Towards securing pervasive computing systems by design: a language approach

Henner Jakob

UNIVERSITÉ DE BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux
N° d'ordre : 4269

Vers la sécurisation des systèmes d'informatique ubiquitaire par le design: une approche langage

THÈSE

soutenue le 27/06/2011

pour l'obtention du

Doctorat de l'Université de Bordeaux

(spécialité informatique)

par

Henner Jakob

Jury

Président :	Serge Chaumette,	Professeur à l'Université Bordeaux 1
Rapporteurs :	Matthias Hollick,	Professeur à Technische Universität Darmstadt
	Philippe Lalanda,	Professeur à Grenoble University (UJF)
Examinateurs :	Charles Consel,	Professeur à l'ENSEIRB, Bordeaux
	Nicolas Loriant,	Research Associate à Imperial College, London

Laboratoire Bordelais de Recherche en Informatique — UMR 5800

VERS LA SÉCURISATION DES SYSTÈMES D'INFORMATIQUE UBIQUITAIRE PAR LE DESIGN: UNE APPROCHE LANGAGE

TOWARDS SECURING PERVASIVE COMPUTING SYSTEMS BY DESIGN: A LANGUAGE APPROACH

HENNER JAKOB

UNIVERSITÉ DE BORDEAUX

Dedicated to:
my parents, Barbara and Hans Jakob,
my longest and best friend, Marco Ebel,
and my (s)well of motivation, the Atlantic Ocean.

ACKNOWLEDGMENTS

First of all, I thank my advisor Charles Consel for offering the opportunity to write a thesis, his valuable comments and his guidance during these three years.

I deeply thank my thesis committee, Philippe Lalanda and Matthias Hollick who read my thesis and provided me with feedback, Serge Chaumette who lead the jury of my thesis defense as president, and Nicolas Loriant with whom I worked during my thesis.

I am also very grateful to the members of the Phoenix research group, for their welcoming attitude and support. I thank Damien for many many responses, long discussions, and an excellent taste concerning music; Nicolas for working together on cross-cutting concerns, weekly croissants, and the famous *saviez-vous*, which boosted my French; Julien B. for valuable feedback and special techniques in table-soccer; Pengfei for discussions about security and teaching me some Chinese; Hongyu for encouraging words during the *final* phase and always offering feedback on my writings; Stéphanie for explaining me difficult nuances and expressions of the French language; Quentin for technical discussions and always being friendly; the engineers Benjamin and Ghislain, who helped me understand the depths of DiaSuite, thus making it possible to integrate my work; Emilie for providing feedback and comments from another angle, and promoting geeky b-movies; Julien M. for his friendly attitude and never saying no when I needed some help; Zoé for useful feedback, motivating words, and the many jokes that increased my French vocabulary and understandings; Julien A. (non Phoenix) for discussions in German, many non-work activities and being a friend.

Beside the support at work, there are some more people I would like to thank. It is more or less in a chronological order. It is a long list, but through these three years, time and distance made it difficult to maintain all friendships as I would like to have done it.

I thank Verena, Diana and Petra for motivating me to work in a foreign country. Without you, I would not have left Germany. I thank my old colleagues at KOM who assured me in my ability of writing a thesis, especially Nick, Tronje, Stefan and Matthias. I thank Union Saint Bruno and the whole water polo section. Sport was my balance and it was a pleasure doing sports on national level with you guys. Special thanks to Xavier, Fabien, Zézé and Guillaume, who integrated me socially in Bordeaux.

Many thanks also to the triathlon section of the Girondins, for your motivation and exceptional team spirit.

I thank all my friends from Germany. Special thanks go to my girlfriend Hanna for extreme telephone support and many visits. Special thanks also to Marco, my best friend. I thank all that came by to visit me, Malte, Felix, Falco, Katrin. The surf crew, Balazs, Maria, Christian, Jason, Verena (again), Lion and Sabrina. Not forgotten are those, who always welcomed me back in Germany, like I never parted, Martin, Olga, Julius, Anna, Anni, Sascha, Jürgen.

Last but not least, I thank my family. My parents Barbara and Hans who supported my decisions through all my studies and my brother Bastian for encouraging words, his excellent cooking, and long board advices.

RÉSUMÉ

Dans de multiples domaines, un nombre grandissant d'applications interagissant avec des entités communicantes apparaissent dans l'environnement pour faciliter les activités quotidiennes (domotique et télémédecine). Leur impact sur la vie de tous les jours des utilisateurs rend ces applications critiques: leur défaillance peut mettre en danger des personnes et leurs biens. Bien que l'impact de ces défaillances puisse être majeur, la sécurité est souvent considérée comme un problème secondaire dans le processus de développement et est traitée par des approches ad hoc.

Cette thèse propose d'intégrer des aspects de sécurité dans le cycle de développement des systèmes d'informatique ubiquitaire. La sécurité est spécifiée à la conception grâce à des déclarations dédiées et de haut niveau. Ces déclarations sont utilisées pour générer un support de programmation afin de faciliter l'implémentation des mécanismes de sécurité, tout en séparant ces aspects de sécurité de la logique applicative. Notre approche se concentre sur le contrôle d'accès aux entités et la protection de la vie privée. Notre travail a été implémenté et fait levier sur une suite outillée existante couvrant le cycle de développement logiciel.

ABSTRACT

A growing number of environments is being populated with a range of networked devices. Applications leverage these devices to support everyday activities in a variety of areas (*e.g.*, home automation and patient monitoring). As these devices and applications get woven into our everyday activities, they become critical: their failure can put people and assets at risk. Failures can be caused by malicious attacks and misbehaving applications. Although the impact of such situations can be major, security concerns are often considered a secondary issue in the development process, and treated with ad hoc approaches.

This thesis proposes to address security concerns throughout the development lifecycle of a pervasive computing system. Security is addressed at design time thanks to dedicated, high-level declarations. These declarations are processed to implement security mechanisms, and to generate programming support to ease the development of the security logic, while keeping it separate from the application logic. Our approach is studied in the context of access control and privacy concerns. Our work has been imple-

mented and leverages an existing software-design language and a suite of tools that covers the software development lifecycle.

RÉSUMÉ ÉTENDU

Les progrès des technologies de télécommunication et la prolifération des entités communicantes permettent une intégration transparente des systèmes informatiques dans notre vie quotidienne. Aujourd'hui, comme l'avait prévu Weiser [88], les systèmes d'informatique ubiquitaire sont déployés dans de nombreux domaines tels que la domotique et l'assistance à la personne.

L'émergence de l'informatique ubiquitaire combinant des éléments de différents domaines informatiques (systèmes distribués, informatique mobile, *etc.*) fait apparaitre de nombreux défis à aborder. Prenons l'exemple d'une maison intelligente afin d'illustrer ces défis. Typiquement, une maison intelligente se compose de plusieurs applications d'informatique ubiquitaire. Ces applications reçoivent des données provenant de différents capteurs, prennent des décisions à partir de ces données collectées et agissent sur des actuateurs en fonction des décisions prises. Par exemple, des capteurs de mouvement et de température sont utilisés pour automatiser l'éclairage et réguler le chauffage.

Le développement d'applications d'informatique ubiquitaire demande de répondre à de nombreuses difficultés techniques telles que l'hétérogénéité des entités communicantes et des protocoles de communication utilisés, la découverte de services et d'entités, ou encore l'orchestration de ces entités afin qu'elles aident les utilisateurs dans leurs activités quotidiennes.

L'informatique ubiquitaire implique que l'utilisateur soit entouré d'entités communicantes, et que celles-ci deviennent partie intégrante de sa vie quotidienne. Ainsi, l'informatique ubiquitaire inclut des applications critiques telles qu'un système anti-incendie. Une défaillance de ce type de système peut donc impacter de manière très critique les utilisateurs et leurs biens.

A ce stade, les propriétés non fonctionnelles[1] (PNF) d'un système doivent être considérées pour assurer sa fiabilité. Taylor et al. [85] définissent une PNF d'un logiciel en tant que "contrainte sur la manière dont le système implémente et délivre ses fonctionnalités".

La sécurité est une PNF primordiale et propose de nombreux défis. La sécurité et les PNF[2] en général offrent de nombreuses possibilités de recherche et demandent bien plus qu'une seule thèse. En conséquence, nous limitons le champ de cette thèse à deux aspects de sécurité particulièrement intéressants pour les

1 Dans la littérature, ces propriétés sont également appelées *qualités* d'un système [3]

2 D'autres exemples de PNF sont la fiabilité, la tolérance aux fautes, la performance, *etc.*

systèmes d'informatique ubiquitaire: le *contrôle d'accès* et *le respect de la vie privée* [10].

Comme il a été mentionné précédemment, les maisons intelligentes sont des systèmes critiques et ont donc besoin d'une protection convenable.

Le contrôle d'accès garantit que seules les personnes et les applications autorisées peuvent interagir avec les services et ressources de la maison (par exemple, empêcher des personnes malveillantes de contrôler la maison à distance ou d' assurer un partage fiable des ressources entre les applications). Pour aider l'homme dans ses activités de tous les jours, la maison intelligente collecte de nombreuses données. Le respect de la vie privée contraint à se soucier de la fuite de données sensibles vers le monde extérieur, (par exemple, un cambrioleur pourrait pirater et utiliser les caméras vidéo transmettant des images à travers le Wifi sans protection). De toute évidence, ce type de système doit garantir que de telles situations ne peuvent pas se produire.

En général la sécurité impacte tous les aspects d'un système domotique, à la fois les applications déployées et le cœur du système lui même. L'intégration des mécanismes de sécurité de manière ad hoc mène à du code imbriqué à plusieurs endroits, ce qui rend la maintenance et l'évolution du système pratiquement impossible. Ceci est d'ailleurs vrai pour toutes les PNF. Afin de résoudre ce problème, Bass et al. [3] propose de les considérer à toutes les étapes du cycle de développement d'un système logiciel: depuis la conception jusqu'au déploiement en passant par l'implémentation.

Cependant, les approches existantes se concentrent sur le support de développement d'environnements intelligents, ou bien adoptent une approche plus formelle en examinant et adaptant des modèles de sécurité pour répondre aux différents défis de l'informatique ubiquitaire. Aucun support n'est fourni au développeur pour intégrer la sécurité systématiquement au cours du processus de développement. Puisque la sécurité et les autres PNF sont toujours en concurrence avec (et perdent souvent contre) les exigences fonctionnelles et de temps de mise sur le marché [85], aider et faciliter le travail des développeurs dans ces domaines est crucial pour construire des systèmes sûrs et fiables.

APPROCHE

La première partie de cette thèse présente un modèle de contrôle d'accès qui cible un problème particulier des systèmes d'informatique ubiquitaire: l'utilisation conflictuelle de ressources. Notre approche couvre le cycle de développement logiciel et permet notamment d'enrichir la description d'un système in-

formatique ubiquitaire avec des déclarations sur le partage des ressources. Ces déclarations sont utilisées pour automatiser la détection des conflits, gérer les états d'un système d'informatique ubiquitaire et orchestrer, à l'exécution, les accès aux ressources.

La deuxième partie présente notre approche sur l'intégration des PNF dans un système d'informatique ubiquitaire. Nous illustrons cette approche au travers de deux aspects de la sécurité informatique qui visent à améliorer la protection de la vie privée. Pour ce faire, nous utilisons la programmation orientée aspect (POA [44]), une technique éprouvée de modularisation des PNF. Afin de s'abstraire des détails de bas-niveau d'un système d'informatique ubiquitaire, nous présentons un langage orienté aspect s'appuyant sur leur description conceptuelle, permettant ainsi un tissage fin des aspects.

Les deux parties de cette thèse s'appuient sur DiaSuite[3], une suite d'outils fournissant un environnement de développement dédié aux systèmes informatiques ubiquitaire. DiaSuite repose sur un langage de conception dédié: DiaSpec. A partir de la spécification DiaSpec d'une application, un environnement de développement est généré comprenant une aide à l'implémentation, une série de tests, et un outil de déploiement. Nous proposons d'enrichir ces spécifications avec des configurations de déploiement cohérentes et de tisser des aspects lors de l'implémentation.

En effet, notre approche repose sur la séparation de la logique applicative et des aspects non fonctionnels. Nous avons mis en place et validé cette approche sur diverses applications de domotique.

CONTRIBUTIONS

Ce travail met en œuvre et détaille une approche d'intégration d'aspects de sécurité dans les systèmes d'informatique ubiquitaire. Les contributions apportées par cette thèse sont les suivantes.

- *Extension du cycle de développement* – Nous avons identifié les exigences sur les différentes étapes du développement logiciel afin de détecter, résoudre et prévenir les conflits. Nous avons intégré des activités de gestion des conflits dans le cycle de développement logiciel.

- *Déclarations de gestion des conflits* – Nous avons étendu un langage de conception spécifique au domaine de la domotique pour la résolution des conflits. Une approche déclarative est introduite pour définir les états d'un système

3 DiaSuite est une suite d'outils open source librement disponible à l'adresse suivant : http://diasuite.inria.fr/.

d'informatique ubiquitaire et leur criticité. Ces déclarations constituent la base pour définir la logique de gestion des conflits.

- *Support de programmation* – Les déclarations dédiées à la gestion des conflits au niveau de la conception du logiciel sont utilisées pour enrichir le canevas de programmation avec la génération de code dédié à la gestion des conflits. Ce code (1) guide la mise en œuvre de la logique de gestion des conflits intra et inter-applications, et (2) gère les accès aux ressources à l'exécution pour empêcher les conflits.

- *Un langage orienté aspects* – Le langage développé, DiaAspect, permet l'expression des fonctionnalités transversales au niveau de la conception. Pour ce faire, le langage fournit un modèle des points de jonction pour manipuler les motifs de conception de DiaSpec. Les aspects déclarés sont automatiquement tissés dans le canevas de programmation généré.

- *Extensions vers un système domotique sécurisé* – Nous avons validé DiaAspect en l'utilisant pour implémenter, d'une part, une communication cryptée entre des composants et, d'autre part, un contrôle d'accès dans un système domotique. Ces deux mécanismes sont nécessaires pour la protection de la vie privée: les données échangées sont protégées par cryptage, tandis que les accès aux données sont supervisés.

