle the concept of security related context in security policy.

6.1.2.1 *Security enforcement points*

We propose three kinds of security entities to introduce the basic entities (subject, object and security related context) in pervasive computing systems: security information collecting point (SICP_{se}), authentication enforce point (AEP_{se}), policy enforce point (PEP_{se}). For the sake of clarity, we call the SICP_{se}, AEP_{se} and PEP_{se} security enforcement points.

A security information collecting point (SICP_{se}) is used to build the concept of security relevant context. If the pervasive computing environment can provide directly the security relevant context information, the security information collecting point is responsible for sending the context information to the security information repository (SIR_{sm}). If the pervasive computing system does not have the concept of context, the security information collecting point is responsible for collecting raw data from sensors and refining these raw data to produce the security relevant context information. Then it sends the context information to the security information repository.

An authentication enforcement point (AEP_{se}) is used to build the concept of subject. In some cases, the pervasive computing system contains already the concept of user. In these pervasive computing systems, the authentication enforcement point creates credentials based on the identity of users and expiration time. Then the request sent by users will be tagged by the credential. If the pervasive computing system does not have the concept of user. The authentication enforcement point is responsible for embedding the concept of device in the concept of user (subject). It forces the users who use these devices to authenticate themselves. To do so, the authentication enforcement point introduces the identity and identity proof of users. Based on the identity and identity proof, it forms the authentication request and sends it to the authentication decision module.

The policy enforcement point (PEP_{se}) is used to build the concept of object. The policy enforcement point is responsible for embedding the concept of actuating devices in the concept of objects. Objects need to be monitored by security policies. To do that, a policy enforcement point forms the authorization requests based on the credential of the receiving data and the identity of the device. Then it sends the request to the policy decision module.

6.1.3 *Embedding security enforcement points in pervasive computing systems*

The security enforcement points play the role of the bridge between pervasive computing systems and security policies. They guide the implementation of the required functionalities of the key concepts. They need to be embedded at the right places of the pervasive computing systems. So the first step is to identify the entities which represent the key concepts in pervasive computing systems. Then tag them with security enforcement points. These entities may be concrete or abstract. For instance, a concrete device such as smoke sensor can be used to provide context information, an abstract context class "Fire" also can be used to provide context information. We propose specific annotations to tag basic entities with security enforcement points.

6.1.3.1 *Annotations for security enforcement points*

An annotation, in our approach is a structural declaration to mark different entities (e.g. devices with actuating capabilities) with dedicated security enforcement points (e.g. Policy enforcement point). The annotation allows us to use more abstract concepts to encapsulate low level entities. For instance, an entity smart phone tagged with an annotation authentication enforcement point represents the concept of user. This smart phone needs to realize the user required functionalities which we described in Paragraph 6.1.2.1.

The annotations allows us to automatically generate the functionalities of the security enforcement points inside the entities. We illustrate how we define an annotated entities in Section 6.2. In Section 7.1, we present how we generate support to implement the required functionalities in the annotated devices.

6.2 APPLYING OUR APPROACH IN DIASUITE

In this section, we use DiaSuite [31] to illustrate our approach. We can separate DiaSpec declarations for devices into two categories: sensing devices and actuating devices. We annotate a sensing device declaration to promote it to type subject, an actuating device declaration to promote it to type object, and a context declaration to introduce a type security related context. For simplicity we write subjects (resp. objects and security related contexts) instead of entities of type subject (resp. object and security related context).

6.2.1 *Device*

Devices are used to describe the computational entities and their sensing or actuating capacities. In DiaSpec, there is no explicit definition of users. Everything is represented by devices. So we divide sensing devices into two different types: user device, infrastructure device.

- User devices are responsible for representing users and reflecting the intention of users. The user devices must have sensing capabilities to collect the intentions of users. Then they translate the intentions of users into requests which can be managed by the pervasive computing system.
- Infrastructure devices collect the information of the physical environment such as temperature and smoke.
- Actuating devices provide services that can be executed by users.

User devices may need to be used to represent more than one user. The user devices need to provide authentication to support this kind of capabilities. To avoid the abuse of the services provided by the pervasive computing system, the actuating devices need to be monitored by the security policy. To achieve the security requirements at the design stage, we enriched the devices declaration with security enforcement point annotations.

6.2.1.1 *Enrich user device*

In DiaSpec, we enriched user devices (sensing device in DiaSpec) to introduce the concept subject in pervasive computing application. We use authentication enforce point annotations to indicate which user device need to be enriched. Listing 14 shows a user device `UserDevice` declaration in DiaSpec with an authentication enforcement point annotation.

```
device UserDevice {
attribute id as String;
source userRequest as request;
      security{ require AEP on source userRequest; }
 }
```

Listing 14: An example of authentication enforcement point in an user device

Inside the security statement, we can declare a security enforcement point. Line 4 shows how we tag a device source with authentication enforcement point. It starts with key word **require**, followed by `AEP` (the type of security enforcement points). The key word **on** indicates which source is tagged by user's credential. In this example, the authentication enforcement point indicates that when a `UserDevice` wants to send a `userRequest`, it needs the user to identify himself. The declaration also indicates that the userRequest are tagged with users credentials.

6.2.1.2 *Enrich actuating device*

In DiaSpec, we enriched actuating devices to introduce the concept of object. We use policy enforce point annotations to indicate which action device needs to be enriched. Listing 15 shows an action device `ActionDevice` declaration with the policy enforcement point. The policy enforcement point declaration indicates that the action `DeviceAction` of this device are monitored by the security policies.

```
1 device ActionDevice{
2    attribute id as String;
3    action DeviceAction;
4        security{ require PEP on action DeviceAction; }
5 }
```

Listing 15: An example of policy enforcement point in an action device

Line 4 shows how we declare a policy enforcement point. In this example, the policy enforcement point declaration indicates when the device `ActionDevice` wants to execute action `DeviceAction`, it must ask the authorization of the security policies.

6.2.2 *Enrich Context*

In DiaSpec, the entity *context* is designed for refining the raw data collected from devices. Some of the context can produce data which describes the surrounding environment of users. These data may influence the decision of security policy. The security policy needs to be aware of these data.

We enriched the context declaration to introduce the concept of security related context. We use security information collecting point (SICP) to indicate which context need to be enriched. Listing 16 shows an example of a context declaration enriched by a security information collecting point. We suppose the name of the context is `ContextA`. It receives data which has type `dataA` from

`DeviceA` and data which has type `dataB` from `DeviceB`. It refines these data and produces more abstract data (context information).

```
1 context ContextA as Boolean{
2   source dataA from DeviceA;
3   source dataB from DeviceB;
4 
5         security{ require SICP on context ContextA; }
6 
7 }
```

Listing 16: An example of security information collecting point

Line 5 shows how we declare security information collecting point. To make the security policies aware of this context information, the SICP sends the context information to the *security information repository*.

In listing 17, we give a grammar of the security enforcement point annotation.

```
annotation: □security□ □{□ annotation_body □}□;

annotation_body: aep_annotation
               | pep_annotation
               | sicp_annotation
               ;

aep_annotation : □require□ □AEP□ □on□  sensor_source □;□ ;

pep_annotation : □require□ □PEP□ □on□  actuator_action □;□ ;

sicp_annotation : □require□ □SICP□ □on□  context_info □;□ ;

sensor_source: □source□ sourceName ;

actuator_action: □action□ actionName ;

context_info: □context□ contextName ;

sourceName: ID;
actionName: ID;
contextName: ID;

ID: (□a□..□z□ | □A□..□Z□)(□a□..□z□|□A□..□Z□|□0□..□9□|□_□)* ;
```

Listing 17: Grammar of the security annotation

6.3 Designing the applications in the case study

In this section, we use the scenario which we give in Chapter 3 to illustrate how to design a secured pervasive computing application with our approach.

At the design stage, we introduce two kinds of developers: a pervasive computing application designer and a security expert. The pervasive computing application designer first describes the available entities of the pervasive computing environment. Then he designs the architecture of the pervasive computing application. The security expert first identifies the DiaSpec entities which can be encapsulated by the security concepts (subject, object and security related context), based on the security requirements and the architecture of the pervasive computing applications. Then the security expert tags the identified entities with security enforcement point to indicate which entity needs to be encapsulated. Lastly, the security expert specifies security policies to fulfil the security requirements.

6.3.1 *Design of the secured pervasive computing application*

First, the designer of the pervasive computing applications needs to identify the required entities. Then he gives the descriptions of those entities by using DiaSpec. Listing 18 shows the taxonomy of the three pervasive computing applications of the scenario (see Section 3.1).

```
device Devices{
   attribute identity as String;
   attribute location as Location;
}

device TemperatureSensor extends Devices{
   source temperature as Float;
}

device SmokeSensor extends Devices{
   source smoke as Boolean;
}

device MotionDetector extends Devices{
   source intrusion as Boolean;
}

device SmartPhone extends Devices{
   source userRequest as Request;
}

device Sprinkler extends Devices{
   action OnOff;
}

device Alarm extends Devices{
   action OnOff;
}

device Light extends Devices{
   action OnOff;
}

device DoorLock extends Devices{
   action DoorAction;
}

device logger {
   attribute fileLocation as String;
   action LogAccessEvent;
}
/* Description of the supported actions*/
action OnOff {
    on();
    off();
}

action DoorAction {
    open();
    close();  }

action Log{
    LogAccessEvent(event as String);
}
```

Listing 18: The taxonomy of the fire detection application

The security expert analyzes both the security requirements and the taxonomy, then he decides which devices are user devices, and which actuating devices require security policy monitoring. We have two criteria for identifying the user devices which are enriched to represent the concept of subject. Firstly, these devices must have sensing capabilities. Secondly, these devices can receive direct orders from users. The actuating devices which can be encapsulated by the concept object must have the actuating capabilities. And their action can be activated by users. Then the security expert tags these devices with policy enforcement points and authentication enforcement points.