PLAN DE LA THÈSE

Cette thèse est divisée en quatre parties. La première partie présente le contexte et le problème, la deuxième partie décrit notre approche qui vise le contrôle d'accès, la troisième partie présente notre approche qui traite de la protection de la vie privée, et la quatrième partie conclue par les résultats et travaux futurs.

Contexte

La première partie présente le contexte et les travaux existants. Le chapitre 2 définit la terminologie de base et introduit le domaine de la domotique, avec ses motivations et ses enjeux, particulièrement au regard des problèmes de sécurité. Ensuite, nous détaillons les approches existantes et leurs limites au niveau des aspects fonctionnels et non fonctionnels. Le chapitre 3 donne un aperçu de notre approche et définit des hypothèses qui cadrent la portée de cette thèse. Le chapitre 4 présente notre langage de

conception sur lequel notre approche est basée, et introduit des exemples illustratifs qui sont utilisés tout au long de ce travail.

Contrôle d'accès

La deuxième partie décrit notre approche d'intégration d'un modèle de contrôle d'accès dans les maisons intelligentes. Le chapitre 5 présente une approche dédiée à la gestion des conflits de ressources. Pour ce faire, DiaSpec et le processus de développement sont étendus afin d'assurer une intégration au plus tôt et de configurer le mécanisme au niveau de la conception. Le chapitre 6 évalue notre approche et présente les travaux existants dans ce domaine.

Vie privée

La troisième partie présente une approche de Programmation Orientée Aspect (POA) qui est suffisamment expressive pour intégrer des propriétés de sécurité dans le système. Le chapitre 7 commence par une introduction du POA. Ensuite, nous donnons les détails de notre approche et nous démontrons son apport en mettant en œuvre des solutions concrètes sur deux problèmes de sécurité: la distribution de certificats entre des composants et la mise en place de listes de contrôle d'accès dans un système domotique. Le chapitre 8 évalue cette approche et présente les travaux existants dans ce domaine.

Résultats

La quatrième partie résume les résultats de la thèse. Le chapitre 9 conclue sur les travaux réalisés dans le cadre de la thèse et le chapitre 10 présente des problèmes restants et des pistes de travaux futurs.

LISTE DES PUBLICATIONS

Les travaux discutés dans cette thèse ont fait l'objet de publications internationales dans la domaine de l'informatique ubiquitaire.

- Architecturing Conflict Handling of Pervasive Computing Resources, in *Proceedings of the 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'11)*, Reykjavik, Iceland, June 6-9, 2011, 92-105. Henner Jakob, Charles Consel et Nicolas Loriant
- An Aspect-Oriented Approach to Securing Distributed Systems, in *Proceedings of the 6th International Conference on Pervasive Services (ICPS'09)*,London, United Kingdom, July 13-17, 2009, 21-30, Henner Jakob, Nicolas Loriant et Charles Consel

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION

Security aspects of software systems should be considered from a project's start. During system conception, the security requirements should be identified and corresponding security measures designed. Patching security problems after a system is built can be prohibitively expensive, if not technically infeasible.

— Taylor et al. [85]

The advances in telecommunication technologies and the proliferation of embedded networked devices are allowing the seamless integration of computing systems in our everyday lives. Nowadays, pervasive computing systems, as envisioned by Weiser [88], are being deployed in an increasing number of areas, including building automation and assisted living.

With this new paradigm of pervasive computing, which combines elements from different computer science domains (*e.g.*, distributed systems, mobile computing), comes a number of challenges that have to be addressed [74]. To illustrate these, let us take the example of a smart home. Typically, a smart home consists of multiple applications that gather data from sensor devices, compute decisions from the data collected, and carry out these decisions by orchestrating actuating devices. For example, motion and temperature sensors are used to automate lighting and regulate heating.

Functional challenges for application development include the heterogeneity of the deployed devices, the different communication protocols they use, service and device discovery, and the orchestration of the devices such that they support the users in their daily activities.

Pervasive computing implies that technical devices are everywhere, surrounding the user, becoming a part of the user's everyday life [88]. As a result, pervasive computing includes critical areas such as people evacuation in case of fire. Thus a failure of the home automation system can have critical effects on the physical world, putting people and assets at risk.

At this point, the non-functional properties[1] (NFP) of a system have to be addressed to make it dependable. Taylor et al. [85] define an NFP of a software system as "a constraint on the manner in which the system implements and delivers its functionality".

Security is an NFP of great, and growing importance that opens up a range of challenges. Security, and NFPs[2]in general, provide many opportunities for research and require more than a single thesis. Therefore, we narrow the scope of this thesis and focus on two aspects from the security domain that are of special interest for home automation systems: *access control* and *privacy* [10].

As previously mentioned, smart homes are critical systems, and therefore require proper protection. Access control ensures that only authorized people and applications can access and use services and resources in the smart home (*e.g.*, preventing malicious persons from remotely controlling a smart home, and enforcing safe resource sharing between applications). Privacy addresses the problem of leaking sensitive data to the outside world (*e.g.*, a burglar could take advantage of wireless cameras transmitting video streams in clear). Obviously, home automation systems require guarantees that situations like this cannot occur.

Security concerns typically impact every aspect of a home automation system, that is, both the deployed applications and the core system itself. Integrating security in an ad hoc manner leads to entangled code at several places, making maintenance and upgrading of the system virtually impossible. Since this is generally true for all NFPs, Bass et al. [3] propose to consider them throughout the design, implementation, and deployment phases of software systems. But, existing approaches either concentrate on developing smart environments by providing programming support for the functionality or take a more formal approach that examines and adapts security models to address the various challenges introduced by the pervasive computing domain. No support is given to the developer to systematically integrate security concerns during the development process. Since security and other NFPs always compete with (and often lose out to) *time-to-market* and functional requirements [85], supporting and facilitating the work for the developer in this area are critical issues in building secure and reliable systems.

1.1 APPROACH

The first part of this thesis presents an access control model that targets a particular problem in pervasive computing systems: re-

1 In the literature they are also referred to as the *qualities* of a system [3]
2 Other examples for NFPs are reliability, fault-tolerance, efficiency, *etc.*

source conflicts. This approach covers the software development lifecycle and consists of enriching the description of a pervasive computing system with declarations for resource sharing. These declarations are used to automate conflict detection, manage the states of a pervasive computing system, and orchestrate resource accesses accordingly at runtime.

The second part presents our approach to integrating non-functional properties into a pervasive computing system. We illustrate this approach with two examples of security concerns that improve privacy. To do so, we use aspect-oriented programming (AOP [44]), a well-proven technique to properly modularize non-functional concerns. To abstract over low level details of the pervasive computing system, we provide an aspect-oriented language that makes use of the design description of such systems, enabling an accurate coordination of aspects.

Both parts of this thesis leverage DiaSuite[3], a toolkit which provides a development environment dedicated to pervasive computing systems. The core of DiaSuite is a domain-specific design language for pervasive computing systems named DiaSpec. A dedicated programming framework is generated from a DiaSpec specification, which supports the implementation of the functionality, the testing, and the deployment of applications. The existing specifications are reused by our approach to generate coherent configurations, and for aspect weaving in the implementation code.

In effect, our approach features the separation of non-functional concerns from the application logic. We have implemented and validated our approach on various building automation applications.

1.2 THESIS CONTRIBUTIONS

This work implements and examines an approach to integrating security concerns into pervasive computing systems. The specific contributions of this thesis are described below.

- *Extended development cycle* – We have identified the requirements at different development stages to detect, resolve, and prevent conflicts. We have seamlessly integrated conflict-management activities into a software development lifecycle.
- *Conflict-handling declarations* – We have extended a domain-specific design language to declare conflict resolution at a design level. A declarative approach is introduced to define

3 DiaSuite is freely available http://diasuite.inria.fr/ and open source.

the states of a pervasive computing system and their critical nature. Such declarations form the basis used to define the conflict-handling logic of a pervasive computing system.

- *Programming support* – Conflict-handling declarations are used to augment the generated programming framework with code dedicated to conflict handling. This code (1) guides the implementation of the conflict-handling logic within and across applications, and (2) generates code that manages resource accesses to prevent runtime conflicts.
- *An aspect-oriented language.* – The language developed, DiaAspect, allows the expression of crosscutting concerns at design level. To do so, it provides a join point model to manipulate the design patterns used in DiaSpec. The declared aspects are then automatically woven into a generated programming framework.
- *Extensions towards a secure home automation system* – We validated DiaAspect by securing the communication between components and enforcing access control in a home automation system. Both mechanisms are required to improve the privacy: exchanged data is protected by encryption, while accesses to data are supervised.

1.3 ROADMAP

This thesis is split into four parts: The first part introduces the context and the problem, the second part describes our approach that targets access control, the third part presents our approach that addresses privacy, and the fourth part concludes with the results and outlines future work.

Context

The first part introduces the context and examines related work. Chapter 2 defines basic terminology and introduces the domain of home automation, with its motivations and challenges, specifically security concerns. Afterwards we examine existing approaches and their limitations in relation to functional and non-functional concerns. Chapter 3 gives an overview of our approach and makes assumptions to limit the scope of this thesis. Chapter 4 introduces a design language dedicated to pervasive computing, DiaSpec, which is the foundation of this work, and presents the illustrating examples that appear throughout this work.

Access Control

The second part describes our approach to integrating an access control model into smart homes. Chapter 5 presents a dedicated approach to using access control for resource conflict handling. Therefore, DiaSpec and the coherent development process are extended to ensure early integration and context-aware configuration of the mechanism at design level. An evaluation of the approach and specific related work follow in Chapter 6.

Privacy

The third part presents an aspect-oriented approach to integrate system-wide security properties. Chapter 7 starts with an introduction to AOP, details our approach and shows its usefulness by implementing concrete solutions on two widespread security problems: the distribution of certificates over an encrypted network and the enforcement of access control lists. Chapter 8 evaluates the approach and presents related work in this area.

Results

The fourth part summarizes the results. Chapter 9 draws overall conclusions while Chapter 10 points out remaining problems and outlines present avenues for future work.

Part I

CONTEXT

2 BACKGROUND

"smart homes" – domestic environments in which we are surrounded by interconnected technologies that are, more or less, responsive to our presence and actions –

— Edwards and Grinter [24]

This chapter starts with an introduction of the home automation domain, which serves as the working example throughout this thesis. We look into the reasons behind the design of *smart homes*, which have been under development since the seventies, and examine why they are not yet publicly available.

From a developer's perspective, the main difficulties lie in developing the functionalities of applications and securing home automation systems. Afterwards, we present approaches that address these problems and show their limitations.

2.1 HOME AUTOMATION SYSTEMS

The home automation domain applies pervasive computing technologies in living environments. This thesis uses scenarios and examples from this domain to illustrate various challenges and requirements, as well as the contributions of this work.

2.1.1 *Terminology*

We begin by introducing the terminology that is often used in the context of home automation, that is, *pervasive computing*, *ubiquitous computing*, and *ambient intelligence*.

Ubiquitous Computing and Pervasive Computing

Weiser [88] envisioned in the early nineties that "technology would disappear by weaving itself into the fabric of everyday life and finally become indistinguishable from it". This *ubiquitous computing*[1] is based on two requirements: technology and interaction. On the technical side, the miniaturization of processors

1 Most definitions put pervasive computing on the same level with ubiquitous computing. In this work both terms are used interchangeably.

and the evolution of telecommunication technologies (*e.g.*, 3G[2], WLAN[3]) made it possible to deploy networked computing devices everywhere: nowadays people carry around smartphones that have more computing power than a personal computer from the late nineties. On the interaction side Weiser et al. [90] believed that the interaction with technology would move into the periphery, with technology reading the signs and gestures of users and doing many things automatically. In the case of direct interaction, the necessary information must be moved into the center of the user's attention, and then back into the periphery, so that users are not overwhelmed by the amount of information offered by a ubiquitous computing system [89].

Ambient Intelligence

Ambient intelligence, a term mainly used in Europe, positions itself more within the domain of human-computer interaction. Shadbolt [76] sees ambient intelligence as a convergence of ubiquitous computing, intelligent systems, and context awareness. While the first provides networking capabilities, the second concentrates on new interfaces, such as speech recognition, gesture classification and situation assessment, and the third focuses on locating objects and object-environment interactions.

2.1.2 *Smart Homes*

Aldrich [1] defines 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".

Aldrich proposes five hierarchical classes for smart homes[4]:

1. *Homes which contain intelligent devices* – homes which contain devices that function in an intelligent manner.
2. *Homes which contain intelligent, communicating devices* – homes which contain devices that communicate with each other to increase functionality.
3. *Connected homes* – homes which allow for interactive and remote control of the system.

2 3rd Generation of standards concerning information and communication technologies provided by the International Telecommunication Union (ITU).

3 Wireless Local Area Network.

4 Since this is a hierarchical listing, a certain class includes all the functionalities of the lower classes, so, for example, a smart home of class three, also provides the functionality of class two and class one.

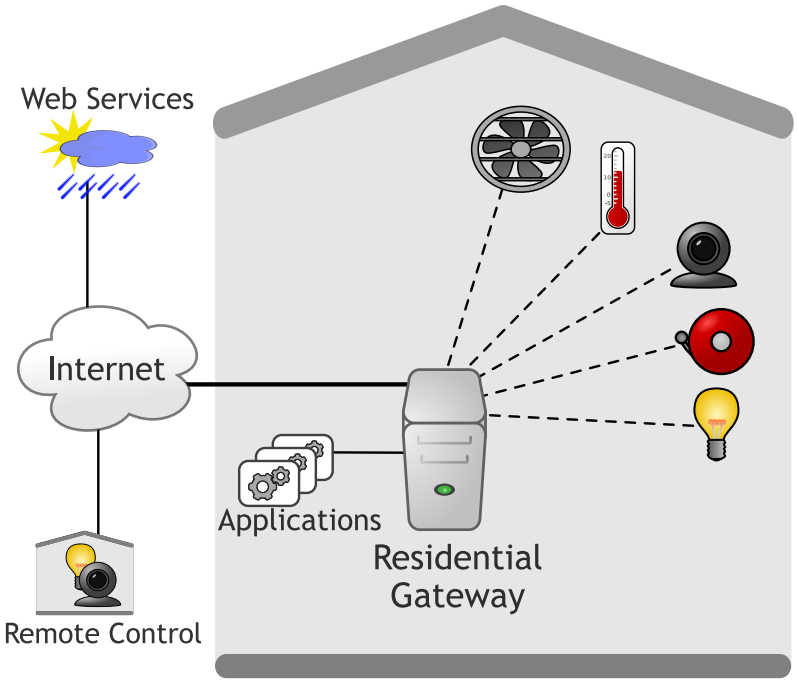

Figure 1: A smart home and its resources

4. *Learning homes* – homes which learn activity patterns, anticipate users' needs, and act on their behalf.
5. *Attentive homes* – homes which register the location and activity of occupants and use this information to control technology.

In this work we refer to smart homes of class five, thus all the above-mentioned functionality is supported. Specifically, this means that devices deployed in a living environment are capable of communicating with each other. Sensors gather data of various kinds that are processed and interpreted by applications. According to this decision process, the applications use acting devices to manipulate the environment to achieve their specified goals (*e.g.*, a `Light` application collects data from motion detectors and light sensors, and uses blinds and lamps to manage the lighting in the smart home).

Figure 1 shows a smart home and the various devices deployed within it. The *residential gateway* is a central node in this environment. On one side, it is connected to all the deployed devices within the smart home and provides software services, such as an address book, for example. On the other side, it connects the smart home with the internet, providing the possibility of integrating web services into deployed applications (*e.g.*, weather forecasts), and remotely controlling the smart home. This central role makes it an ideal candidate to host the runtime environment for applications. Chapter 4 introduces the approach of the Phoenix research group to providing a runtime environment

on the residential gateway. Our goal is to provide a platform, where the user can plug in new hardware and then download and install applications to make use of the deployed devices[5].

2.1.3 *Motivation*

The purposes of smart homes are varied. Some general goals are comfort, convenience, security and entertainment [1]. With the rising prices for electricity, gas, and oil, energy savings also become an interesting factor. Another promising domain is aided or assisted living. Here, the smart home aids elderly or challenged people through their daily activities, making them less dependable on others [81]. This is also of major relevance in the healthcare domain, where many new sensors that can be carried around by patients are being developed. Applications in the smart home could use the data from these sensors to call for help in case of an emergency or remind the patient to take his or her medication. Moore uses the term *telehealth* to describe "the full array of technologies, networks and healthcare services provided through telecommunication " [51].

2.2 CHALLENGES IN HOME AUTOMATION SYSTEMS

The idea of a smart home is nothing new. In the 1970s people thought that by now we would live in apartments with several automated tasks and would be able to do nearly everything with a remote control. Even though the technology of today has all the potential needed for a smart home, several challenges have yet to be solved.

Edwards and Grinter [24] divide the various challenges that have to be overcome to make smart homes available to the general public into seven categories.

1. *The "Accidentally" Smart Home*
   A smart home contains many different devices that communicate with each other. It is important that homeowners understand their smart homes. That means, pervasive computing has to provide insights into what these devices can do, what they have done, and how they can be controlled. For example, a stereo system that connects to multiple speakers throughout the smart home using wireless technologies must not accidentally connect to the neighbours' speakers.

5 We envision application stores for smart homes, like those currently available for smartphones.

2. *Impromptu Interoperability*
   Networked computing devices are the essence of the smart home. Instead of tearing down their houses to build a new smart home, homeowners will deploy new technologies over time in their existing houses. This raises the question of how the devices will interconnect and achieve a certain goal together without planning this in advance.
3. *No System Administrator*
   All the new technologies entering the home have to be installed and configured in some way. Tasks that were performed by specialists a few years back for large computer systems will now need to be done by the end users, *e.g.* installing and configuring new devices.
4. *Designing for Domestic Use*
   Adding technology to the living environment will impact the existing routines of occupants. Since it is impossible to predict how people will use new technology, the design of smart homes and applications is complicated.
5. *Social Implications of Aware Home Technologies*
   New technologies do not only impact daily routines as mentioned in the previous point, they also change these routines. The social consequences that arise from these changes are unforeseeable.
6. *Reliability*
   When people buy equipment for their home, they expect it to work, regardless of its complexity, *e.g.*, microwaves, televisions. In general, products that are used in homes are thoroughly tested to prove their reliability, since there are a number of regulations to be respected. Pervasive computing makes prior testing difficult, because the system is constructed in an ad hoc manner by plugging several devices together.
7. *Inference in the Presence of Ambiguity*
   Smart homes should learn from user habits and adapt automatically to support daily goals. Somehow the system has to infer the necessary information. A crucial requirement is that the occupants understand how the system infers information. Only with this knowledge can they then understand why the system has failed in given situation[6].

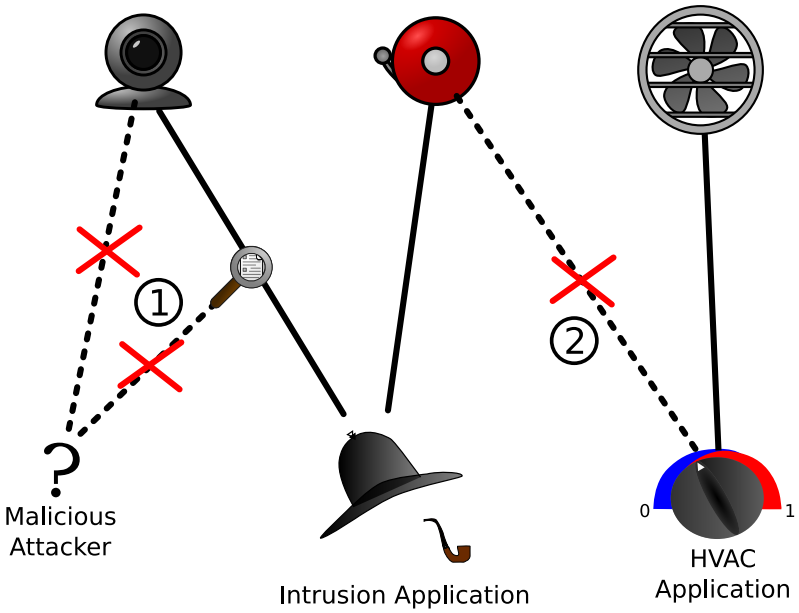

Figure 2: Security goals in a smart home

2.3 DOMAIN-SPECIFIC SECURITY REQUIREMENTS

The security of a system is defined through security policies that state what is and is not allowed [5, 6].

Security is an important and very broad domain in computer science, and in information technologies (IT). Since more and more business includes the transfer of digitalized data, the need for protection rises [23, 82]. Still, most users ignore or are unaware of the danger of unsecured IT devices such as smartphones or computers. In the worst case (*e.g.*, a hacked and virus-infested computer), most users will simply turn it off and have the operating system reinstalled. Home automation systems however, surround the user with technology that directly affects him. Simply turning the system off can have severe consequences, such as lack of emergency management during a fire, no HVAC[7] during winter, *etc.*

The problem is that most approaches concentrate on building useful applications or improving functionality, and therefore neglect security concerns [10]. Figure 2 shows two typical scenarios that should be prevented: (1) a malicious attacker accessing devices directly (*i.e.*, cameras), or sniffing on unsecured exchanged data (*i.e.*, video stream), and (2) an application manipulating devices it should not be able to access (*i.e.*, an HVAC application accessing the alarms). A worst case scenario would be a malicious attacker taking over a residential gateway and thus, transforming the smart home into a distributed surveillance system giving away too much information about the occupants [47]. This sce-

6 In this case, *failure* means that the smart home did something that was not expected by the occupants.

7 Heating, Ventilating, Air-Conditioning

nario is not that far-fetched: recently a car system[8] was hacked by scientists via Bluetooth, giving them complete control over the car (*e.g.*, dashboard, brakes, engine, *etc.*) [52]. It is evident that security must not be taken lightly in these domains.

"A security mechanism is an entity or procedure that enforces some part of the security policy." Bishop [5]

Two security concerns are of particular interest to pervasive computing: *access control*, because old models and mechanisms do not address certain attributes of pervasive computing [10], and *privacy*, which seems to be incompatible with the pervasive computing paradigm [11].

2.3.1 *Access control*

Access Control has received much attention in the pervasive computing domain, because approaches and mechanisms that worked well in the past, such as Unix file system permissions, do not address certain attributes of pervasive computing systems. The most important challenge and difference is that a pervasive computing environments combines the virtual and the physical world [71, 72]. Thus, the permission to do something does not only depend on the identity of the user, it also depends on contextual information. For example, a ten year old child may only have the right to use the cooker when an adult is present in the kitchen. Additionally, pervasive computing systems require physical entities to enforce access restrictions, *e.g.*, cutting off power to the cooker or locking the kitchen door.

Another problem that physical entities impose is safe resource sharing. Pervasive computing implies the decoupling of services and devices [66]. A door, for example, offers the services *lock* and *unlock*. Decoupling these services from the door implies, making them available to every application running in the system. This enables resources to be used in new ways, for example, a security application can use a keypad to let the user type in a code to lock/unlock the doors, while an emergency application automatically unlocks doors during emergency situations like fires so as to ensure the evacuation of the occupants. The problem lies in the fact that applications access resources without any coordination between them. In this situation, it is very common for a resource to be accessed by multiple applications, potentially leading to conflicts. Resolving conflicts with simple strategies like first in first out (FIFO) and time-sharing makes no sense in this case, since this kind of strategy does not take the current context into account: a FIFO strategy will therefore not prevent the security application from locking doors during a fire, for

8 Car systems are quite similar to smart homes, as they use several deployed sensors and actuators to support the driver.

example. Moreover, these strategies require the virtualization of a resource which is not always possible for physical resources.

Since smart environments are multi-user systems, another important aspect is the collaboration between users to obtain certain rights [86]. This aspect is not addressed in this thesis.

2.3.2 *Privacy*

Privacy is a key concern in our modern society. Nobody likes the idea that personal data is available to everybody, everywhere. Therefore, the Organization for Economic Cooperation and Development (OECD) Privacy Guideline [53] and the European Union Data Protection Directive 95/46/EC [26] give a more precise definition of privacy in the context of information technology. Both specify that the collection of personal data should be limited and that people must be aware of and have agreed to the collection of such data.

Cas [11] shows the incompatibility of these privacy principles with pervasive computing. While the privacy principles protect the dissemination of data, pervasive computing tries to acquire as much data as possible about people to provide services that match their needs. This uncontrolled data collection through sensors and devices of pervasive computing environments poses a threat to privacy, especially if the data can be exploited by malicious persons [10, 74].

To control the dissemination of personal data, and therefore protect privacy, a number of mechanisms, mainly in relation to *confidentiality*, *integrity*, and *authentication*, are applied [5, 23]. Confidentiality mechanisms are used to encrypt the data to make it unreadable by third parties, integrity ensures the data cannot be manipulated without detection, and authentication verifies the identity of users or applications that access the data. Identity management techniques like anonymization of user data and the usage of pseudonyms are not addressed in this work [11, 23].

2.4 DEVELOPING FUNCTIONAL REQUIREMENTS

Application development must be supported throughout all phases of the development process, which include design, implementation, testing, and deployment phases [85]. This support must target the specific challenges of the pervasive computing domain (Section 2.2).

Communication between devices is the essence of pervasive computing and must be supported by the approach. More specif-

ically, it must cover the *heterogeneity* of the devices (*e.g.*, different platform[9]) and the *combination of technologies* that are used (*e.g.*, Bluetooth, Ethernet), and must provide service/device discovery to cope with the *dynamic* nature of the environment (*i.e.*, devices entering and leaving the environment) [13, 93].

To develop user-friendly and useful applications, the developer needs support to implement the functionalities of applications. In the home automation domain, the functionalities rely heavily on the orchestration of various devices, on interpreting events, and on remotely issuing commands. Adequate abstractions and mechanisms for these central aspects are required.

To make smart homes a success, they must be able to easily adapt to the different user preferences and requirements. Developers need support to make their applications customizable, flexible, and extensible. This requires easy parameterization and configuration of applications, *e.g.*, to test different interaction mechanisms (*e.g.*, tablet, gestures, voice), for example, or to add new devices.

Without support in these areas, more time will be spent on making applications work, rather than on creating useful applications.

In the following, we present existing approaches concerning the support in developing the functionalities of applications. Section 2.5.1 revisits the approaches with a focus on security aspects.

2.4.1 *Middleware*

The general idea of middleware is to make services or application components remotely available [4]. This feature is of particular interest if an application uses several components (or other applications) that run on distant servers. To facilitate the creation of such a distributed application, a middleware provides high level primitives that abstract over the location and platform of networked hosts that host the required components [25].

To do so, middleware, as the name suggests, sits between the application components and the operating system. Figure 3 shows the middleware pattern. For the application components, the middleware provides an application programming interface (API), the previously mentioned high level primitives. The platform interface connects the middleware to a specific platform. Application components communicate with the middleware, and are unaware of the underlying platform and the

9 A platform consists of an operating system (OS) and a processor architecture, *e.g.*, Linux and Intel x86.

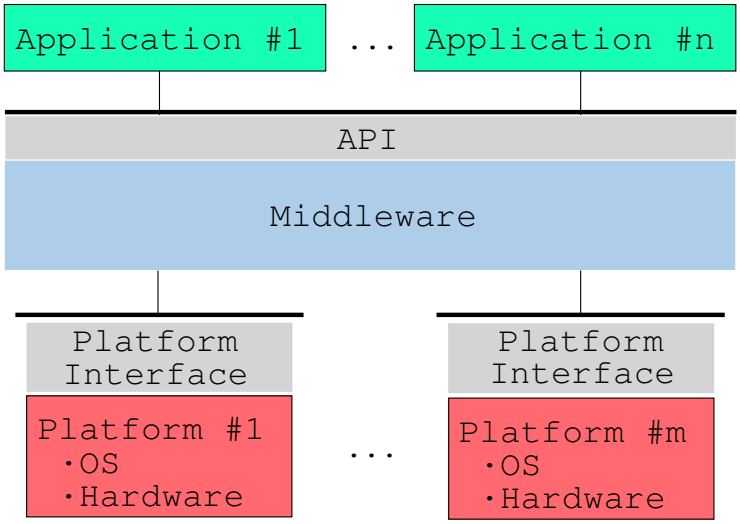

Figure 3: Basic concept of middleware

location of other components. Depending on the location of a required application component, the middleware either forwards a request to a local component, or uses the platform interface to send it over the network (Figure 4).