In this scenario, based on the description of the pervasive computing applications and the security requirements, the security expert identifies that device smart phones are user devices, and device door locks and lights are actuating devices. Because smart phones can interact with users, door locks and lights can be activated or deactivated by users. The security expert tags the smart phone with the authentication enforcement point. He tags door lock and light with the policy enforcement point. Listing 19 shows how we use the security enforcement point annotations to tag devices.

```
device Light extends Devices{
   action OnOff;
       security{ require PEP on action OnOff; }
}

```

```
device DoorLock extends Devices{
   action DoorAction;
       security{ require PEP on action DoorAction; }
}

device SmartPhone extends Devices{
   source userRequest as Request;
       security{ require AEP on source userRequest;}
}
```

Listing 19: Secured devices in the taxonomy

Then the designer of the pervasive computing system describes the architecture of the pervasive computing applications. Listing 20 shows the architecture description of the remote control application with DiaSpec.

```
context DoorContext as Request{
source request from SmartPhone;
}

controller DoorController{
context UserContext;
action DoorAction on Door;
action Log on Logger;
}

controller LightController{
context UserContext;
action OnOff on Light;
}
```

Listing 20: The architecture description of the remote control application

Listing 21 shows the architecture description of the fire detection application. In this application, we could notice that the entity context OnFire can produce the context information which indicates if the building is on fire. The security requirement also requires the security policies to be aware of this context information to determine whether a user can open a door or not. The security expert tags the context OnFire with the security information collecting point annotation (Line 4 of listing 21). This annotation indicates that this context provides context information named OnFire with type boolean. This context information can be used in the conditions of policy rules .

```
1 context OnFire as boolean{
2 source temperature from TemperatureSensor;
3 source smoke from SmokeSensor;
4    security{require SICP on context OnFire;}
5 }
6
7 controller FireController{
```

```
context OnFire;
action OnOff on Alarm;
action OnOff on Sprinkler;
action Log on Logger;
}
```

Listing 21: The architecture description of the fire detection application

Listing 22 shows the architecture description of the intrusion detection application in DiaSpec. This application detects an intrusion. The security requirement requires the security policies to be aware of the intrusion. As a result, the security expert tags the context Intrusion with the security information collecting point annotation. This annotation indicates that this context provides context information named Intrusion with type boolean. This context information can be used in the conditions of policy rules.

```
context Intrusion as boolean{
source intrusion from MotionDetector;
   security{require SICP on context Intrusion;}
}

controller IntrusionController{
context Intrusion;
action OnOff on Alarm;
action Log on Logger;
}
```

Listing 22: The architecture description of the intrusion detection application

6.3.2 *Specifying security policies*

After the design of the pervasive computing applications with security enforcement. The security expert specifies the security policy. First he needs to specify the basic entities such as location and role. Based on these entities, he could write positive and negative authorization rules.

6.3.2.1 *Location definition*

Based on the description of the location in Figure 13, we can define the location as shown in Listing 23.

```
Locations{ Building A29 {
              Floor F2{
                Hallway East {
                     Room R1;
                     Room R2;
                     Room R3;
```

```
                    Room R4;
                     }  } }
        }
```

Listing 23: The location definition

6.3.2.2 *Role definition*

Based on the description of the scenario, we have five kinds of users: professor, student, intern, phd student and janitor. Interns and phd students are the special students. Listing 24 presents the role definition of this scenario.

```
Roles {
       Role professor;
       Role student;
       Role intern inherit student;
       Role phd_student inherit student;
       Role janitor;}
```

Listing 24: The role definition

6.3.2.3 *Actions and the type of the objects*

The taxonomy with policy enforcement point indicates which kind of devices represents the concept object. The name of the entity description represents the object type. For instance, Lines 1 to 9 of Listing 19 is the description of device light and door lock with a policy enforcement point. As a result, in this scenario, the type of the objects consists of `Light` and `DoorLock`.

The taxonomy also provides the description of the supported actions. We use the name of the action and method to generate the action which is used in policy rules. For each method, we generate an action. For instance, The declarations in lines 43 to 50 of listing 18 define action `DoorAction` and `OnOff`. Based on these declarations, the set of actions consists of DoorAction_open, DoorAction_close, OnOff_on and OnOff_off.

The definition of action and the type of the objects are provided by the taxonomy. The security expert does not need to specify them.

6.3.2.4 *Policy rule specifications*

Based on the security requirement descriptions in Section 3.2, the security expert specifies policy rules. Listing 25 shows an example of policy rules.

```
1  //Rules for room R1
2  Permission(professor, DoorAction_open, DoorLock) if subjectId ==
       Alice and targetLocation == A29/F2/East/R1 and Intrusion ==
       false;
3  Permission(professor, DoorAction_close, DoorLock) if subjectId ==
       Alice and targetLocation == A29/F2/East/R1 and Intrusion ==
       false;
4  Permission(professor, OnOff_on, Light) if subjectId == Alice and
       targetLocation == A29/F2/East/R2;
5  Permission(professor, OnOff_off, Light) if subjectId == Alice and
       targetLocation == A29/F2/East/R2;
6  //Rules for room R2
7  Permission(professor, DoorAction_open, DoorLock) if subjectId ==
       Bob and targetLocation == A29/F2/East/R2 and Intrusion ==
       false;
8  Permission(professor, DoorAction_close, DoorLock) if subjectId ==
       Bob and targetLocation == A29/F2/East/R2 and Intrusion ==
       false;
9  Permission(professor, OnOff_on, Light) if subjectId == Bob and
       targetLocation == A29/F2/East/R2;
10 Permission(professor, OnOff_off, Light) if subjectId == Bob and
       targetLocation == A29/F2/East/R2;
11 //Rules for room R3
12 Permission(phd_student, DoorAction_open, DoorLock) if
       targetLocation == A29/F2/East/R3 and Intrusion == false;
13 Permission(phd_student, DoorAction_close, DoorLock) if
       targetLocation == A29/F2/East/R3 and Intrusion == false;
14 Permission(phd_student, OnOff_on, Light) if targetLocation ==
       A29/F2/East/R3;
15 Permission(phd_student, OnOff_off, Light) if targetLocation ==
       A29/F2/East/R3;
16 //Rules for room R4
17 Permission(student, DoorAction_open, DoorLock) if targetLocation
       == A29/F2/East/R4 and Intrusion == false;
18 Permission(student, DoorAction_close, DoorLock) if targetLocation
       == A29/F2/East/R4 and Intrusion == false;
19 Permission(student, OnOff_on, Light) if targetLocation ==
       A29/F2/East/R4;
20 Permission(student, OnOff_off, Light) if targetLocation ==
       A29/F2/East/R4;
21 //Rule for fire emergency
22 Permission(subRole, DoorAction_open, DoorLock) if Onfire == true;
23 //Rules for janitor
24 Permission(janitor, DoorAction_open, DoorLock);
25 Permission(janitor,  DoorAction_close, DoorLock);
26 //Rule for denying all undefined requests
27 Prohibition(_, _, _);
```

Listing 25: Security policy rules to fulfil the security requirements

The policy rules in lines 2 to 5 of listing 25 represent that professor Alice is allowed to access to room R1 and use the light (fulfil the first requirement description in Section 3.2). In the condition, the last statement (i.e. Intrusion == false) defines that the access permission is given if no intrusion is detected. The policy rules in lines 7 to 10 assign privileges to professor Bob to access his office and use lights. Rules in lines 12 to 15 define all users with role `phd_student` are allowed to access to room R3 if no intrusion has been detected (fulfil the third requirement description). Rules in lines 17 to 20 define all student are allowed to access to room R4. As role intern and `phd_student` inherit rights from role student (lines 4 to 5 of Listing 24), all interns and phd-students can access to room R4 and use lights. Line 22 gives the rule which defines all members can open all doors in case of fire emergency. In this rule, we use variable `subRole` without any constraint which represents all possible roles. Line 27 gives the default rule which denies all undefined requests.

6.4 SUMMARY

In this chapter, we presented the architecture of our approach. Then, we explained how to apply our approach in DiaSuite at the design stage. We used the applications which we introduced in the case study (Chapter 3) to illustrate how to design a secured pervasive computing applications by using our approach.

7 IMPLEMENTING THE SECURED PERVASIVE COMPUTING APPLICATIONS

In this chapter, we first present how we enrich the generated programming framework to embed the security mechanism into pervasive computing applications. Then we present the basic entities of the security management. Lastly, we present how to implement the applications in the case study based on the design specification which we described in Section 6.3.

7.1 DEVELOPMENT OF SECURED PERVASIVE COMPUTING APPLICATION

DiaSuite provides a compiler, named DiaGen which can generate a Java programming framework based on the DiaSpec specification. We enriched the generated framework with the code which can realize the functionalities of the required concepts (subject, object and security related context). The programming framework is dedicated to guide developers. It contains an abstract class for each entity declaration (devices, context and controller). These abstract classes provide methods to support the development. Implementing a DiaSpec-declared entity is done by subclassing the corresponding generated abstract class. To enforce the security policies in pervasive computing applications, we create new classes in the programming framework and enrich the generated abstract classes.

7.1.1 *Security enforcement points*

To realize the functionalities of the concept subject, object and security related context, we need to establish communication channels between secured pervasive computing applications and security management. We introduce class SecurityEnforcePoint in the generated framework. This class establishes communication between secured pervasive computing applications and security management by using the request/decision mechanism. Listing 26 shows the code fragment of this class. This class implements protocol SSL V3.0 to communicate with the security management. To enable the SSL communication, the developer needs to create the keystore to store the certificate which the security management provided. This class provides three methods: `authenticate`, `authorize` and `securityInformation`. The method `authenticate` takes the identity of the user and his proof, then forms the authentication request and sends the request to the security decision point. Lastly, this method returns the decision based on the results which are sent back by the security decision point. The method *authorize* is responsible for forming the authorization request and return decisions. The method *securityInformation* is responsible for sending the security relevant information to the *security information repository*.

```
  public class SecurityEnforcePoint{
        protected boolean authenticate(String userId, object
            proof)
                 {...}
        protected boolean authorize(String userId, String
            action, String targetId)
                 {...}
        protected void securityInformation
              (String informationName, Object
                  securityInformation)
                 {...}
     }
```

Listing 26: The code fragment of class SecurityEnforcePoint

7.1.2 *Enrich user device*

We use authentication enforcement point to encapsulate user device (sensing device in DiaSpec) with the concept of subject. Sensing devices in DiaSpec collect raw data from the pervasive computing environment and send them to context. The main functionality consists of sensing data and sending data.