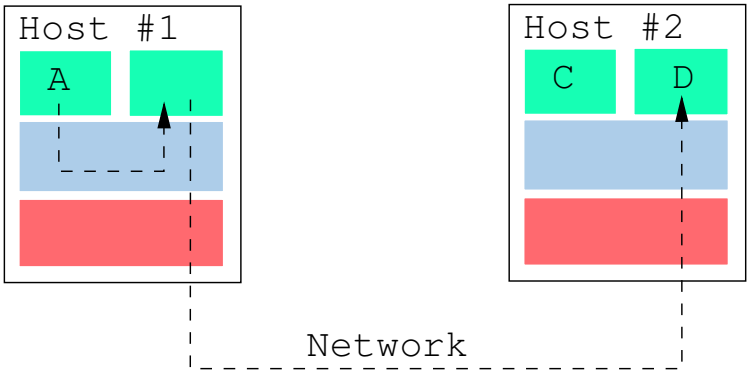

Figure 4: Transparent communication (local: A to B, distant: B to D)

Middlewares provide a basis from which to design home automation systems, in that they promote distribution. But they do not cover everything. Indeed, middlewares were developed to build fixed distributed systems, that is, systems in which everything is connected at the development stage. This limits their ability to support the dynamic nature of pervasive computing systems. Also, middlewares neglect the design phase of software development and provide no support for functionalities and evolution of applications.

2.4.2 *Gaia/Olympus*

Gaia is a middleware that was specifically developed for active spaces by the research group led by Roy H. Campbell at the University of Illinois at Urbana-Champaign [67]. Campbell *et al.* define *active spaces* as physical spaces with clear boundaries that contains physical objects, networked devices and users. Gaia is seen as the operating system for an active space, in that it abstracts the space and all the resources within it as a single programmable entity. It uses the Common Object Request Broker Architecture (CORBA [54]) to handle interactions between entities in the active space.

To support user mobility, Gaia keeps track of the applications a user is running, and the data he is using, in so-called *sessions*. These sessions are dynamically mapped to the available resources in an active space, for example when the user enters.

To cope with the typical problems of middlewares (*e.g.*, static setting, *etc.*), Ranganathan et al. [62] have developed a high level programming model, Olympus, that builds upon Gaia. Olympus has two main features:

- Semantic entity discovery – Developers specify space entities (services, applications, devices, physical objects, locations, users) in an ontology. At runtime these abstract entities are resolved to real entities taking the current context, space policies, and user preferences into account. This facilitates the portability of applications.
- High level space operations – Common operations (*e.g.*, start, stop, user moves) are included in the programming model. As a result, developers can simply use these operations, without worrying how they are actually performed in an active space.

The limitations of Olympus are the lack of support during the design phase and the fact that the space operations cannot be extended.

2.4.3 *Aura*

While Gaia was centered on the active space, the Aura Project led by David Garlan at Carnegie Mellon University proposes a user centric approach [28, 80]. The idea is that each user has a personal Aura that supports him in accomplishing his tasks in a pervasive computing environment. To achieve this, Aura interprets these tasks and maps them to services that are available in the current environment. For example, the task "writing an

article" would be interpreted as *editing text* and then mapped to Notepad[10]. During this mapping process, other information is also taken into account, such as the quality of service, context changes, mobility.

Treating user tasks as first class citizens facilitates the work of the developer in designing user-centric applications, because he does not have to consider which actual services will be used at runtime. However, this automatic mapping is accomplished through the introduction of certain constraints. Service suppliers must conform to the uniform and proprietary Aura API. As a result, services require a wrapper and an abstract service description: Notepad and Emacs are both described as text editors, for example. Another limitation of Aura is the lack of support in implementing services and context components. This makes system upgrades costly.

2.4.4 *Centaurus/Vigil*

Finin *et al.* have developed Centaurus, an infrastructure and communication protocol for providing services to heterogeneous mobile clients in smart environments [38, 41]. Centaurus was specifically designed for flexibility in communication to address the properties of pervasive computing. Like middlewares, Centaurus does not provide particular support for designing and implementing applications. Vigil is based on the Centaurus infrastructure and has a strong focus on security [39, 40]. The fact that Centaurus was built from scratch, has made it possible to easily integrate a number of security services into Vigil. The security features of Vigil are detailed in Section 2.5.1.

2.4.5 *Ponder/Ponder2*

The research group working with Morris Sloman at the Imperial College at London has its roots in distributed systems management. Their approach differs from those taken previously, in that it focuses on describing the behavior of a distributed system in the form of policies. Thus, they have developed the policy language *Ponder* to describe authorization and obligation policies [18, 19]. Obligation policies specify actions that entities *must* or *must not* do, while authorization policies specify actions that entities *can* or *cannot* do.

Ponder2[11] is based on the original Ponder language, but has been reimplemented and augmented with additional tools and

10 Or any other text editor that is available

11 http://ponder2.net

functionality to improve its applicability in pervasive computing systems [49]. Resources such as sensors, devices, *etc.* are wrapped by *Managed Objects* that are implemented in Java. The Managed Objects create events, which are then interpreted by the obligation policies, which can trigger actions on Managed Objects. Authorization policies intercept action requests to enforce access control on the Managed Objects.

The implementation of applications is done in two steps: (1) policies are specified in Ponder and Managed Objects are implemented in Java, (2) the system is instantiated using a dedicated language, PonderTalk, which provides the necessary parameters. While this separation promotes reuse of existing components, and facilitates the task of deploying the system with different configurations, it offers no specific support for pervasive computing. Another limitation is the lack of support during the design phase.

2.4.6 *PervML*

PervML is a model-driven development (MDD) approach for context-aware pervasive computing systems that has been developed by the research group led by Vicente Pelechano at the Technical University of Valencia [75]. Their approach covers the entire development process, for which they have identified and separated specific tasks and assigned them to different roles. For each aspect of the application development, they propose an adequate model. The developer designs various models on the basis of which the application is fully generated in Java and OWL[12], and then deployed using OSGi[13].

PervML provides support throughout the entire development process. The downside of the MDD approach is that the developer has to master several different technologies, as an application requires: (1) a description in the form of a UML[14] class diagram, (2) pre and post conditions, and triggers to be expressed in OCL[15], (3) a graphical state transition diagram, (4) a UML interaction diagram, (5) deployment information in the form of a location model, and (6) optional non-functional properties in the form of security policies that define the access rights of users in the system. The extend to which such an MDD approach that avoids general-purpose languages facilitates the work of the developer is questionable [60].

12 Ontology Web Language, http://www.w3.org/2004/OWL/
13 Open Service Gateway initiative, http://www.osgi.org/
14 Unified Modeling Language, http://www.uml.org/
15 Object Constraint Language, http://www.omg.org/spec/OCL/2.2/

2.5 SECURITY CONCERNS IN APPLICATION DEVELOPMENT

One piece of advice concerning security is to be found again and again in the software engineering literature: security has to be considered from the very beginning of a project [3, 5, 82, 85].

This is particularly difficult in pervasive computing, because a pervasive computing environment consists of many devices and applications, and evolves constantly. Additionally, security concerns typically impact all applications, as well as the core system itself, at various places.

The last section presented existing approaches to developing the functionalities of applications. The main idea was to provide the developer with abstractions to hide low level details. Security impacts the code at these low levels. To facilitate securing a system, the approach has to support basic security mechanisms and hide implementation details, just as middlewares hide communication. Preferably, the developer only configures the mechanism, to avoid changes at the code level of the middleware or the applications. Another important aspect is that security requires system-wide implementation and configuration. To be able to easily adapt security aspects without impacting the functionality of the application, the code that manages security should be separate from the application logic.

Section 2.3 described the reasons behind the need for privacy and access control for home automation systems. As a result, mechanisms for access control, integrity, confidentiality, and authentication should be supported[16].

Concerning the design and implementation of security mechanisms, Saltzer and Schroeder [70] presented eight principles that are widely accepted in the security domain [5, 85]. These principles are: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. We provide more precise definitions when we justify our design and implementation later in this work (Chapters 5, 7).

2.5.1 *Practical approaches*

Some middlewares provide support for security aspects. For example, in the Enterprise Java Bean (EJB) component model [83], EJB containers provide support for security through encryption and authentication. Such component models are either dependent

16 As previously mentioned, privacy is mainly based upon integrity, confidentiality, and authentication.

on a specific middleware or provide little development support, if any.

Concerning security, Gaia offers an authentication service that separates the protocol (*e.g.*, Kerberos[17]) from the mechanism (*e.g.*, retina scan) allowing easy extensions of both parts. Additionally, Sampemane [71] developed an access control model dedicated to smart spaces that takes the presence of multiple users into account, and integrated it into Gaia. The access control mechanism requires a configuration in the form of security policies. Gaia offers no support in creating coherent policies, concerning the deployed applications and resources. As a result, this task is error prone.

Vigil [39] is based on the Centaurus infrastructure and stands out for its integration of security mechanisms. Specifically, it adds a number of components to the system that provide security services: a *Certificate Authority* issues certificates to new clients, a *Capability Manager* responds to access control and delegation requests, a *Trust Agent* keeps track of the currently delegated rights and enforces security policies. The policies are specified in *Rei*, a policy language dedicated to pervasive computing [42]. Rei provides constructs for expressing rights, prohibitions, obligations, and dispensations. Rei also provides tools for policy analysis and consistency checking, and supports meta policies to define priorities between policies, so as to resolve conflicting modalities, for example [57].

Ponder supports positive and negative obligation and authorization policies by default, and enforces access control on entities. Ponder, like Rei, provides policy analyses and consistency checking. In [69] Russello *et al.* presented an approach to resolving certain types of conflicts in Ponder policies. Recently, Sloman and Lupu [79] introduced the *Self-Managed Cell* (SMC) as an architectural pattern. An SMC features an optional security service that provides authentication, confidentiality, and anomaly detection.

PervML provides little support for security: the developer can define access rights for users on applications in the form of security policies. In the case of Aura, security was not considerd. Both approaches concentrate on the functionality of applications.

2.5.2 *Formal approaches*

Concerning access control, the approaches mainly focus on taking additional information, also referred to as *context*, into account to derive access rights (Section 2.3.1). Rashwand and Mišić [63] have designed a framework for access control that takes context

17 http://web.mit.edu/kerberos/

information and user intent into account to meet the requirements of pervasive computing. A similar work is presented by Kumar et al. [46], who extended the well-known Role Based Access Control (RBAC [73]) model to include context sensitivity.

Park and Sandhu [55], the designers of RBAC, go one step further by defining a completely new model, which they call the $UCON_{ABC}$ usage control model[18]. They concentrate on the "essence of usage control" represented by authorizations, obligations and conditions and provide a "fresh look at the fundamental nature of access control itself". Jin et al. [37] have extended this model to introduce the delegation of rights, an important mechanism for multi-user systems.

Specifically for the home automation domain, Gupta et al. [31] present a criticality-aware access control model that proactively adapts access rights when critical events occur, such as, for example, allowing everybody to unlock the basement door to ensure safe evacuation of the building during an earthquake.

To protect sensitive data, Hengartner and Steenkiste [34] propose an access control model for information that takes the relation between information into account to derive access rights. To do so, the relations between information are formally represented in the Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA [16]). In [35] they presented their access control algorithm that prevents privacy violations that may be caused by context-sensitive services (for example, the location of a user can be revealed by his public calendar).

Privacy is difficult to achieve, since it depends on several mechanisms and contradicts the ubiquitous computing paradigm [11]. Langheinrich [47] proposes to integrate privacy into the design of pervasive computing systems by using six design principles – notice, choice and consent, proximity and locality, anonymity and pseudonymity, security, and access and recourse. Langheinrich [48] opts for a privacy-aware system, which aims to achieve a reasonable compromise between privacy and ubiquity. To implement this *privacy by design* idea, the mechanisms that are necessary for privacy have to be provided by the development approach for home automation systems.

2.6 SUMMARY

This chapter presented requirements for developing applications in the home automation domain, specifically concerning func-

18 UCON stands for usage control, while *A* represents authorizations, *B* obligations, and *C* conditions.

tionalities and security aspects. To sum up, the core issues are as follows:

1. Specific programming support is needed to facilitate development of functional requirements. Without support, developers have to spend a lot of time and effort on communication and orchestration issues, complicating evolution and maintenance.
2. Security concerns must be dealt with from the beginning of a project, that is to say from the design phase. Most approaches neglect the design phase completely and only concentrate on implementation. As a result, no high level system information is available, making it necessary to configure security mechanisms at low level.
3. Security mechanisms affect the code of applications and the home automation system at multiple places. Developers require support and guidance to implement and configure security mechanisms efficiently and without interfering with the functionalities[19].

19 In the same way as with middlewares in relation to communication.

3

PROBLEM STATEMENT

At the end of the previous chapter we defined three main issues that have to be addressed: development support for functionalities, early integration of security concerns, and support in implementing security mechanisms on a system-wide basis.

To tackle these three issues, our approach leverages DiaSuite, a toolkit for developing pervasive computing systems that has been developed by the Phoenix research group. DiaSuite covers the entire development lifecycle: design, implementation, testing, and deployment. The design specification of an application is used to generate a dedicated programming framework that provides support and guidance to the application developer in relation to functional requirements.

OUR APPROACH

In this thesis we propose to extend our design language to manage resource conflicts throughout the development process. The design specification of a software system is enriched with conflict-handling declarations. These declarations are used to automatically detect resource conflicts, provide support for implementing their resolution, and generate and parameterize the enforcing mechanism. As a result, the conflict-handling logic is separated from the application logic and the developer is liberated from low level tasks, such as implementing and configuring the enforcing mechanism.

To facilitate the integration of security aspects into a pervasive computing system, we have identified several places in the core system and the generated framework where security code has to be added. We combine abstractions for these places on design level with aspect-oriented programming to enable easy integration of security aspects system-wide. As a result, the developer is shielded from low level implementation details of the middleware and the dedicated framework.

ASSUMPTIONS

Privacy is very complex and difficult to achieve in pervasive computing. In this thesis we limit our solution to improving

privacy in smart homes, where occupants have full control over the deployed devices and the collected data. We do not consider the problems that occur in public pervasive computing environments such as shopping malls, for example, where privacy is more difficult to protect due to the permanent recording by third parties [11]. To handle these problems, identity management techniques such as anonymizing and pseudonymizing of data can be used [11, 23]. Additionally, the information repository can be kept local [77], *i.e.*, ensuring that data collected in the shopping mall will not be used elsewhere. Other important concepts are trust and reputation between users and the data collector [30].