The concept subject requires three functionalities which consists of forming the authentication requests, sending the requests to the authentication decision module and waiting for the decision of the authentication decision module. After the authentication, it creates a credential with subject's identity and the validity of the credential.

To encapsulate the user devices with the concept subject. We enriched the generated abstract class of user devices. The abstract class AbstractSmartPhone depicted in Listing 27 are generated based on the DiaSpec specification showed in Listing 19.

```
public abstract class AbstractSmartPhone{
...
private Credential userCredential=null;

public void logIn(String userId, Object proof){
  if (SecurityEnforcePoint.authenticate(userId, proof))
7          userCredential= new Credential(userId,ExpiryTime);
8      else throw new SecurityException("Unauthorize authentication
              request ...");
9  }
10   public void logOut(){
11     userCredential=null;
12 }
13   protected void publishRequest(Request newRequestValue ) {
14         if ((userCredential!=null) && !userCredential.isExpired())
15          { getProcessor().publishValue(getOutProperties(),
                  "request",
                  newRequestValue.setCredential(userCredential)); }
16          else throw new SecurityException()
17         }
18        ...
19   }
```

Listing 27: The code fragment of abstract class AbstractSmartPhone

We added an object `userCredential` (line 3), two methods `logIn` (lines 5 to 8) and `logOut` (lines 10 to 12) and enriched the method `publishRequest`. The method `logIn` is responsible for authenticating the subjects who are using these devices. The method `logOut` allows user to log out.

The method `publishRequest` is originally generated in DiaSpec programming framework which allows the device `SmartPhone` to send his source data to the context DoorContext. After we enriched the method, the method checks first if a user has the credential to send the source. If the user has no credential or his credential is expired, the system raises a security exception. If the user has the valid credential, then the method tags the user requests with the user's credential and sent them (lines 13 to 18) to the dedicated context.

7.1.3 *Enrich actuating device*

We use policy enforcement point to encapsulate actuating devices with the concept of object. Actuating devices receive orders from controllers, then execute actions based on these orders.

The concept object requires two functionalities which consists of forming the authorization request and sending the requests to the policy decision module and waiting for the decision of the policy decision module.

For instance, the abstract class `AbstractDoorLock` are generated to support the development of the device `DoorLock`. Listing 28 shows a fragment of the class `AbstractDoorLock` based on the DiaSpec description showed in Listing 19. The method `orderCalled` is generated as an interceptor which can block the order "open/close" sent by the controller `DoorController`. We enriched this method to archive the required functionalities. This method first forms an authorization request and sends it to the security management by calling method in the class `SecurityEnforcePoint`. If the decision is positive, then the dedicated action can be executed. Otherwise the action can not be executed, and the system raises a security exception (lines 8 and 14 in listing 28).

```
1  public abstract class AbstractDoorLock{
2  ...
3  public abstract void open();
4  public abstract void close();
5  public final Object orderCalled(..., RemoteServiceInfo source,
       String orderName,...)
6       throws Exception {
7            if (orderName.equals("open")) {
8                 if(SecurityEnforcePoint.authorize(source.getId(),
                      "open", this.Id))
9                 {open();
10                 return null;}
11           else throw new SecurityException("Unauthorize request
                 ... ");
12
13      } else if (orderName.equals("close")) {
14        if(SecurityEnforcePoint.authorize(source.getId(),
              "close", this.Id))
15             {close();
16              return null;}
17        else throw new SecurityException("Unauthorize request
              ... ");
18      } else
19    throw new InvocationException("Undefined method name " +
          orderName);
20 }
21 ...
22 }
```

Listing 28: The code fragment of abstract class AbstractDoorLock

7.1.4 *Enrich context*

We use security information collecting point to encapsulate context with the concept of security related context. Contexts receive raw data from sensing devices and refine them to produce more abstract data (i.e. context information). Then they send the context information to dedicated controllers or contexts. As a result, the main functionality consists of refining data and sending data. The main functionality of the concept security related context is to send the context information to the security management.

```
public abstract class AbstractOnFire extends Service{

```

```
3    public void valueReceived(RemoteServiceInfo source, String
         eventName, ... ) {
4        if (eventName.equals("smoke")) {
5                            ...
6           FireContextIndexedValuePublishable  mayPublish =
                onSmokeFromSmokeSensor(smokeFromSmokeSensor,
                getContext, discover);
7             //Security information collecting point sent the
                  context information to Security information
                  repository
8             SecurityEnforcePoint.sendSecurityInformation("Onfire",
                  mayPublish.getValue());
9
10             if (mayPublish != null && mayPublish.doPublish())
11               setFireContext(mayPublish.getValue() ,
                     mayPublish.getLocation() );
12     }
13        if (eventName.equals("temperature")) {
14                       ...
15          FireContextIndexedValuePublishable  mayPublish =
16             onTemperatureFromTempSensor(temperatureFromTempSensor,
                   getContext, discover);
17           //Security information collecting point sent the context
                 information to Security information repository
18             SecurityEnforcePoint.sendSecurityInformation("Onfire",
                   mayPublish.getValue());
19
20             if (mayPublish != null && mayPublish.doPublish())
21               setFireContext(mayPublish.getValue() ,
                     mayPublish.getLocation() );
22          }
23     }
24     public abstract
25          FireContextIndexedValuePublishable
26       onSmokeFromSmokeSensor( ... );
27
28     public abstract
29          FireContextIndexedValuePublishable
30       onTemperatureFromTempSensor( ... );
31
32     }
```

Listing 29: The code fragment of generated abstract class AbstractOnFire

For instance, the abstract class `AbstractOnFire` are generated to support the development of context `OnFire`. Listing 29 shows a fragment of the generated class `AbstractOnFire` based on the DiaSpec description showed in Listing 21. We highlight three methods in this class. The method `valueReceived` is called, when a new event has been received by the context Onfire. Based on the name of event, it calls method `onSmokeFromSmokeSensor` or `onTemperatureFromTempSensor` (lines 4 to 6 and lines 13 to 16 of Listing 29).

These two methods are generated to refine the raw data submitted by devices and produce context information. Then method `valueReceived` sends this context information to other contexts or controllers. We enriched the method `valueReceived` to send the context information to the security information repository. To realize this functionality, this method calls the method `sendSecurityInformation` in class `SecurityEnforcePoint` (lines 8 and 18 of listing 29).

7.2 DEVELOPMENT OF SECURITY MANAGEMENT

In this section, we present the basic entities in security management. These entities are implemented by using Java. For the sake of clarity, we do not show the Java code, we only show the functionalities of these entities.

The **security management** contains a security information repository (SIR_{sm}), an authentication decision module (ADP_{sm}), a policy engine (PEP_{sm}), a policy decision module (PDP_{sm}), a policy repository (PR_{sm}) and an administration console (AC_{sm}). Figure 17 shows the functionalities of these entities and their relationship.

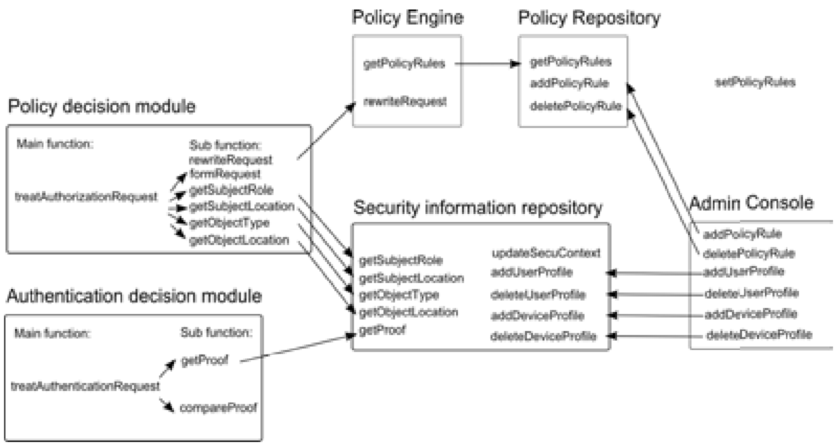


Figure 17: Entities in security management.

The **security information repository** stores security related information (e.g. subject profile, device profile and context) and offers functions to access or edit these information (see figure 17): to get the role of a specific subject, to get its location, to get the type of a specific object which the subject wants to access to, to add new subject profiles, to deletes subject profiles, to add new device profiles, to delete device profiles.

The **authentication decision module** gets the registered proof of the specific subject's identity from the security information repository, and compares it with the proof submitted by the subject, and returns true if they are identical, false otherwise.

The **policy decision module** handles the access control/authorization requests. It contains several auxiliary functions to respectively get the role of the specific subject, the location of the specific subject, the location of the object which the subject wants to access.

To decide whether the authorization request is permitted or denied, the **policy engine** retrieves policy rules from the **policy repository** that may contain several policies (but for simplicity we will consider only one here), and applies the policy rules to rewrite the authorization request with an appropriate strategy (defined in the policy) until getting an irreducible form. In general, an analysis of the policy rules and strategy ensures that there is always a unique irreducible form which is permit or deny (or some other chosen decision). We will give more details in Chapter 8.

An access control/authorization request can be seen as a term whose constants are values in the security information repository (actual subject's identity, actual object, contextual value,...). To compute a decision, the policy rules are matched against the request and the rule variables are instantiated by constant values, giving a matching substitution σ. If C is the constraint associated to the rewrite rule, $\sigma(C)$ is checked for satisfiability. If it is satisfiable, the rule can be applied. Otherwise the application of this rule fails and another one (according to the chosen strategy) can be tried. Again, an analysis of the policy rules and strategy guarantees that at least one rule applies if the request is different from permit or deny.

The **administration console** allow administrators to edit the subject profiles, device profiles and policy rules. It contains several functions to realize these functionalities (see figure 17) : to return the list of policy rules which are stored in policy repository, to add new policy rules, to delete policy rules, to add new subject profiles, to delete subject profiles, to add new device profiles, to delete device profiles.

7.3 IMPLEMENTING THE APPLICATIONS IN THE CASE STUDY

In this section, we use the scenario which we described in Section 3.1 to illustrate how to implement a secured pervasive computing application by using our approach.

At development stage, we introduce a new kind of developers which is the application developers. They are responsible for implementing the entities of the pervasive computing application by using the generated framework. The security expert needs to generate a RSA key pair and publishes the certificate of the public key. As we presented in Section 7.1.1, the functionality of the security enforcement points has been integrated into the generated framework. So most of the functionalities are transparent to the application developers.

7.3.1 *Implementation of the secured pervasive computing application*

7.3.1.1 *Implementing context*

```
 1 public class OnFire extends AbstractOnFire{
 2   ...
 3 @Override
 4 public FireContextIndexedValuePublishable
       onSmokeFromSmokeSensor(SmokeFromSmokeSensor smoke,
 5 GetContextForSmokeFromSmokeSensor getContext,
 6 DiscoverForSmokeFromSmokeSensor discover)
 7 {
 8   Location location = smoke.indices().location();
 9
10   TemperatureSensorComposite temperatureSensor =
         discover.TemperatureSensors().whereLocation(location);
11
12 if(temperatureSensor.getTemperature()>50)
13
14 return new FireContextIndexedValuePublishable(fire, location,
       true);
15 }
16 ...
17 }
```

Listing 30: A developer-supplied implementation of the OnFire context in Java

For instance, the code fragment in Listing 30 presents the implementation of the `OnFire` context declaration in Listing 21. This class extends the generated abstract class `AbstractOnFire` (not shown here, in Listing 29). When a smoke sensor sends a new smoke value to the context `OnFire`, the method `onSmokeFromSmokeSensor` are called. In this example, the method checks also the temperature of the area where smoke is detected to make sure there is a fire (lines 8 to 11). Now the context `OnFire` has the new security relevant context information, it needs to send them to the security information repository. There are generated methods in the abstract class AbstractOnFire which can send the context information to the security information repository (see line 8 and line 18 of Listing 29).

7.3.1.2 *Implementing the user devices*

To implement the user devices, the application developer needs to do more work. When he extends the abstract class AbstractSmartPhone of the generated framework, he obtains a Java class *SmartPhone*. He needs to map this Java class to a real device. In this scenario, we choose a smart phone with the Android OS as the concrete device.

```
1  public class HttpAuthenticationActivity extends Activity {
2          ...
3      @Override
4      public void onCreate(Bundle savedInstanceState) {
5          ...
6      button.setOnClickListener(new View.OnClickListener() {
7
8      @Override
9      public void onClick(View v) {
10     String login;
11     String password;
12     login = edit1.getText().toString();
13     password = edit2.getText().toString();
14     SmartPhone client = new SmartPhone();
15     boolean decision = client.logIn(login, password);
16
17      if (decision == true) {
18          // Jump to GUI of remote control of the door and light
19         Intent myIntent = new
               Intent(v.getContext(),ClientInterface.class);
20             myIntent.putExtra("UserId", login);
21             startActivityForResult(myIntent, 0);
22                 }
23      else { ... } }
24         });
25     }
26 }
```

Listing 31: A developer-supplied implementation of the smart phone in Android 2.2

Listing 31 shows a developer supplied implementation of the smart phone in Android 2.2. This class provides a graphic user interface which allows users to enter his identity and his proof (line 12 and line 13). In this case, the proof is a password. Then this class calls the method *logIn* of class *SmartPhone* which extends the class *AbstractSmartPhone* of the generated framework (line 14 to line 15 of Listing 27). The code of the class *AbstractSmartPhone* is shown in Listing 27. If the authentication succeeds, this class jump to a new GUI which allows users to control the devices (line 17 to line 22 of Listing 27). This step of the development needs to be done manually by the application developer, because the concrete implementation of each device is different due to the different variety of the hardware and software. But they can all benefit the methods such as logIn and logOut of the generated class. This can facilitate the development of the devices.

7.3.1.3 *Implementing the action devices*

To implement the action devices, the application developer needs to extend the generated abstract class of the framework. Then he maps this Java class with real devices. Listing 32 is an example of the door lock implementation which extends the generated abstract class showed in listing 28. As we enriched the framework with the functionalities of the concept object. The enforcement of the security policies is completely transparent to the application developers. They do not need to worry about how to enforce security policies.

```java
public class DoorLock_X10 extends AbstractDoorLock{

   ... // defines x10Ctrl and x10Addr

@Override
public void close() {
   x10Ctrl.addCommand(new Command(x10Addr, Command.OFF));
   }
@Override
public void open() {
   x10Ctrl.addCommand(new Command(x10Addr, Command.ON));
   }
}
```

Listing 32: An example of door lock implementation

7.3.2 *Implementation of the security management*

The implementation of the security management is totally automatic from the developers' point of view. The only work for developers is to configure the certificates to secure the communication and databases to store the profile of users and devices.

The crucial component of our architecture is the policy engine. Our policy engine is implemented by using the term rewriting system. In Section 5.2.3.2, we already presented how to use the term rewriting system to evaluate an authorization request. The generation of the term rewriting system is ensured by our generator. The rewriting engine is ensured by the tool Tom [13]. The other components (e.g. policy decision module, authentication decision module) in our architecture are also generated by our generator.

7.4 SUMMARY

This chapter presented how we enriched the generated programming framework of DiaSpec to embed the security mechanism into pervasive computing applications. The enriched programming framework embeds security policies seamlessly. Then we presented the basic entities of the security management. Lastly, we used the scenario in the case study to illustrate how to implement the pervasive computing applications based on the design specification which we described in Section 6.3. As we mentioned before, a successful deployment of security policies requires verification and test support. The next chapter presents how we provide support at verification and test stage.

8 VERIFICATION AND TEST, DEPLOYMENT AND MAINTENANCE

In this chapter, we present how to verify and test security policies in a pervasive computing environment. In Section 8.1, we first present several properties of the security policies which we want to verify before the deployment. Then we present how to detect conflicts between policy rules. In Section 8.2, we give a brief introduction about the simulator which we use to test our security policies. In Section 8.3, we present how our approach supports the change of secured pervasive computing systems and security policies.

8.1 VERIFICATION OF THE SECURITY POLICIES

The key requirement for the successful deployment of security policies is the availability of tools for analyzing policies. Kami Vaniea et al. [92] said *"Managing large sets of access-control rules is a complex task for security administrators. Each addition, deletion or modification of a rule may cause potential and unknown side effects ranging from rule con icts to security breaches"*.

According to Kami, many properties (e.g. conflict, termination, completeness and conformity) need to be verified. In this thesis, we concentrate on how to detect conflicts between policy rules, it is the most vital property of a security policy. We also give hints to check completeness and termination.

8.1.1 *Policy analysis*

For the sake of clarity, in our work, we use the terminologies and classification of the approach, named *Exam* [59]. It is useful to highlight that there are two types of policy analysis: policy property analysis and policy similarity analysis [6, 7, 11]. Policy property analysis refers to the verification of a given property on a single policy. Policy similarity analysis refers to a comparison among two or more policies. In this thesis, we suppose there is only one policy in a pervasive computing system. As a result, we only address the policy property analysis.

The authors of Exam identify three main categories of policy property analysis: policy metadata analysis, policy content analysis and property analysis.

8.1.1.1 *Meta-data of security policy*

The meta-data of the security policies such as author of the security policy, date of the creation and date of the modification are important for security administrators. For instance, if a policy rule leads to a security breach, the meta-data of this rule can help administrator to find out since when the security breach exists.

8.1.1.2 *Content of security policy*

According to the authors of Exam, the content of a security policy such as the number of rules in the policy, the total number of attributes referenced in the policy, the privilege of a specific role, can help the security administrator to maintain the security policy. For instance, checking the privilege of a specific role verifies if there are rules in the policy which fulfil a specific requirement.

8.1.1.3 *Property of security policy*

A security policy evaluates a given request, in a given environment context, and is expected to return one and only one decision. With a given request, if a security policy can not return a decision, then this policy is incomplete. If a security policy returns more than one decision, then this policy has conflicts and is not consistent.

8.1.2 *Property verifications*

As we use a term rewriting approach [35] to implement our policy engine, we leverage the verification power of this framework. As our generated rewrite rules have some particularities (e.g. decisions are constants). Some verification techniques need to be adapted to better fit the particularities of our approach. In this work, we focus on how to detect conflicts between rewrite rules.

To be self-contained, we need to introduce some basic terminologies.

The derivation tree or evaluation tree of a request in a term rewriting system (TRS) can be expressed as a directed graph (O, S) in which nodes represent terms and edges represent rewrites. For instance, if the term *a* can be rewritten into *b*, this is represented by using arrow notation: $a \rightarrow b$. Intuitively, this means that the corresponding graph has a directed edge from a to b. The nodes in O are called objects, the oriented edges in S are called steps.

If there is a path between two graph nodes (for instance c and d), then the intermediate nodes form a reduction sequence. For instance, if $c \to c_1 \to c_2 \to ... \to d_n \to d$, then we can write $c \overset{*}{\to} d$

De nition 4. *(Con uence and local con uence). An element* $a \in O$ *is said to be con uent if for all* $b, c \in O$ *with* $a \overset{*}{\to} b$ *and* $a \overset{*}{\to} c$ *there exists* $d \in O$ *with* $b \overset{*}{\to} d$ *and* $c \overset{*}{\to} d$.
An element $a \in O$ *is said to be locally (or weakly) con uent if for all* $b, c \in O$ *with* $a \to b$ *and* $a \to c$ *there exists* $d \in O$ *with* $b \overset{*}{\to} d$ *and* $c \overset{*}{\to} d$.