4

DIASPEC

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.

— Martin Fowler [27]

This chapter introduces DiaSpec [13], our design language for pervasive computing systems that is part of DiaSuite[1]. DiaSuite is a suite of tools to support the development process. Specifically, the DiaSuite approach covers the entire development lifecycle and provides guidance and support to implementing the application logic. We present how to design and implement an application that is refined with security concerns in the chapters ahead.

4.1 THE SENSE-COMPUTE-CONTROL (SCC) PATTERN

The DiaSpec language enforces an architectural pattern, named sense-compute-control, commonly used in the pervasive computing domain [20]. This pattern distinguishes three types of components, as depicted in Figure 5: (1) *resources*, which provide sensing and actuating capabilities on a pervasive computing environment[2], (2) *contexts*, which aggregate and process sensed data, and (3) *controllers*, which receive information from contexts and invoke actuators. This architectural pattern goes beyond the pervasive computing domain and enables high-level programming support and a range of verifications [12, 14, 15, 29].

Figure 6 shows how a DiaSpec description drives a five-stage development process. (1) A domain expert declares a taxonomy of resources that can be found in the pervasive computing environment. (2) An architect describes the interactions between resources, contexts and controllers. Given a taxonomy and an application design, a compiler, named DiaGen, generates a customized programming framework in Java. (3) The generated framework is used by the developer to implement the application.

1 DiaSuite is freely available http://diasuite.inria.fr and open source.

2 Resources are devices (*e.g.*, a motion detector) or software components (*e.g.*, an address book).

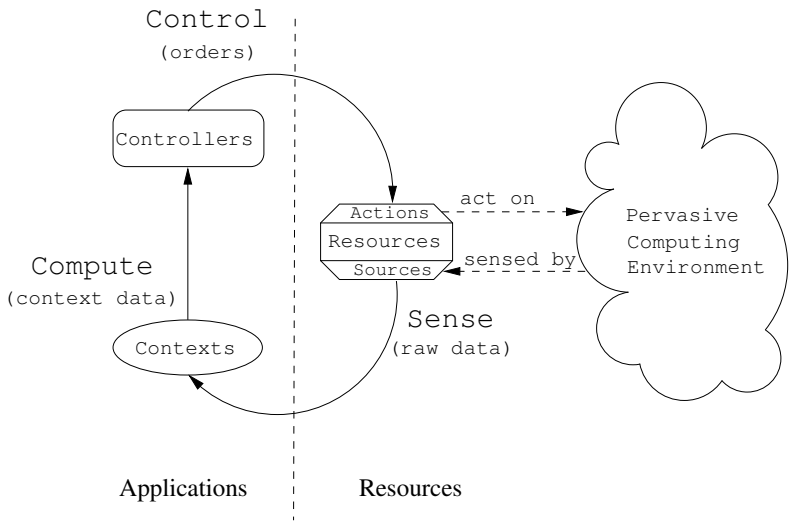

Figure 5: DiaSpec design pattern

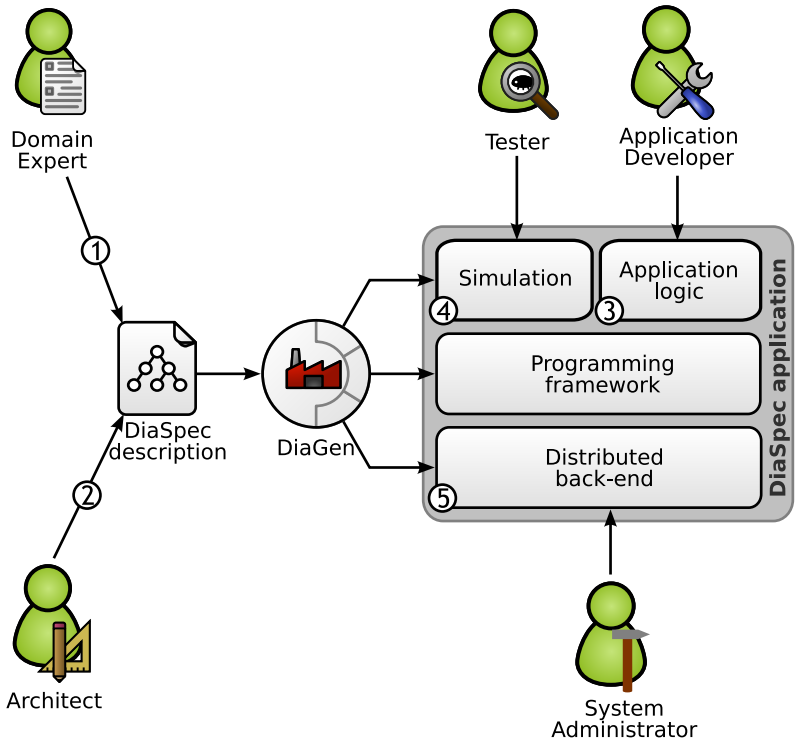

Figure 6: DiaSpec development cycle

(4) The application code can be tested as is, prior to deployment, using a simulator for a pervasive computing environment, named DiaSim [7]. (5) A system administrator can deploy the application in a real pervasive computing environment.

We now focus on the first three steps of our development process with an application that treats different types of emergencies in a building.

4.2 DESCRIBING THE ENVIRONMENT

First, the domain expert declares the available resources of a pervasive computing environment, as is done using an interface description language (*e.g.,* WSDL[3]) to declare external resources. In DiaSpec, this process is supported by a language layer dedicated to describing classes of entities that are relevant to a given application area. An entity declaration models sensing capabilities that produce data, and actuating capabilities that provide actions. Specifically, a declaration includes a data source for each one of its sensing capabilities. An actuating capability corresponds to a set of method declarations. Additionally, attributes are included in an entity declaration to characterize properties about instances (*e.g.*, their location). Entity declarations are organized hierarchically, allowing entity classes to inherit attributes, sources, and actions.

Listing 1 shows an excerpt of the taxonomy for the emergency application. In particular, to detect an intrusion, the application uses several classes of entities that are deployed in the home: keypads, cameras, break detectors, alarms, doors, and messenger services. Keypads tell if the home is in the security mode. Cameras are used as motion detectors, and to track and take pictures of the intruder. The break detectors indicate broken doors and/or windows. Upon the detection of an intrusion, the doors are locked, the alarms are turned on, and the occupants are notified via a messenger service.

The domain expert introduces the resource classes with the *device* keyword. Lines 2 to 4 of Listing 1 define a root device, which introduces the location attribute. Attributes serve mainly as filters for entity discovery in the pervasive computing environment.

The *source* and *action* keywords define the capabilities of an entity. Line 12, for example, declares that cameras provide a boolean value to the smart home, indicating the presence of people. Cameras also provide the action `Move` (line 14) that is further detailed in lines 39 to 43.

3 Web Service Description Language http://www.w3.org/TR/wsdl20

```
/* Description of the available entities. */
device LocatedDevice {
  attribute location as Location;
}
device Alarm extends LocatedDevice {
  action OnOff;
}
device BreakDetector extends LocatedDevice {
  source broken as Boolean;
}
device Camera extends LocatedDevice {
  source presence as Boolean;
  source picture as JPEG;
  action Move;
  action Track;
}
device Door extends LocatedDevice {
  source status as DoorStatus;
  action LockUnlock;
}
device Keypad extends LocatedDevice {
  source status as HomeStatus;
}
device Logger {
  action Log;
}
device Messenger {
  action Send;
}

/* Description of the supported actions. */
action LockUnlock {
  lock();
  unlock();
}
action Log {
  logEvent(event as String);
}
action Move {
  roll(degree as Integer);
  pitch(degree as Integer);
  yaw(degree as Integer);
}
action OnOff {
  on();
  off();
}
action Send {
  send(message as String);
  send(picture as JPEG);
}
action Track {
  trackPresence();
  stopTracking();
}
```

Listing 1: Extract of the emergency management taxonomy used by the intrusion module

```
1  /* Context components. */
2  context Occupancy as Boolean indexed by location as Location {
3    source presence from Camera;
4    source status from Keypad;
5  }
6  context Intrusion as Boolean indexed by location as Location {
7    context Occupancy;
8    source broken from BreakDetector;
9  }
10 context Surveillance as JPEG indexed by location as Location {
11   source picture from Camera;
12 }
13
14 /* Controller component. */
15 controller IntrusionCtrl {
16   context Intrusion;
17   context Surveillance;
18   action LockUnlock on Door;
19   action Log on Logger;
20   action OnOff on Alarm;
21   action Send on Messenger;
22   action Track on Camera;
23 }
```

Listing 2: Extract of the design description of the intrusion module of the emergency application

4.3 DESIGNING THE APPLICATION

To support application design, the DiaSpec language offers a language layer based on the design pattern depicted in Figure 5, and comprises resource, context and controller components.

To illustrate the design layer, let us examine the emergency application. Listing 2 presents an excerpt of the corresponding DiaSpec declarations, describing the intrusion module. Figure 7 shows a graphical view of the intrusion module. The arrows indicate the flow of information. The resources at the bottom of the diagram provide information to context components; the resources at the top provide the controller components with actions on the environment.

The cameras in the rooms transmit a boolean value to the `Occupancy` component indicating the presence of persons. Lines 2 to 5 introduce this component using the *context* keyword. The *source* declarations within define the input of this component, *e.g.*, `presence` from cameras (line 3) and `status` from keypads (line 4). The *as* keyword in line 2 is followed by the type of the output value, in this case `Boolean`. This value is indexed by a location: the room were the presence is detected. Another context component, `Intrusion`, uses the information provided by the `Occupancy` component and break detectors that are deployed at the doors

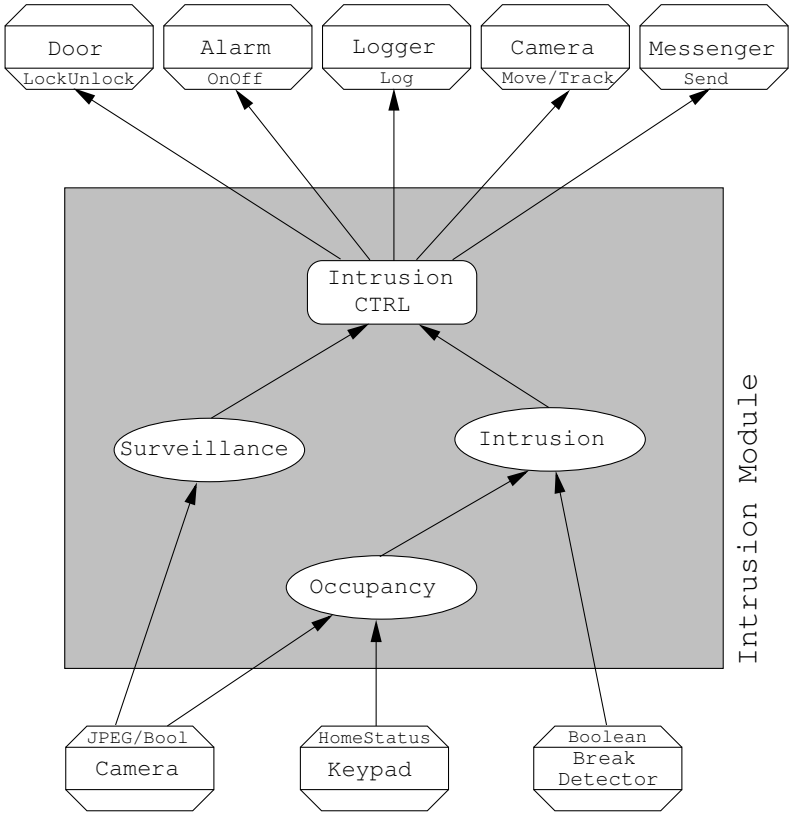

Figure 7: Design of the intrusion module

and windows (lines 6 to 9), to determine whether there is an intrusion. Eventually, if there is an intruder, the IntrusionCtrl component is invoked. It is declared by the *controller* keyword (line 15). This component declares two input sources using the *context* keyword and referring to `Intrusion` and `Surveillance`[4] (lines 16 to 17). The *action* keyword defines the actuator operations that can be invoked by a controller component. In our example, the `IntrusionCtrl` component can lock/unlock doors, turn on/off alarms, set cameras to the tracking mode, log events, and notify the occupants via messenger services (lines 18 to 22).

4.4 IMPLEMENTING THE APPLICATION

The DiaSpec compiler, DiaGen, is implemented using the ANTLR parser generator [56], and uses the declarations in the taxonomy and the application design to generate a dedicated Java programming framework and a software layer, DiaEnv.

4 The `Surveillance` component is used to retrieve pictures from the cameras.

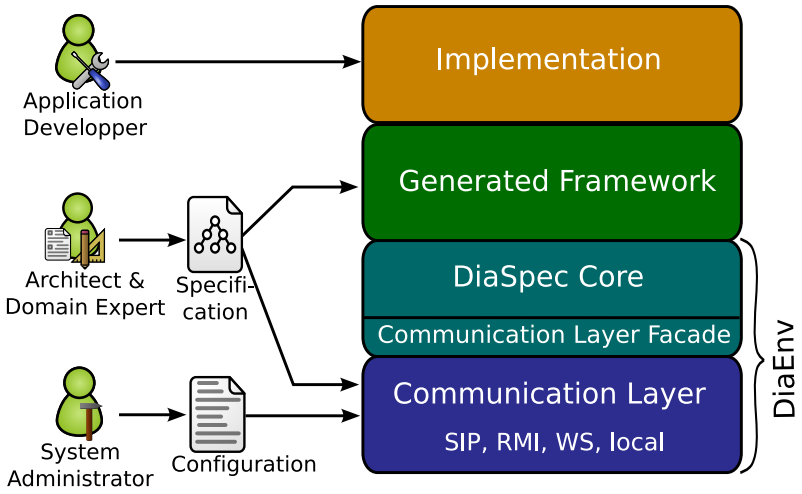

Figure 8: DiaSpec/DiaGen generated code structure.

4.4.1 *DiaEnv*

DiaEnv supports runtime execution of applications, *e.g.*, component registration and discovery. On entering the pervasive computing environment, a component registers at the local registry providing its signature, *e.g.*, capabilities and attributes. After a successful registration, the component can be acquired via the component discovery service.

In addition to performing runtime consistency checks, DiaEnv abstracts over the communication layer, allowing to transparently deploy a systems implementation. To do so, DiaGen leverages existing distributed-system technologies by generating glue code to customize them with respect to the needs of pervasive computing. Currently, DiaGen supports local, Web Services [17], RMI [21], and SIP [68] technologies, in addition to our simulation environment DiaSim [7] (Figure 8).

4.4.2 *The DiaSpec generated framework organization*

Given a software design declaration, the DiaGen compiler generates a typed framework on which to develop the distributed application.

For each action and (implicit) query declaration[5], DiaGen includes a Java interface defining the provided methods. Similarly, each type of event a component may subscribe to, creates a Java interface defining a listener to that event.

5 Whenever a component declares a source or context as input, it may implicitly query the concerning value.

```
1 public interface CameraTrack {
2   void trackPresence(RmiRemoteServiceInfo source) throws
        RemoteException;
3   void stopTracking(RmiRemoteServiceInfo source) throws
        RemoteException;
4 }
```

Listing 3: The Java interface generated for the `Track` action that was declared for the `Camera` device in Listing 1

For each component definition (device, context, controller), DiaGen includes an abstract class with abstract method definitions for the provided actions, queries, and event listeners. The developer implements the application logic by subclassing a DiaGen-generated abstract class. This pattern, generation gap [27], ensures a clean separation between programming support and implementation code.

To support the developer during the implementation, DiaGen generates a proxy interface of the providing component (*e.g.*, a device) that exposes a limited view of its capabilities (*e.g.*, sources and actions) to the requiring component (*e.g.*, a controller). This is done for each couple of components that are connected. In DiaSpec, all sources and contexts that are used as input imply a connection, as do actions that controller can invoke on devices.

The proxies only provide the methods of the connected interfaces. For example, Listing 2 declares that `IntrusionCtrl` uses the action `Track` of the `Camera` device (line 22). As a result, DiaGen generates a Java interface (Listing 3) that must be declared by both abstract classes `AbstractCamera` and `CameraProxy`.

Let us illustrate the implementation of the application logic by considering the declaration of the `IntrusionCtrl` component in Listing 2. This controller component relies on two contexts, `Intrusion` and `Surveillance`, and uses several devices to handle intrusions.

Listing 4 shows an implementation of this controller component. Most of the application logic can be found in the `onIntrusion` method (lines 8 to 24). The generated framework supports the developer with three parameters in this method (line 8):

- `IntrusionValue` contains the boolean value indicating an intrusion, and the values of the indices, in this case the location (Listing 2, line 6).
- `GetContextForIntrusion` contains a set of proxies of other context components the controller connected to. It is used

in line 19 to get a picture of the room where the intrusion has been detected from the `Surveillance` context.

- `DiscoverForIntrusion` contains a set of proxies that allows the controller to discover and act on instances of the deployed devices.

```
1  public class IntrusionCtrl extends AbstractIntrusionCtrl {
2
3    public IntrusionCtrl(ServiceConfiguration serviceConfiguration) {
4      super(serviceConfiguration);
5    }
6
7    @Override
8    public void onIntrusion(IntrusionValue intrusion,
         GetContextForIntrusion getContext, DiscoverForIntrusion
         discover) {
9      if (intrusion.value()) {
10       Location loc = intrusion.indices().location();
11       String message = "Intrusion detected at: " + timeStamp() +
12           " in room " + loc.toString();
13       discover.doors().all().lock();
14       discover.alarms().all().on();
15       discover.loggers().anyOne().logEvent(message);
16       discover.cameras().whereLocation(loc).trackPresence();
17       discover.messengers().anyOne().sendMessage(message);
18       discover.messengers().anyOne().
19           sendPicture(getContext.surveillance(loc));
20     }
21     else {
22       discover.alarms().all().off();
23     }
24   }
25
26   @Override
27   public void onSurveillance(SurveillanceValue surveillance,
         GetContextForSurveillance getContext,
         DiscoverForSurveillance discover) {
28     // Do nothing.
29   }
30
31   public String timeStamp() {
32     Calendar cal = Calendar.getInstance();
33     SimpleDateFormat sdf = new SimpleDateFormat();
34     return sdf.format(cal.getTime());
35   }
36 }
```

Listing 4: Implementation of the `IntrusionCtrl` component

Lines 13 to 19 show the various actions the controller executes if an intrusion has been detected. Note that the framework supports only interactions with components that have been specified in the application design (*e.g.*, `IntrusionCtrl` cannot access methods of the `Move` action of `Camera`).

The generated method `onSurveillance` (lines 27 to 29) does not require an implementation, since the concerning context component (*i.e.*, `Surveillance`) is not supposed to publish events. It is only used by `IntrusionCtrl` to retrieve current pictures from the cameras[6]. Cassou et al. [15] address this issue of unused methods by enriching the application design with interaction contracts between components. This allows a more precise description of the information flow within applications and results in a more specialized framework, *e.g.*, the `Surveillance` context would not provide the possibility to publish events. The interaction contracts are not detailed in this work, we only refer to them in Chapter 10 concerning future work.

The `timeStamp` method (lines 31 to 35) is a helper method that provides the current time that is used in the messages send to the occupants and the logger.

4.5 SUMMARY

This chapter presented DiaSpec, our design language for pervasive computing systems. We illustrated how the functionalities of applications are designed and implemented throughout the different development phases. But, as we mentioned in the previous chapter, functional requirements are not sufficient. Security concerns have to be addressed during the development process as well. The next chapters present our approach to integrate security concerns into the existing development process of DiaSpec. To do so, we reuse existing information, *e.g.*, the design specification, and introduce missing information in an unobtrusive way.

6 This detour is necessary to be conform to the SCC paradigm.

Part II

ACCESS CONTROL

5

RESOURCE CONFLICT HANDLING

In future smart homes functionality will be provided to the inhabitants by software services decoupled from the underlying hardware devices. While this will enhance flexibility and will allow to provide cross-functionalities across multiple devices it will also lead to resource conflicts.

— Retkowitz and Kulle [66]

Typically, a pervasive computing environment consists of multiple applications that gather data from sensing devices, compute decisions from sensed data, and carry out these decisions by orchestrating actuating devices.

The rapid development of new devices (*i.e.*, resources), and development tools opened to third-parties, have paved the way to an increasing number of applications being deployed in pervasive computing environments. These applications access resources without any coordination between them because a pervasive computing platform needs to evolve as requirements change. In this situation, it is very common for a resource to be accessed by multiple applications, potentially leading to conflicts. For example, in a home automation system, a security application that grants access to the home can conflict with another application dealing with emergency situations like fires, preventing the building to be evacuated. In fact, conflicts do not only occur across applications but also within an application. For example, different modules of an application may be developed independently of each other, creating a risk that conflicting orders to be issued to devices. Detecting, resolving and preventing intra- and inter-application conflicts is critical to make a pervasive computing system reliable. To do so, a systematic and rigorous approach to handling conflicts throughout the development lifecycle is required.

Detecting conflicts is a daunting task. Pervasive computing systems are complex and involve numerous applications that may conflict on one or multiple resources. Scaling up conflict handling for real-size pervasive computing systems requires to distinguish potential conflicts from safe resource sharing. This may depend on the type of a resource, for example, a conflict may occur on a resource providing mutually exclusive operations (*e.g.*, locking and unlocking a door). This may also depend on the applications

being deployed in a pervasive computing environment (*e.g.*, two applications may access a device inconsistently), precluding application developers from anticipating potential conflicts. Without any support, detecting potential conflicts requires to examine the code of all the applications to identify each resource usage, and determine whether it may conflict.

After potential conflicts are pinpointed, it is necessary to resolve each of them. It requires intimate knowledge about the code of the corresponding applications to resolve the conflicts by making code changes. Because of the lack of high-level programming support, writing system-wide conflict-handling strategies is often overlooked. This situation results in polluting the logic of applications with ad hoc code, compromising the system maintainability.

The situation is exacerbated by the fact that pervasive computing environments are prone to changes: applications as well as resources emerge, evolve, and may disappear over time. These changes directly impact conflict management. This problem is well known in the telecommunications domain where it was observed that the number of potential conflicts grows exponentially as new applications are added to an existing system [43]. Manually handling conflicts thus becomes impractical.

Overview of our approach

Managing conflicts is often decomposed into three stages: detection, resolution and prevention [43]. In practice, these stages crosscut the development lifecycle of applications and pervasive computing systems.

We introduce an approach to conflict management that covers the lifecycle of a pervasive computing system. It consists of a design method for applications, supported by declarations and tools, separating conflict management tasks. This approach facilitates the work of architects, developers and administrators: requirements for conflict management are propagated throughout the development stages.

Our approach leverages DiaSuite and extends our design language DiaSpec (Chapter 4) with conflict-handling declarations that allow domain experts to characterize resources from a conflict-management viewpoint. This information, in combination with the design descriptions, allows to automatically pinpoint places where conflicts can occur.

To resolve the detected conflicts, we propose to raise the level of abstraction beyond the code level, by providing declarative support for conflict resolution. Within an application, the developer

uses declarations to specify states for a pervasive computing system and order them with respect to their critical nature (*e.g.*, fire is more critical than intrusion). These states are enabled and disabled depending on runtime conditions over the pervasive computing system (*e.g.*, fire detection). State changes are used to update access rights to conflict-sensitive resources (*e.g.*, in case of fire, the fire module takes precedence over the intrusion module). Our approach is incremental in that states and priorities can be added as a pervasive computing system is enriched with new applications. Its declarative nature allows to prevent conflict-handling logic from polluting the application logic.

Conflict-extended architecture descriptions are used to generate customized programming frameworks. These frameworks guide and support the implementation of the conflict-handling logic. Generating the underlying framework from the architecture description guarantees that the architecture implementation can only access the required resources. Additionally, runtime support ensures that access to resources are granted in conformance with conflict-handling declarations.

5.1 DELIMITING RESOURCE CONFLICTS

The previous chapter introduced the intrusion module that is part of the emergency application. Another part of the emergency application is dedicated to detecting and managing fire situations.

Figure 9 shows a graphical view of the emergency application, including both modules, fire and intrusion. The fire module works as follows. Temperature sensors of a room send their values to the `AvgTemp` component. The value is indexed by a location: the room where the average temperature is measured. Another context component, `SmokeDetected`, gathers information from smoke detectors. Both contexts, the average temperature and the smoke information, are used by the `Fire` component to determine whether there is a fire in the home and its location. Eventually, if there is fire, the `FireCtrl` component is invoked. In our example, the `FireCtrl` component can lock/unlock doors, turn on/off alarms and sprinklers, and log events. The complete specification of the emergency application can be found in the appendix A (Listing 13 and Listing 14).

Let us now define our notion of resource conflict and examine the issues to be resolved within the DiaSpec development approach.

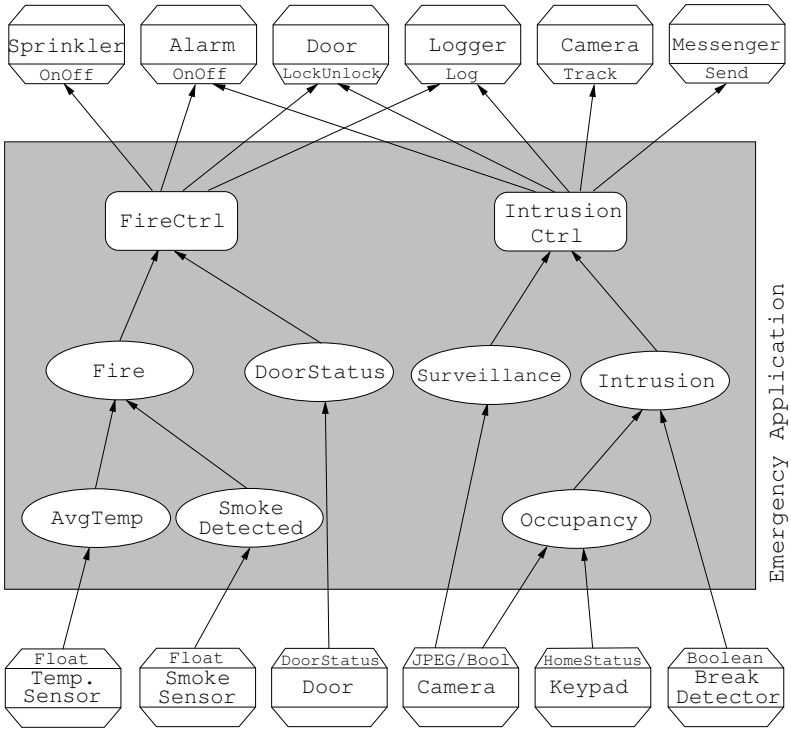

Figure 9: Architecture of the emergency application

Intra-application resource conflicts

Sensors and actuators need to be distinguished when it comes to resource conflicts. Indeed, sensors can sustain many consumers, requesting values either directly (*e.g.*, remote procedure call) or via some runtime support (*e.g.*, notification server). The situation would be comparable for actuators, if only they did not have side effects on the environment. This is illustrated in Figure 9, where the `FireCtrl` and `IntrusionCtrl` controllers share resources. These controllers can, for example, have conflicting effects on the door resource, depending on whether the current state of the pervasive computing environment requires anti-intrusion or firefighting measures.

What this example illustrates is that resolving resource conflicts relies on some notion of state that determines which consumer should acquire the resource. A pervasive computing environment can be in different states depending on a variety of conditions. Expressing these conditions is a key to providing a practical approach to conflict resolution. To separate this concern from the application logic, the approach should target the design level. In the door example, we would need to introduce states, enabled by conditions over relevant sensed data (*e.g.*, smoke intensity, motion detection). Based on the enabled states, the attempts of the controllers to acquire the doors would be prioritized.