If all elements in O are (locally) confluent, we say that the relation $\overset{*}{\to}$ induced by the term rewriting system is (locally or weakly) confluent. For short, we simply say that the term rewriting system is (locally or weakly) confluent.

For terminating term rewriting systems, (local) confluence is checked by computing superposition between rules obtained by unification of the left-hand sides.

De nition 5. *(Unification). An equation is a formula of the form* $s == t$ *where s and t are terms. An unification problem is a conjunction of equations denoted* $P = (s1 == t1 \wedge ...sn == tn)$.

- *Two terms s and t are unifiable if there is a substitution* σ *such that* $\sigma(s) = \sigma(t)$. σ *is called an unifier.*
- *A substitution* σ *is a solution for P if* $\sigma s_i == \sigma t_i$ *for i = 1, . . . , n. The set of all unifiers of an unification problem P is denoted* $U(P)$.

Anderson [35] defines a security policy, $\wp$, as a five-tuple(Σ, D, R, Q, ζ) (see Section 2.3.3). He addressed three properties: consistency, termination and decision completeness.

CONSISTENCY A security policy is consistent if it computes at most one access decision for a given input request.

De nition 6. *(Consistency). A policy* $\wp = (\Sigma, D, R, Q, \zeta)$ *is consistent if for every query* $q \in Q$, *q rewrites to at most one result with the rules in* R *applied with the strategy* ζ.

This definition means that for every query evaluation, a deterministic result is computed by rewriting.

With the strategy *universal*, a policy $(\Sigma, D, R, Q, \mathrm{universal}(R))$ is consistent if the set of rewrite rules R is confluent, and all elements in the decision set D are irreducible.

In our case, the consistency is easy to check. Let us note first that our policy rules rewrite an access request into a decision in one step, and the decisions are constant and irreducible which can be easily checked. Let us examine more precisely how to detect conflicts among rules.

CONFLICT Suppose we have two rules R1 and R2 in a security policy of the form:

- R1: $l_1 \rightarrow r_1$ if c1
- R2: $l_2 \rightarrow r_2$ if c2

where r_1 and r_2 are constants

Suppose that these two rules are applied in an environment context denoted by Con.

De nition 7. *(Con ict). R1 and R2 are in con ict, if* $\exists\sigma, \sigma(l_1) = \sigma(l_2)$*,* $r_1 \neq r_2$*, and* $\sigma(c_1), \sigma(c_2)$ *are true in the environment* Con.

In other words, if the left-hand sides of R1 and R2 are unifiable, and the right-hand sides of R1 and R2 are different, and the constraints c1 and c2 instantiated by the unifier σ are true in the environment context Con, then we can say that R1 and R2 are in conflict.

For instance, let us consider two policy rules written in our policy specification:

1. r1: $\mathrm{Permission}(sr1, a1, ot1)$ if c1
2. r2: $\mathrm{Prohibition}(sr2, a2, ot2)$ if c2

Suppose SR is a set of *subject roles*, A is a set of *actions*, OT is a set of *object types*, C is a set of *constraints* and $sr1, sr2 \in SR$; $a1, a2 \in A$; $ot1, ot2 \in OT$; $c1, c2 \in C$. These two rules generate two rewrite rules as follows:

1. R1: $req(sub(sid1, sr1, sloc1), a1, obj(oid1, ot1, oloc1)) \rightarrow d1$ if c1
2. R2: $req(sub(sid2, sr2, sloc2), a2, obj(oid2, ot2, oloc2)) \rightarrow d2$ if c2

Note that sid1, sr1, sloc1, a1, oid1, ot1, oloc1, sid2, sr2, sloc2, a2, oid2, ot2 and oloc2 are constants or variables. The left-hand sides of the policy rules have the same form. As a result, the unification problem is trivial and amounts to solve a system of simple equations (e.g. sid1==sid2, sr1==sr2, etc.). If there is a solution σ, we need to instantiate c1 and c2 by σ and check if c1 and c2 are true in the environment context Con.

TERMINATION Termination is crucial for rewriting based policies. To check the termination of term rewriting system, techniques have been developed [84].

De nition 8. *(Termination). A security policy* $\wp = (\Sigma, D, R, Q, \zeta)$ *is terminating if for every query* $q \in Q$*, all derivations of q in ζ are finite.*

In our case, our policy rules rewrite an access request into a decision in at most one step to a constant term. As a result, the termination problem is trivial.

COMPLETENESS

De nition 9. *(Completeness). A security policy* $\wp = (\Sigma, D, R, Q, \zeta)$ *is complete if for every query* $q \in Q$*,* $\exists d \in D$*, such that q rewrites to d with R applied with strategy ζ*

In other words, for any term in the set of requests, the evaluation of this term following the rewrite strategy defined by the policy will result in a term in the set of decisions.

There are existing approaches (e.g. [18, 44]) for checking the completeness of constrained term rewriting systems. In our case, to verify whether a security policy is complete is complicated at design time. To make sure a policy is complete, we must guarantee that for all possible requests, the policy can give back a decision. At design time, we do not have enough information to generate all possible requests. But subject roles, actions and object types are already defined. If there is a role, a type or an action who has never been used in policy rules, we can be sure this policy is not complete. In Section 8.1.3, we explain how to resolve the incomplete policies.

8.1.3 *Query language*

A tool for analyzing security policies should allow users to easily specify the properties which they want to analyze. Some approaches [58, 64, 69, 82] use Datalog-like language [60]. Anu Singh et al. use deductive spreadsheets to analyze security policies [87].

To allow administrators to easily analyze security policies, we propose a little query language. Listing 33 shows examples of policy meta-data queries. First query inquires about the author of a specific policy rule. It starts with keywords **show author of**, followed by the identity[1] of a policy rule. Second query inquires about the creation time of a policy rule.

1 In our approach, we attribute an unique identity to every policy rules

```
/*ask the author of a policy rule "r1"*/
show author of r1;

/*ask the time of creation of a policy rule "r1"*/
show CRtime of r1;
```

Listing 33: Example of policy Meta-data queries

Listing 34 shows examples of policy content queries. First query inquires about the privilege of a specific subject. It starts with keywords **get right of user**, followed by the identity of a user. The second query inquires about the privilege of a specific role. The third query inquires about the subjects which can execute action open on a specific object.

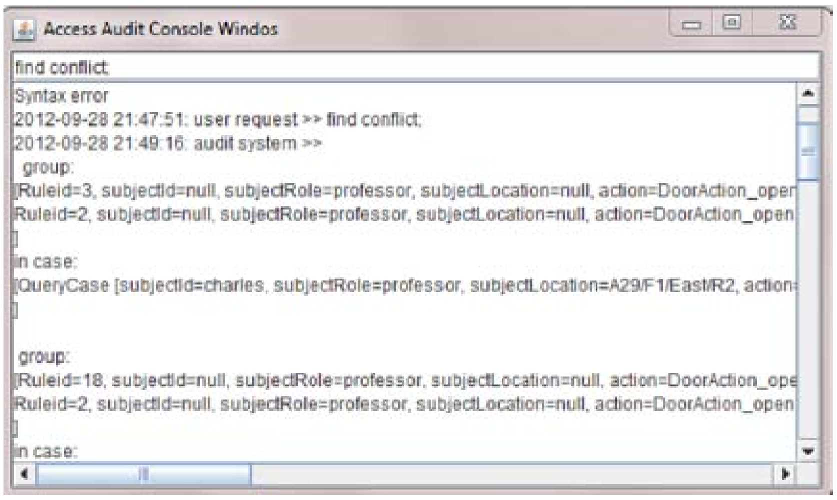
```
/*ask the privilege of subject "Alice" in policy*/
get right of user "Alice";

/*ask the privilege of professor in policy*/
get right of professor;

/*ask who can open door d1*/
who can open door d1;
```

Listing 34: Example of policy content queries

We also provide queries to find conflicts. Figure 18 shows the administration console for policy analysis. We first show the groups of policy rules which are in conflict. Then we give the situations in which the policy rules are in conflict.



```
find conflict
Syntax error
2012-09-28 21:47:51: user request >> find conflict;
2012-09-28 21:49:16: audit system >>
 group:
[Ruleid=3, subjectId=null, subjectRole=professor, subjectLocation=null, action=DoorAction_oper
Ruleid=2, subjectId=null, subjectRole=professor, subjectLocation=null, action=DoorAction_open
]
in case:
[QueryCase [subjectId=charles, subjectRole=professor, subjectLocation=A29/F1/East/R2, action=
]


 group:
[Ruleid=18, subjectId=null, subjectRole=professor, subjectLocation=null, action=DoorAction_ope
Ruleid=2, subjectId=null, subjectRole=professor, subjectLocation=null, action=DoorAction_open
]
in case:
```

Figure 18: The administration console for policy analysis

CONFLICT RESOLUTION To resolve the conflicts, we adopt in this work a very simple methodology inspired from XACML.

XACML uses the notion of evaluation strategy (e.g. first-applicable, permit-override, deny-override and only one applicable). The concept of *strategy* has been already included in security policy design based on rewrite system [35]. Strategies in rewrite systems define in which order rewrite rules are executed. In the specific context handled in this thesis, all policies use the strategy *first-applicable*. This means that the first rule which can be applied gives the final decision. In this strategy, the policy rule which has the lower rank has the higher priority. Indeed the language supports other strategies, which is especially useful when several strategies are composed. This is not the topics of this thesis.

INCOMPLETE POLICY RESOLUTION To resolve incomplete policy, we propose two default rules: "Permission(_, _, _)" and "Prohibition(_, _, _)". The default rule "Permission(_, _, _)" defines that which is not explicitly denied is permitted. The default rule "Prohibition(_, _, _)" defines that which is not explicitly permitted is denied.

8.2 TESTING SECURITY POLICIES IN A SIMULATED ENVIRONMENT

In Section8.1, we presented that conflicts between rules can be detected. However, we can not verify whether a policy fulfils the security requirement or not. As a result, it is helpful for the administrators to test whether certain security requirements are fulfilled or not. To overcome this problem, we introduce a 2D simulator DiaSim [23]. In this section, we present how we use DiaSim to test the behaviour of the security policies in a simulated pervasive computing environment.