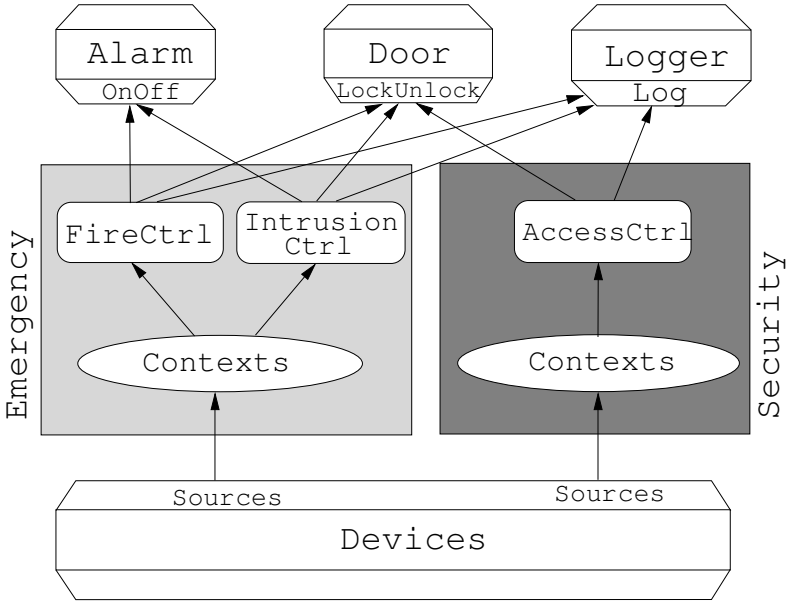

Figure 10: Potential resource conflicts between multiple controller components (intra and inter application)

Note that some actuators can be insensitive to conflicts. An example is the `log` action (Listing 1, lines 36 to 38): it can record data in any order, assuming each invocation has the necessary contextual information (*e.g.*, a time stamp).

Inter-application resource conflicts

The emergency application is only a part of the home automation system. The system administrator also deploys a security application to manage access to the home. Figure 10 shows a simplified, graphical representation of both applications: emergency and security. Both applications operate the same type of resources, in this case door and logger.

As can be noted, resource conflicts occur at different levels and must be managed globally. Even though, conflicting usage of resources can be resolved with respect to a given state, there needs to be a global, system-wide approach to combining unitary strategies in a transparent and predictable way.

5.2 CONFLICT MANAGEMENT

This section presents our approach to conflict management. It addresses the requirements discussed previously, and illustrates the approach with the home automation system.

5.2.1 *Detecting potential conflicts*

Our approach to conflict management revolves around the DiaSpec description of an application. Such a description exposes the interactions with actuators, allowing resource conflicts to be detected within an application, for the application developer, and between applications, for the system administrator.

Let us examine how the intra-application conflicts between the fire and the intrusion modules are solved (Figure 10). The process is the same for inter-application conflicts.

In DiaSpec, conflicts may occur when a resource is used by more than one controller component. Information about the resource usage can be extracted from the DiaSpec description of an application. This information needs to be refined to account for actions that are insensitive to resource conflicts (*e.g.,* the `Log` action).

Categorizing actions in the taxonomy

We extended the taxonomy language of DiaSpec with effect declarations for resource actions. An effect declaration applies to an action (*i.e.,* an interface and its associated operations), which is part of a device declaration. In practice, we have identified three main effects that need to be expressed. First, a device includes an action with operations that are mutually exclusive in their effects. For example, a door is either locked or unlocked. Such an action is declared with the *exclusive* keyword. Second, a device combines operations that interfere with each other. For example, a camera supports two actions: a tracking mode and standard movement operations; if used simultaneously, they interfere with each other. The list of interfering actions of a device is declared with the *interfering* keyword. Lastly, when an action is conflict insensitive, it is declared without effect keywords.

In our example, the domain expert has to enrich the declaration of `Door`, `Alarm` and `Sprinkler` with the exclusive keyword, and the `Camera` device with the interfering keyword, as is shown in Listing 5. The `Log` and `Send` actions are left unchanged because they are conflict insensitive.

Analyzing the architecture description

Given the taxonomy declarations enriched with conflict-handling information, the application developer and, later in the process the system administrator, investigate potential resource conflicts. A resource usage raises a potential conflict when two or more

```
1  device Alarm extends LocatedDevice {
2    exclusive action OnOff;
3  }
4  device Camera extends LocatedDevice {
5    action Move;
6    action Track;
7    interfering Move, Track;
8  }
9  device Door extends LocatedDevice {
10   source lockedStatus as LockedStatus;
11   exclusive action LockUnlock;
12 }
13 device Sprinkler extends LocatedDevice {
14   exclusive action OnOff;
15 }
```

Listing 5: Conflict-sensitive devices in the taxonomy

controllers may access it. These controllers may be defined within an application or across applications. In our approach, potential resource conflicts are automatically detected from a DiaSpec description. Conflict resolution is expressed with declarations, leaving the application logic unchanged.

5.2.2 *Declaring conflict resolution*

To resolve conflicts, we partition resource users with respect to a set of *states* in which a pervasive computing environment can be. These states are totally ordered with respect to their assigned priority level; they are associated with resource users (*i.e.*, controller components). For example, our home can be in either of the following states, listed in order of increasing priority: normal, security, or emergency. In doing so, applications and controllers, within an application, can be assigned different states, resolving their access to conflicting resources.

To complete our approach, we need to enable and disable states depending on evolving conditions of the pervasive computing environment. This is done by introducing *state component*, leveraging the DiaSpec notion of context component. Recall that such a component receives information about the pervasive computing environment (*e.g.*, smoke, fire, ...). A state component uses this information to determine whether the conditions for a given state hold, producing a boolean value.

Let us illustrate our approach with inter- and intra-application conflict resolution. Consider Listing 6 where two system state components are defined (lines 2 to 8): `SecuritySt` and `EmergencySt`. These components are declared with the *system* keyword to indicate that they apply system-wide, allowing the system admin-

```
1  /* System level (inter application) */
2  system state SecuritySt  priority 5 to Security {
3    source date from Calendar;
4  }
5  system state EmergencySt priority 10 to Emergency {
6    application state FireASt;
7    application state IntrusionASt;
8  }
9
10 /* Application level (intra application) */
11 application state FireASt priority 15 to FireCtrl {
12   source temperature from TemperaturSensor;
13   source smoke from SmokeSensor;
14 }
15 application state IntrusionASt priority 10 to IntrusionCtrl {
16   context Intrusion;
17 }
```

Listing 6: System and application state-components declarations

istrator to resolve inter-application conflicts. With the *priority* keyword, they are assigned priority values of 5 and 10, respectively, indicating that `SecuritySt` is less critical than `EmergencySt`. Following the *to* keyword are the applications to which the declared state applies. The conditions under which a state holds are parameterized by information sources, as is declared for the `SecuritySt` state with the `Calendar` source. As well, the conditions may be parameterized by other states, as is defined by the `EmergencySt` state with `FireASt` and `IntrusionASt`. In fact, these two states are used to resolve intra-application conflicts, promoting state-component reuse – these states are defined in lines 11 to 17.

Application state components are declared with the *application* keyword by the application developer and apply to controller components declared within an application. For example, the `FireASt` state applies to `FireCtrl` and `IntrusionASt` to `IntrusionCtrl`. Both controllers, and associated states, are local to the `Emergency` application. This local nature also applies to the priority defined by application states. That is, these priorities resolve conflicts within an application. In our example, these declarations prioritize `FireCtrl` over `IntrusionCtrl`. In doing so, intra-application conflicts for resources, such as doors, can get resolved.

5.2.3 *Implementing conflict resolution*

Declarations of conflict handling are enforced by additional code produced by DiaGen, shielding the application developer and system administrator from low-level implementation details.

Let us illustrate the implementation of the conflict-handling logic by considering the declaration of the `FireASt` state in Listing 6. This state component relies on two information sources, temperature and smoke, to determine whether the home is on fire.

```
1  public class FireASt extends AbstractFireASt {
2
3    private Map<Location, Map<String, Value>> status;
4
5    @Override
6    public void initialize() {
7      status = new HashMap<Location, Map<String, Value>>();
8      discoverTemperatureSensorForSubscribe.all().
           subscribeTemperature();
9      discoverSmokeDetectorForSubscribe.all().subscribeSmoke();
10   }
11
12   @Override
13   public void onTemperature(Temperature temperature,
         GetContextForTemperature getContext, DiscoverForTemperature
         discover) {
14     Map<String, Value> values = getValues(temperature.location());
15     values.put("temperature", temperature.value());
16     status.put(temperature.location(), values);
17     checkFire();
18   }
19
20   @Override
21   public void onSmoke(Smoke smoke, GetContextForSmoke getContext,
         DiscoverForSmoke discover) {...}
22
23   private Map<String, Value> getValues (Location loc) {...}
24
25   private void checkFire(){
26     boolean fireDetected = false;
27     for(Location loc : status.keySet()){
28       Map<String, Value> values = status.get(loc);
29       if(values.get("temperature").equals(Temperature.HIGH)
30         && values.get("smoke").equals(Smoke.HIGH)){
31           fireDetected = true;
32           break;
33       }
34     }
35     setFireASt(fireDetected);
36   }
37 }
```

Listing 7: An implementation of the `FireASt` state component

Listing 7 shows an implementation of this state component. In lines 8 and 9, the component subscribes to all the required sensors. To keep track of the home situation, the component stores temperature and smoke values from all locations within the home. Specifically, the component implementation updates the value (temperature or smoke) for each location (lines 13 to 21). After refreshing the value, it checks whether the condition for a fire holds by calling the `checkFire` method (line 17). This method determines whether or not a fire is occurring by publishing a boolean value (line 35), which in turn will enable or disable the corresponding state of the pervasive computing system (*i.e.,* `FireASt`).

Analyzing the resolution

Controller components use multiple resources to accomplish their goals. By defining states, a situation can occur where a controller component only has access to some, but not all resources it requires. In our example, if a fire and an intrusion are detected, `IntrusionCtrl` can send messages to the logging device, but cannot lock the doors nor turn on the alarms, since `FireCtrl` has the higher priority. While this can be ignored for some controller components, others may show unwanted behaviour. Currently our tool uses the information from the declared application and system states to check if such situations are possible. In the future we intend to add information at the controller components, whether a resource is mandatory or optional. This would allow a more precise analysis. Additionally our tool verifies that all potential conflicts are resolved.

5.3 IMPLEMENTATION

To achieve our conflict management approach, we have extended DiaSpec, DiaGen, and the DiaSpec runtime. The extended DiaSpec runtime is illustrated by our home automation example in Figure 11. We introduce a `ConflictCtrl` component that subscribes to state components to gather information about states that get enabled/disabled at runtime. It combines this information with state priorities to compute access rights, and update the `enforcer` components, associated with each resource class (*e.g.,* `Door`). The `enforcer` components intercept resource accesses and decide whether or not to grant them. Specifically, an `enforcer` component intercepts a method call and creates a request of the form `(controller, action, resource)`. Such a request is matched against an access control list (ACL) attached

to each resource class; this ACL comprises rules of the following form.

```
(controller, action, resource, [true|false])
```

When the request matches an ACL entry, the access to the resource is granted depending on the boolean value of the corresponding rule. Here we apply multiple security principles (Section 2.5): (1) we check every access (complete mediation), (2) we only grant required accesses[1] (least privilege), (3) we deny accesses if not specified otherwise (fail-safe defaults).

Globally, the conflict management process is performed in two stages. Statically, potential conflicts are detected based on the taxonomy and architecture declarations. For each detected conflict on a resource, the DiaSpec description is searched to identify state components dedicated to resolving it. This information is used to parameterize the `ConflictCtrl` component. Dynamically, this component will update the ACL of the `enforcer` component of all the resources impacted by a state change. The updated ACL is calculated based on the enabled/disabled states and their priorities. Here, our main goals concerning the security principles are (1) keep the mechanism as simple as possible (economy of mechanism), which is achieved in using a well-known mechanism (ACL) and a simple conflict resolution strategy by prioritizing states, (2) provide as much support as possible to make it easily applicable during the development process (psychological acceptability).

In the following we take a more detailed look into the implementation and illustrate how the intra application conflicts between the fire module and the intrusion module are resolved.

5.3.1 *Detecting and resolving conflicts at design time*

The information in an ACL is not sufficient to resolve conflicts, *e.g.*, the states and their priorities are not represented. Therefore we group access rules in policy objects. Besides the access rules, a policy contains links to the corresponding applications or controllers, a priority, and an activation condition in form of a state component.

The idea is to use available information, that is, the taxonomy, the design description, and the conflict-handling declarations, to generate a set of conflict free policies (Figure 12). Each policy contains rules concerning exclusive and interfering actions. Conflicts between policies are resolved by comparing priorities. All specified accesses that pose no problem are grouped in the

Application policies are local to an application and resolve intra application conflicts, while system policies do so on system level

1 The required accesses are described in the application design.

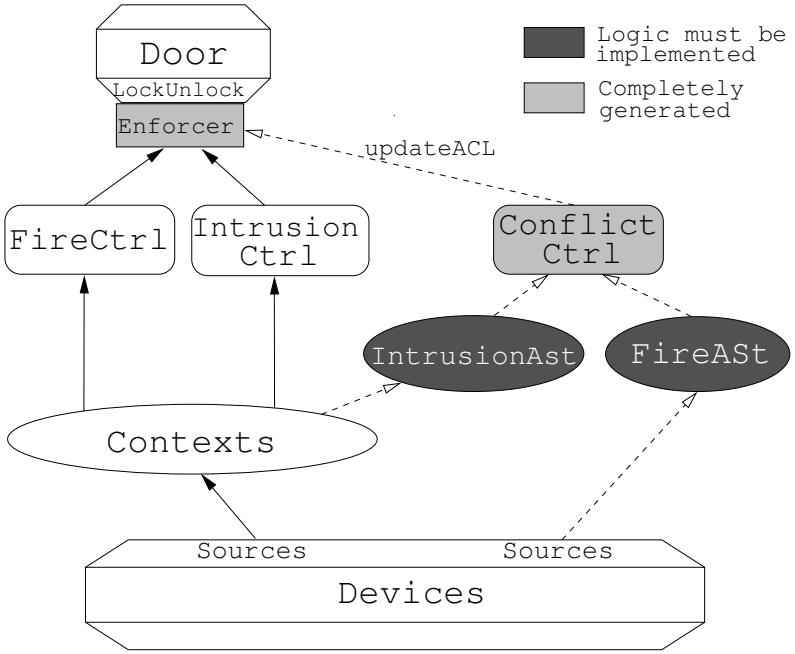

Figure 11: Extended DiaSpec runtime system

default policy. The default policy is always active and has the priority zero. Thus it is checked last. Figure 13 shows the decision process for an access request.