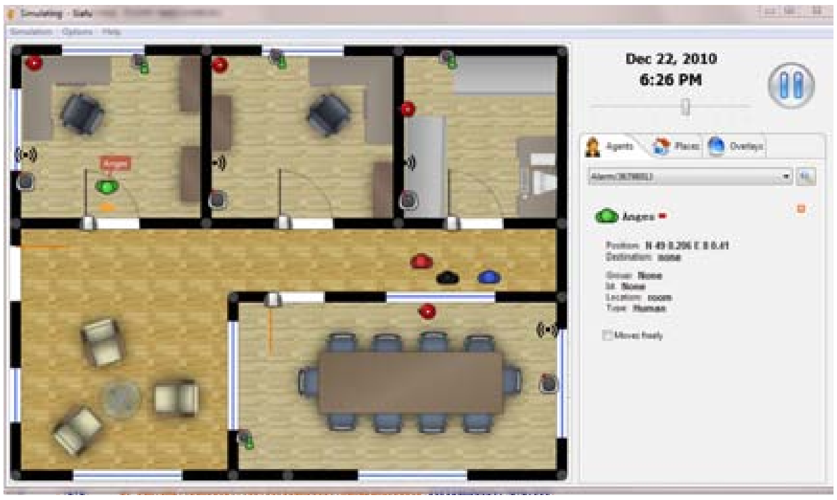


Figure 19: An example of a simulation in DiaSim

Figure 19 shows an example of the DiaSim simulator. The simulation renderer uses the same framework that we generate to deploy the secure pervasive computing system. The security concepts are already embedded inside the generated programming framework.

Suppose we want to verify the first security requirement which we give in Section 3.2 (i.e. Only professor Alice can access to his office). The events are triggered by moving emulated users into special zone. In this example, when we move Alice near to the emulated device door of room 1, the event "Alice wants to open the door of room 1" is triggered. When the simulated pervasive computing environment emits this event in the framework, the security mechanism is automatically triggered.

When the framework receives the event "Alice wants to open the door of room 1", the dedicated actuator door is invoked to execute action "open". When the actuator door receives this order, it triggers the security mechanism to verify whether Alice has the right to execute action "open" or not. If the decision of the security policy is positive, the actuator door orders the emulated door to print a message "open" in the simulation renderer. If the decision is negative, the simulation renderer prints a message "Alice does not have the right to access this service". In this case, we know that this policy does not fulfil the security requirement. The simulated pervasive computing environment also provides simulated context information. This allows us to test the behavior of security policies with the simulated pervasive computing environment.

This simulator has already been used in several secured pervasive computing applications to test the behaviors of security policy with hundreds of simulated users and devices.

8.3 DEPLOYMENT AND MAINTENANCE

In this section, we present how our approach facilitates the deployment and maintenance of secured pervasive computing applications. First, we present how we deploy a secured pervasive computing application. Then, we present how we maintain it.

8.3.1 *Deployment*

The deployment of a secured pervasive computing application can be divided into two parts: the deployment of pervasive computing applications and the deployment of security policies.

8.3.1.1 *Deployment of pervasive computing applications*

The deployment of pervasive computing application consists of building a back-end to connect the distributed entities and instantiating the declared entities of the taxonomy. The DiaSuite approach offers back-ends to support distributed system technology. The instantiation of the entities is done by instantiating the Java class of the programming framework. This is not relevant to our work, more information can be found in Section 6 of [24].

8.3.1.2 *Deployment of security policies*

The deployment of pervasive computing application consists of instantiating the user profiles and device profiles. We offer a GUI which allows developers to initialize and modify user and device profiles.

8.3.2 *Maintenance*

With time, the pervasive computing application may need to be modified to adapt to new situations. As a result, the security policy needs to be maintained to adapt the change of pervasive computing system. Even if the pervasive computing application does not change, new users or devices may enter to the pervasive computing environment.

8.3.2.1 *Change the pervasive computing application*

DiaSuite allows developers to add new entities, delete existing entities or changing the application design by changing the DiaSpec specification. Then the generator of DiaSuite generates a new programming framework for the changed pervasive computing application. The classes which extend the abstract classes in the old framework need to be modified or deleted to adapt to the new framework. It is beyond the scope of our work, more details are given in Section 6 of [24].

8.3.2.2 *Change the secured entities*

By adding or deleting the security enforcement annotations in DiaSpec specification, we create new secured entities (i.e. subject, object, security related context), or delete old entities at any moment. The generator generates dedicated code in the programming framework to realize the functionalities of these entities.

8.3.2.3 *Change the security policies*

The attribute declarations of the subject or object (e.g. role declaration, location declaration) can be changed by modifying the DiaSecur specification. The changed specification is used to generate a new signature. The deletion of an attribute may cause compilation errors. For instance, if a policy rule contains an attribute which has been deleted, this causes a compilation error. These errors need to be corrected by the human developers.

A policy rule can be added, deleted and modified at any moment in the DiaSecur specification. The enforcement of the change of policy rules requires no human intervention. The generator generates automatically dedicated rewrite rules to adapt to the change.

8.3.2.4 *Change the user and device profile*

The profile of users and devices can be modified at any moment. We propose a GUI to facilitate the workload of administrators. The change does not require to regenerate the security management.

8.4 SUMMARY

In this chapter, we presented how to verify and test security policies in a pervasive computing environment. In this work, we just check some basic properties. We need more sophisticated approaches to verify whether a security policy fulfils the security requirements or whether a security policy is safe under attack. We also presented how our approach supports the change of secured pervasive computing systems and security policies. Some changes can be done dynamically, some changes need to recompile the DiaSpec or DiaSecur specifications.

Part III

CONCLUSION

9 CONCLUSION

We have presented our policy specification language and the security mechanism to enforce the security policies. In this chapter, we give some overvall conclusions about this thesis.

Pervasive computing systems provide promising applications in different domains (e.g. smart home, assisted living, health care). On the other hand, it also leads to the emergence of new security challenges. To address these challenges, we propose a policy specification language based on term rewriting systems and a security mechanism to enforce the security policies in pervasive computing systems.

Pervasive computing environments contain a variety of heterogeneous computational entities (hardware or software). Pervasive computing applications use software infrastructures to orchestrate these computational entities and provide services to users in their daily activities.

Since developing pervasive computing applications involves several domains (e.g. networking, smart objects), the application developers need support to facilitate their work. We have presented several approaches which are dedicated to develop pervasive computing applications. Most of them focus on the functional challenges (e.g. dynamicity, heterogeneity and interoperability).

The advance of the pervasive computing systems makes people more and more dependant on the pervasive computing applications. The non functional properties such as security become more and more essential, because the security breaches of critical applications can put people and their property at risk.

We have presented several approached which are used to specify security policies in pervasive computing applications. They are focusing on the expressiveness of their language. Therefore, none of them addresses how to support developers to enforce the security policies in pervasive computing applications. Due to the nature of pervasive computing applications, the main challenge is to make security policies context sensitive.

To address all these challenges, we propose an approach which consists of a domain specific policy specification language and a security mechanism. Our approach provides support throughout the life-cycle of pervasive computing applications. The architecture of our security mechanism allows us to enforce our security policies in any pervasive computing applications without changing the implementation of the security policies. The security mechanism can collect and store the context information which are provided by the pervasive computing environments. To make the security policy context sensitive, the policy rules are parameterized with respect to context information; decisions are made with respect to context values drawn from the running pervasive computing environment.

We have embedded our approach in a tool based approach, named DiaSuite which is dedicated to develop pervasive computing applications. At the design stage, we enriched the design language DiaSpec with specific annotations. These annotations tag the basic entities of DiaSpec with security enforcement points. These annotations allow us to use more abstract concepts to encapsulate the basic building blocks of DiaSpec. At the development stage, based on the annotations, the generator produces the dedicated function for each tagged entities in the generated framework. At the verification and test stage, we provide a tool which allows system administrators to analyze the meta-data, the content and the property of security policies. We leverage the 2D simulator DiaSim to test security policies with simulated pervasive computing environments. At deployment and maintenance stage, our approach provides a GUI which allows administrators to manage security policies and the profile of users and devices.

10 FUTURE WORK

An interesting challenge is to analyze security policies within pervasive computing applications, because even if a security policy is complete and has no conflict, it does not mean the security policy fulfil the security requirements and is safe under attack. We want to use Petri net to model the pervasive computing applications by using the DiaSpec (i.e. taxonomy, architecture description of the pervasive computing application) and DiaSecur (i.e. user profile, location descriptions) declarations. Based on the pervasive computing application model and a given attack model (e.g. Ciphertext-Only Attack, Known-Plaintext Attack, Chosen-Plaintext Attack and Chosen-Ciphertext Attack), we can evaluate the knowledge of the attacker. By using model checking, we could verify which states the attacker can reach. Thus we could verify whether our pervasive computing application is secure under attack.

When an end user wants to access a service, if his request has been denied by security policies, this end user may want to know why he can not access this service. In this case, the feedback on the security policy is essential. For now, our approach can not give a specific reason when a request has been denied. Therefore, it would be useful to provide feedbacks to end users in our framework.

A further step is to add the concept of time and "special area" in our approach. Thus we could express obligations triggered by time (e.g. users have to change their password every three months). The "special area" is useful in many cases: for instance, a research institute has several research groups, located in different offices. The leader of research groups may want to administrate security policy inside his research group's offices. But there are also public spaces inside the research institute where the leaders do not have specific rights. With the notion of special area, we could delegate the right of administrator to *area administrator*. When a user enters a special area (e.g. a research group), the global role (e.g. student) will be changed into a special area role (e.g. group member) which has privileges only in this special area. For now, attributes and their data-types of object and subject are fixed. We want to allow developers to define their own attributes and data-types to adapt to other environments.

Part IV

APPENDICES

A SPECIFYING SECURITY POLICY WITH DIFFERENT APPROACHES

Suppose we are in a teaching facilities, we have two kinds of user professor and student. We suppose there is a printer named p1, professors can use this printer, and student can not use this printer.

A.1 REI

Listing 35 shows policy rule specification in Rei [52]. The first rule specifies that professor can print. The second rule specifies that student can not print.

```
has(Variable, right(printOnP1,(professor(Variable))))
has(Variable, prohibition(printOnP1,(student(Variable))))
action(printOnP1,printer-p1)
```

Listing 35: Policy rule specifications in Rei

Rei uses the basic structure *has(Subject, PolicyObject)* to grant privileges to user. *PolicyObject* could be *right(Action, Condition)*(positive decision) or *prohibition(Action, Condition)*(negative decision) The *professor(Variable)* and *student(Variable)* are constraints which means the subject must has the role professor or student. The third rule defines action *printOnP1* whose target object is *printer-p1*. We can identify key concept such as subject, target object, action, decision and constraint.

A.2 PONDER

List 36 shows policy rule specification in Ponder [33]. Ponder uses inst auth+ to express positive decision, and inst auth- to express negative decision. Every rule contains a subject, a target, an action and a when statement. Based on these two rules, we can identify key concept such as subject, target, action, decision and constraint.

```
inst auth+ Proprint{
   subject variable;
   target p1;
   action print();
   when  variable.role="Professor";
   }
inst auth- Proprint{
   subject variable;
   target p1;
   action print();
   when  variable.role="Student";
   }
```

Listing 36: Policy rule specifications in Ponder

A.3 XACML

Listing 37 shows policy rule specification in XACML 2.0 [38]. In XACML *Rule* is the most elementary unit. The main components of a rule are:

- **target**
- **effect**
- **condition**

The *target* contains *resources, subjects and actions*. The *effect* indicates the decision of the rule. Two values are allowed: *permit* and *deny*. *Condition* represents a Boolean expression that refines the applicability of the rule beyond the predicates implied by its target. Therefore, it may be absent.

```
<Rule RuleId="Rule01" Effect="Permit">
   <Description>
     professors can print on a printer named p1
   </Description>
<Target>
  <Subjects>
    <Subject>
      <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">
            Professor
          </AttributeValue>
            <SubjectAttributeDesignator AttributeId="role"
                 DataType="string"/>
      </SubjectMatch>
     </Subject>
  </Subjects>
  <Resources>
     <Resource>
       <ResourceMatch MatchId="string-equal">
          <AttributeValue DataType="string">
               p1
          </AttributeValue>
            <ResourceAttributeDesignator
               AttributeId="printer-name"
               DataType="string"/>
           </ResourceMatch>
     </Resource>
  </Resources>
  <Actions>
    <Action>
        <ActionMatch MatchId="string-equal">
            <AttributeValue DataType="string">
            print
            </AttributeValue>
            <ActionAttributeDesignator
               AttributeId="action-id"
               DataType="string"/>
        </ActionMatch>
    </Action>
  </Actions>
</Target>
</Rule>
```

Listing 37: Policy rule specifications in XACML

A.4 SPL

Listing 38 shows policy rule specification in SPL [75]. A policy rule in SPL is comprised of two logical binary expressions. The first one is used to establish the domain of applicability such as target name and action name. The other one is used to decide on the acceptability of the event.

```
rule01: ce.target.name="p1" & ce.action.name="print"
      :: ce.author.role="professor";
rule02: ce.target.name="p1" & ce.action.name="print"
      :: ce.author.role!="student"
```

Listing 38: Policy rule specifications in SPL

In the police rules, *ce* stands for current event. We can consider events as requests. The first part of rule01 means this rule is applicable for the event (request) which has target with name "p1" and action with name "print". The decision is given by the second part. If the author's role is professor then ce.author.role="professor" will return true (Permit), otherwise it will return false (Deny). Listing 39 shows an example of event definition given by SPL. SPL consider all entities are typed objects with an explicit interface by which their properties can be queried.

```
type object {
            string name; // The name of the object
            user owner; // The owner of the object
            object set groups;// The sets containing the object
            string homeHost; // The host where the user is defined
            }
type user extends object{
    rule set userPolicy; //  User private policies
}
```

```
type operation extends object{
    number ID; // operation Id
}
type event extends object{
     user author; // The author of the event
     object target; // The target of the event
     operation action; // The performed action
     number time; // The time instant
}
```

Listing 39: Example of event definition in SPL

B GRAMMAR OF DIASECUR

In this page, we present the grammar of our policy specification language (DiaSecur). The syntax which we chose is close to EBNF (Extended Backus-Naur Form) and is used by ANTLR[1].

```
document : typeDef
           policyDef
           EOF
         ;

typeDef : locationDef
          rolesDef
        ;

locationDef  : Locations {  building +   }
             ;

building : Building building_ref { floor +  }
         ;

floor : Floor floor_ref { hallway +  }
      ;

hallway : Hallway hallway_ref { room +  }
        ;

room : Room room_ref ;
     ;

rolesDef : Roles  { roleDef+ }
         ;

roleDef : Role ID ( inherit role_ref (, role_ref)* )? ;
        ;


policy_def :  Policy {
                          strategy
                          policyRules +
                       }
           ;

strategy : Use strategy strategyName ;
         ;

policyRules : permission | prohibition
            ;

```

1 ANother Tool for Language Recognition (ANTLR), is a parser generator that uses LL(*) parsing. http://en.wikipedia.org/wiki/ANTLR

```
permission : 'Permission' '(' ruleBody ')' ('if' conditions )? ';'
           ;

prohibition : 'Prohibition' '(' ruleBody  ')' ('if' conditions )? ';'
            ;

conditions : andcondition ( 'or' andcondition)*
           ;

andcondition : negcondition  ( 'and' negcondition)*
             ;

negcondition : 'not' simplecondition
              | simplecondition
              ;

simplecondition : locCondition | subidCondition | obidCondition | ambientCondition;


locCondition:  ('targetLocation' | 'subjectLocation')('!=' | '==') location_ref
            ;

subidCondition :  'subjectID' ('!=' | '==') (ID | CAP_ID)
           ;

obidCondition :  'targetID'  ('!=' | '==') (ID | CAP_ID)
           ;

ambientCondition
              : contextName ('!=' | '==') contextValue
              ;

ruleBody
      : role_ref ',' action_ref ',' objectType_ref
      ;

location_ref
          : building_ref '/' floor_ref '/' hallway_ref '/' room_ref
          ;

building_ref : CAP_ID;

floor_ref : CAP_ID;

hallway_ref : CAP_ID;

room_ref : CAP_ID;

role_ref : ID ;

action_ref: CAP_ID;

objectType_ref: CAP_ID ;

contextName: CAP_ID ;
```

```

contextValue:  INT | BOOL | ID;

strategyName: CAP_ID ;


ID: ('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
CAP_ID: 'A'..'Z' ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')* ;
INT : '0'..'9' + ;
BOOL : 'true' | 'false' ;
WS: (' '| '\t' | '\n' | '\r'| '\f')+ ;
```

Listing 40: Grammar of the DiaSecur language

BIBLIOGRAPHY

[1] Standard and extended x10 code protocol, 1993. URL http://software.x10.com/pub/manuals/xtdcode.pdf.

[2] E. Aarts, R. Harwig, and M. Schuurmans. *The invisible future: The seamless integration of technology into everyday life*. McGraw-Hill Companies, 2002.

[3] E. H. L. Aarts and S. Marzano. *The New Everyday View on Ambient Intelligence*. Uitgeverij 010 Publishers, 2003.

[4] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, 1999. Springer-Verlag.

[5] M. D. Abrams. Renewed understanding of access control policies. In *Proceeding of the 16th National Computer Security Conference*, pages 87–96, Baltimore, Maryland, USA, September 1993.

[6] D. Agrawal, J. Giles, K. won Lee, and J. Lobo. Policy ratification. In *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 223–232. IEEE Computer Society, 2005.

[7] T. Ahmed and A. R. Tripathi. Static verification of security requirements in role based cscw systems. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 196–203, New York, NY, USA, 2003. ACM.

[8] F. Aldrich. Smart homes: Past, present and future. In *Inside the Smart Home*, pages 17–39. Springer London, 2003.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transaction on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004.

[10] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[11] M. Backes, W. Bagga, G. Karjoth, and M. Schunter. Efficient comparison of enterprise privacy policies. In *Proceedings*

of the 2004 ACM symposium on Applied computing, pages 375–382. ACM Press, 2004.

[12] D. Balfanz, G. Durfee, D. Smetters, and R. Grinter. In search of usable security: five lessons from the field. *Security Privacy, IEEE*, 2(5):19 –24, sept.-oct. 2004.

[13] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 36–47, Paris, France, June 2007. Springer-Verlag.

[14] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2 edition, Apr. 2003.

[15] D. Benslimane, S. Dustdar, and A. Sheth. Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15, Sept. 2008.

[16] M. Bishop. What is computer security? In *Security & Privacy, IEEE*, volume 1, pages 67–69, Davis, CA, USA, Jan 2003.

[17] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[18] A. Bouhoula and F. Jacquemard. Automatic verification of sufficient completeness for conditional constrained term rewriting systems. Rapport de recherche RR-5863, INRIA, 2006.

[19] T. Bourdier, H. Cirstea, M. Jaume, and H. Kirchner. Formal specification and validation of security policies. In *Foundations & Practice of Security*, volume 6888 of *Lecture Notes in Computer Science*, pages 148–163, Paris, France, 2011. Springer, Heidelberg.

[20] T. Braun, M. Diaz, J. Enrquez-Gabeiras, and T. Staub. *End-to-End Quality of Service Over Heterogeneous Networks*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[21] D. Brewer and M. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, volume 2, pages 206 –214, may 1989.

[22] G. Brose, A. Vogel, and K. Duddy. *Java Programming with CORBA, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 2001.

[23] J. Bruneau, W. Jouve, and C. Consel. DiaSim: A Parameterized Simulator for Pervasive Computing Applications. In *6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 09)*, Toronto, Canada, 2009. IEEE.

[24] D. Cassou. *Développement logiciel orienté paradigme de conception: la programmation dirigée par la spécification.* PhD thesis, Université de Bordeaux, 2011.

[25] D. Cassou, B. Bertran, N. Loriant, and C. Consel. A generative programming approach to developing pervasive computing systems. *In GPCE 09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 137–146, october 2009.

[26] T. Chaari, F. Laforest, and A. Celentano. Adaptation in Context-Aware Pervasive Information Systems: The SECAS Project. *Int. Journal on Pervasive Computing and Communications(IJPCC)*, 3(4):400–425, Dec. 2007.

[27] H. Chen, F. Perich, T. Finin, and A. Joshi. Soupa: Standard ontology for ubiquitous ans pervasive applications. In *International Conference on mobile and ubiquitous system: Networking and Services*, 2004.

[28] S. Chetan and R. Campbell. Towards fault tolerant pervasive computing. In *Pervasive 2004 Workshop on Sustainable Pervasive Computing*, pages 38–44, 2004.

[29] R. C. Christopher, C. K. Hess, M. Roman, and R. H. Campbell. Gaia: A development infrastructure for active spaces. In *Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001*, 2001.

[30] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.

[31] C. Consel. DiaSuite:A Paradigm-Oriented Software Development Approach (invited paper). In *20th ACM SIGPLAN workshop on Partial evaluation and program manipulation : PEPM 11*, pages 77–78, Austin, TX, United States, Jan. 2011. ACM.

[32] D. J. Cook, M. Youngblood, and S. K. Das. *A Multi-agent Approach to Controlling a Smart Environment*, volume 4008, pages 165–182. Springer-Verlag, 2006.

[33] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, 2001. Springer-Verlag.

[34] A. De Castro Alves. *OSGi in Depth*. Manning Publications, 2011.

[35] A. S. de Oliveira. *Réécriture et Modularité pour les Politiques de Sécurité*. PhD thesis, Université Henri Poincaré, 2008.

[36] A. K. Dey, G. D. Abowd, and D. Salber. A context-based infrastructure for smart environments, 1999.

[37] W. K. Edwards and R. E. Grinter. At home with ubiquitous computing: Seven challenges. In *Proceedings of the 3rd international conference on Ubiquitous Computing*, UbiComp '01, pages 256–272, London, UK, 2001. Springer-Verlag.

[38] M. T. et al. Extensible access control markup language(xacml) version 2.0. Technical report, OASIS, 2005.

[39] D. Ferraiolo and R. Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[40] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 196–205, New York, NY, USA, 2005. ACM.

[41] M. Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, Oct. 2010.

[42] R. Frohardt, B.-Y. E. Chang, and S. Sankaranarayanan. Access nets: modeling access to physical spaces. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 184–198, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1:22–31, 2002.

[44] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *9th Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165, Tsukuba, Japan, 1998. Springer-Verlag.

[45] S. Hadim and N. Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3):1–16, Mar. 2006.

[46] R. Haux. Individualization, globalization and health about sustainable information technologies and the aim of medical informatics. *International Journal of Medical Informatics*, 75(12):795 – 808, 2006.

[47] R. Hayton, J. Bacon, and K. Moody. Access control in an open distributed environment. *IEEE Symposium on Security and Privacy*, 0:3–15, 1998.

[48] K. Henricksen, J. Indulska, and A. Rakotonirainy. Infrastructure for pervasive computing: Challenges. In *GI Jahrestagung (1)*, pages 214–222, 2001.

[49] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transaction on Database Systems*, 26(2):214–260, jun 2001.

[50] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 31–42, Washington, DC, USA, 1997. IEEE Computer Society.

[51] B. Johanson, O. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1:67–74, 2002.

[52] L. Kagal, T. W. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 63–74, June 2003.

[53] L. Kagal, V. Korolev, S. Avancha, A. Joshi, T. Finin, and Y. Yesha. Centaurus: an infrastructure for service management in ubiquitous computing environments. *Wireless Network*, 8(6):619–635, november 2002.

[54] L. Kagal, V. Korolev, H. Chen, A. Joshi, and T. W. Finin. Centaurus: A framework for intelligent services in a mobile environment. In *Proceedings of the International Workshop on Smart Appliances and Wearable Computing*, pages 195–201, 2001.

[55] L. Kagal, F. Perich, A. Joshi, and T. Finin. A security architecture based on trust management for pervasive computing systems. In *Grace Hopper Celebration of Women in Computing*, October 2002.

[56] L. Kagal, J. Undercoffer, F. Perich, A. Joshi, T. Finin, and Y. Yesha. Vigil: Providing trust for enhanced security in pervasive systems. Technical report, University of Maryland Baltimore County, 2001.

[57] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith. Scalable security policy mechanisms, 2001.

[58] A. Kissinger and J. C. Hale. Lopol: A deductive database approach to policy analysis and rewriting. In *Second Annual SELinux Symposium*, 2006.

[59] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. Exam - a comprehensive environment for the analysis of access control policies. Technical report, Dept of Computer Science, Purdue University, 2007.

[60] D. Maier and D. Warren. *Computing with logic: logic programming with Prolog*. Benjamin/Cummings Pub. Co., 1988.

[61] J. McLean. A comment on the "Basic Security Theorem" of Bell and LaPadula. *Information Processing Letters*, 20(2):67–70, 1985.

[62] S. Mokhtar, D. Fournier, N. Georgantas, and V. Issarny. Context-aware service composition in pervasive computing environments. In N. Guelfi and A. Savidis, editors, *Rapid Integration of Software Engineering Techniques*, volume 3943 of *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin / Heidelberg, 2006.

[63] D. J. Moore, I. A. Essa, and M. H. H. Iii. Exploiting human actions and object context for recognition tasks. *Computer Vision, IEEE International Conference on*, 1:80, 1999.

[64] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert. Netra:: seeing through access control. In *Proceedings of the fourth ACM workshop on Formal methods in security*, FMSE '06, pages 55–66, New York, NY, USA, 2006. ACM.

[65] J. Nehmer, M. Becker, A. Karshmer, and R. Lamm. Living assistance systems: an ambient intelligence approach. In *Proceeding of the 28th international Conference on Software Engineering*, pages 43–50. ACM Press, 2006.

[66] R. d. F. B. Neto and M. d. G. C. Pimentel. Toward a domain-independent semantic model for context-aware computing. In *Proceedings of the Third Latin American Web Congress*, LA-WEB '05, pages 61–, Washington, DC, USA, 2005. IEEE Computer Society.

[67] U. S. D. of Defense. Trusted computer system evaluation criteria, December 1985.

[68] R. Orfali and D. Harkey. *Client/server programming with Java and CORBA (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[69] X. Ou and S. Govindavajhala. Mulval: A logic-based network security analyzer. In *14th USENIX Security Symposium*, pages 113–128, 2005.

[70] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[71] E. Pitt and K. McNiff. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[72] 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 *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, PERCOM '05, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[73] I. Ray, M. Kumar, and L. Yu. Lrbac: A location-aware role-based access control model. In *Proceedings of the 2nd International Conference on Information Systems Security*, pages 147–161, 2006.

[74] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1 edition, 1998.

[75] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. Spl: An access control language for security policies and complex constraints. In *NDSS*, 2001.

[76] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct. 2002.

[77] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol, 2002.

[78] F. Sadri. Ambient intelligence: A survey. *ACM Computing Survey*, 43(4):36:1–36:66, Oct. 2011.

[79] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM.

[80] G. Sampemane, P. Naldurg, and R. H. Campbell. Access control for active spaces. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 343–, Washington, DC, USA, 2002. IEEE Computer Society.

[81] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29:38–47, February 1996.

[82] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

[83] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001.

[84] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs by term rewriting. In *Proceedings of the 16th international conference on Logic-based program synthesis and transformation*, LOPSTR'06, pages 177–193, Berlin, Heidelberg, 2007. Springer-Verlag.

[85] E. Serral, P. Valderas, and V. Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2):254–280, Apr. 2010.

[86] N. Shadbolt. Ambient intelligence. *IEEE Intelligent Systems*, 18(4):2–3, July 2003.

[87] A. Singh, C. R. Ramakrishnan, I. V. Ramakrishnan, S. D. Stoller, and D. S. Warren. Security policy analysis using deductive spreadsheets. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, FMSE '07, pages 42–50, New York, NY, USA, 2007. ACM.

[88] M. Sloman and E. Lupu. Engineering policy-based ubiquitous systems. *Computing Journal*, 53(7):1113–1127, Sept. 2010.

[89] J. a. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the IFIP 17th World Computer*

Congress, WICSA 3, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[90] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[91] D. Thomas. Mda: Revenge of the modelers or uml utopia. *IEEE Software*, 21:15–17, 2004.

[92] K. Vaniea, Q. Ni, L. Cranor, and E. Bertino. Access control policy analysis and visualization tools for security professionals. In *USM 08: Workshop on Usable IT Security Management 2008*, Carnegie Mellon University, Pittsburgh, July 2008.

[93] D. C. Verma. *Policy-Based Networking: Architecture and Algorithms*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[94] T. D. Wang, B. Parsia, and J. Hendler. A survey of the web ontology landscape. In *Proceedings of the International Semantic Web Conference, ISWC*, 2006.

[95] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.

[96] M. Weiser and J. S. Brown. The coming age of calm technology, 1996.

[97] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM Systems Journal*, 38(4):693–696, 1999.

[98] D. M. Weiss. Commonality analysis: A systematic process for defining families. *Lecture Notes in Computer Science*, 1429:214–225, 1998.

[99] D. M. Weiss and L. C.T.R. *Software Product Line Engineering*. Addison-Wesley, 1999.

[100] M. Youngblood and D. J. Cook. The mavhome architecture. Technical report, Department of Computer Science and Engineering, University of Texas at Arlington, 2004.

[101] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in xacml. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 56–65, New York, NY, USA, 2004. ACM.