Verifying the resolution

To verify that conflict management is completed, the default policy is checked for untreated conflicts[2]. This detection algorithm can be separated into three steps.

1. It searches for rules that contain the same resource, but different controllers.
2. It checks for each found rule, using information from the modified taxonomy, whether the actions the controllers execute on the resource are marked as exclusive or interfere with each other. If so, a potential conflict has been detected.
3. It checks whether the controllers are part of the same application (intra application conflict) or not (inter application conflict).

For illustration, we show how the policies for the internal conflict of the emergency application are generated. In this (simplified) example, both controllers are allowed to operate the doors, the alarms, and the logger. For doors and alarms, this can cause conflicts. The application developer resolves the conflicts between both controllers by declaring application states (Listing 6). These

2 This is sufficient, since the generated policies are per definition conflict free.

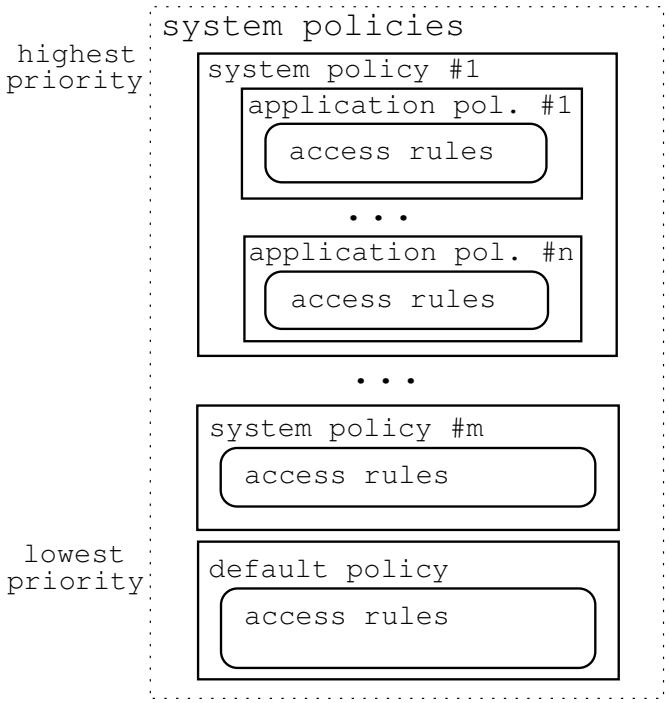

Figure 12: The generated policies in the system

dedicated declarations are now used to generate conflict free policies for both controller components.

Listing 8 shows the resulting policies that handle the conflict between `FireCtrl` and `IntrusionCtrl`. Line 1 declares that the emergency policy has the priority ten, and is active when the system state `EmergencySt` is true. Since the emergency application has an internal conflict, this policy contains two application policies, one for each conflicting controller. The `fire` policy allows `FireCtrl` to access the doors and the alarms and denies `IntrusionCtrl` to do so (lines 3 to 6). The `intrusion` policy does the inverse. Lines 16 to 19 define the default policy that contains all the non-conflicting resource accesses. In this case the rules that allow both controllers of the emergency application to access the logging device (lines 17 to 18). Since the logging device was declared insensitive for conflicts, the conflict handling for the emergency application is done.

5.3.2 *Enforcing the resolution at runtime*

At runtime, the `ConflictCtrl` component ensures that only authorized controllers can access resources and that conflicts are prevented. It contains all policies and connects automatically to all state components (Figure 11). After a change of a state component, `ConflictCtrl` activates or deactivates the concerned policy according to the published boolean value. Afterwards it

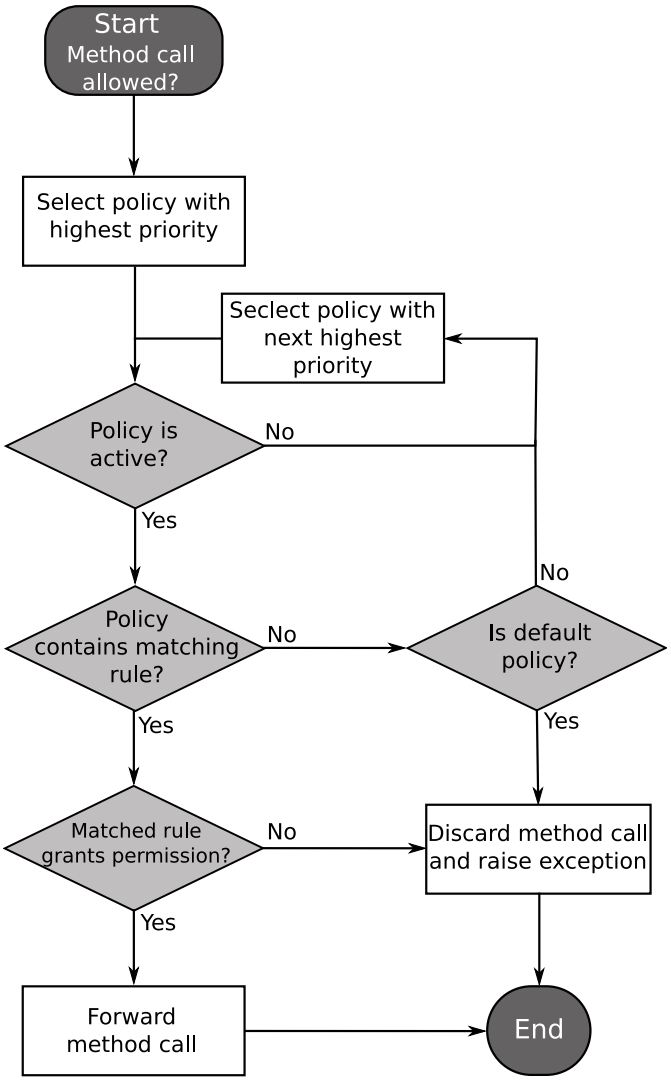

Figure 13: The decision process for an access request

derives the ACLs for the affected resource classes and proactively sends an update to the corresponding enforcer components.

Since resources in a pervasive computing system are likely to be battery-driven, we consciously burden the `ConflictCtrl` component that runs on a server (*e.g.*, the residential gateway). The overhead of computing ACLs is outweighed by the fact that on the resource side only a short list has to be checked for each method call. This ensures that the local overhead is very small.

Other works proposed this structure as well [69, 82, 92]. In these works, the central decision point is called the policy decision point (PDP), while the distributed, local enforcing points are called policy enforcing points (PEP). In our case, `ConflictCtrl` is the PDP, while the enforcer components are the PEPs. As proposed in [92], the enforcer components keep a small instance of a PDP (in our case the ACL), to be able to make a decision

```
emergency::10::[EmergencySt]{
  fire::15::[FireASt]{
    (FireCtrl, LockUnlock, Door, true);
    (IntrusionCtrl, LockUnlock, Door, false);
    (FireCtrl, OnOff, Alarm, true);
    (IntrusionCtrl, OnOff, Alarm, false);
  }
  intrusion::10::[IntrusionASt]{
    (FireCtrl, LockUnlock, Door, false);
    (IntrusionCtrl, LockUnlock, Door, true);
    (FireCtrl, OnOff, Alarm, false);
    (IntrusionCtrl, OnOff, Alarm, true);
  }
}

default::0::[true]{
  (FireCtrl, Log, Logger, true);
  (IntrusionCtrl, Log, Logger, true);
}
```

Listing 8: Policies for the emergency application

locally. This saves a lot of time and prevents the central PDP from becoming a bottleneck.

5.4 SUMMARY

Pervasive computing resources permeate the users lives including critical areas (*e.g.*, emergency management). The more the users rely on technology, the more important it is to ensure the reliability of said technology. The problem lies in the fact that applications that run in pervasive computing systems access resources without any coordination between them. Without proper resource management, no guarantees about the system behaviour at runtime can be given, *e.g.*, firefighting measures can conflict with anti-intrusion measures. A systematic approach to detect,resolve, and prevent these resource conflicts is thus necessary, to make a pervasive computing system reliable.

Our approach to addressing resource conflicts leverages DiaSuite and covers the whole development lifecycle. A DiaSpec design description already contains valuable information. To include missing information, we enriched DiaSpec with declarations dedicated to conflict handling. The added information is used to automate conflict detection, support the implementation, and generate and parameterize the enforcing mechanism.

The implemented mechanism prevents conflicts by enforcing ACLs for each pervasive computing resource. The ACLs are automatically updated according to the current state of the system.

6

EVALUATION

To assess the usability of our approach to resource conflict handling, we applied it to a building management system for an engineering school [7]. This case study was particularly interesting because it had been specified in DiaSpec and implemented, prior to the development of our approach. As a result, it could serve as a reference implementation, and a basis to be extended with our conflict-handling approach. We focus on the comprehensibility and reusability of conflict managing code, and the ability to detect conflicts. To test the correct behaviour of both implementations, original and extended, we used our pervasive computing simulator, DiaSim [7].

Separation between application logic and conflict handling

The original building management system was developed by members of our group who have expert knowledge in DiaSpec. They acted as architect, developer, administrator, and used their expertise to solve the foreseeable conflicts. The lack of proper support made them resort to ad hoc strategies to resolve resource conflicts, or classify them as not critical. For example, to prevent three different controllers to conflict in accessing doors (Figure 10), they had to introduce a dedicated action to the door resource for each kind of controller in the taxonomy. This action would essentially mimic our conflict resolution strategy, taking a state as a parameter and determining whether to grant access to the door. In contrast to our approach, this ad hoc technique requires to structure the taxonomy with respect to conflict-handling concerns and to pollute the application code with conflict-handling logic.

Another ad hoc solution was implemented for the `Screen` device. It was used to show the schedule for students, and general news. To cope with the problem that the displayed information switched too fast (*e.g.*, the news application sent a new message short after the scheduler had done so), a timer was implemented at the `Screen` device that ignored messages for thirty seconds after receiving one. Obviously this could lead to problems, *e.g.*, if the emergency application is enhanced to show warning messages or assembly points during an emergency, and such a message is ignored or delayed due to the timer.

Systematic conflict detection

While Conflicts on the speaker system and the cameras were classified non-critical and therefore not treated, the conflict on the alarms was missed. Even though these three conflicts did not pose a problem to the simulated scenarios, neglecting them could lead to problems in the future. For example, if the system evolves and the speaker system is used for alarms, the conflict would become critical. Indeed, every newly installed application could use a resource in a critical way. Our approach systematically analyses the specification for conflicts after each evolution and detects such problems.

Incremental system development

With our approach, adding a new application to an existing system requires to declare and implement an additional system state component, if a new state is needed[1]. In this case, the new system state component is independent from other components, besides the new priority level to be introduced. Without our approach, it would be necessary to check and adapt local code at each device the new application is using.

Ease of use

The conflict-handling process is completely integrated into the development lifecycle of the home automation system. The required tasks are distributed to the different roles. This has the advantage that nothing is added later on, but is integrated by design. By leveraging the existing context components for the newly introduced states, we benefit of the possibility to easily interact with any other specified component in the system. Thus, the state components, which have to be implemented, can access (and therefore reuse) any existing context component or retrieve data directly from deployed devices. The fact that they are dedicated to conflict handling allows to completely generate conflict-handling components that interpret the states and enforce the conflict resolution. Focusing on a single problem enables us to provide the developer with a maximum of support. A similar approach for error handling has been implemented by Mercadal et al. [50].

1 Defining one system state per application is sufficient, because it gives an application a certain priority on all problematic resources it uses.

Related work

Conflicts are a major problem in a variety of domains. For example in telecommunications, Keck and Kühn show that feature interaction is an exponential problem that appears when new services are added to an existing system [43]. This problem can be directly mapped to pervasive computing, their services and features are our applications and their actions on resources [8]. Calder and Miller [9] use the Spin model checker to analyze telecommunication systems. To do so, a system (services and features) is modeled in Promela using temporal properties. Our approach circumvents the feature interaction problem by relying on existing system specifications and conflict-handling declarations provided by the domain expert and the application developer.

There exist different strategies to resolve conflicts in pervasive computing environments. Haya *et al.* assign a priority to every operation [33]. The priority is calculated by a central component using information about the current state, the caller and the type of operation. In comparison, our approach incurs little overhead for resource invocations because the *enforcer* component is coupled with the resource, preventing any central component from becoming a bottleneck.

The work closest to ours is that of Retkowitz and Kulle [66]. They use the notion of dependency management for handling resource conflicts. It is exemplified in the context of smart homes where it allows fine-grained configuration of a conflict-aware middleware. It is designed so a user can interact with the system and set priorities for different applications. In comparison, our approach is not limited to the home automation domain and covers conflict management throughout the development lifecycle: from design to programming, to runtime.

Tuttlies *et al.* have a different approach to resolving conflicts. They propose to describe the side effects of an application on the physical environment [87]. Additionally, each application states, what it considers a conflict. As a result, they can detect conflicts between interfering applications. Devising and applying a suitable strategy is left to the application developer. In contrast, we aim for a system-wide conflict management to allow system-wide reasoning.

Part III

PRIVACY

7

INTEGRATION OF SECURITY MECHANISMS

> *An aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches).*
>
> — Wikipedia [91]

In the previous chapter, we used an access control mechanism to enforce conflict management. Now, our goal is to improve privacy in the home automation system. Privacy is a key challenge in the development of pervasive computing systems (Section 2.5). Even if most middlewares provide support for the implementation of security policies (*e.g.*, encryption and authentication) developers must properly design systems to guarantee a correct implementation of these policies. The security mechanisms that are necessary for privacy typically impact every aspect of a system. As a result, privacy concerns must be dealt with at an early stage of the application design. Section 2.5 has shown that most approaches to developing pervasive computing systems ignore the design phase, thus hindering an early integration.

Aspect-oriented programming (AOP) [44] is a software engineering approach to combining non-functional concerns with software design. AOP provides techniques, languages, and tools to systematically represent, modularize and compose concerns that are crosscutting an entire system.

This chapter presents our approach to defining security concerns alongside the description of a pervasive computing system. These concerns are automatically mapped into the system implementation, using an aspect-oriented approach.

7.1 BACKGROUND

Let us now introduce a few notions that are relevant to the rest of this chapter.

We start with a few concepts of the DiaSpec language that was presented in Chapter 4. These concepts provide a basis

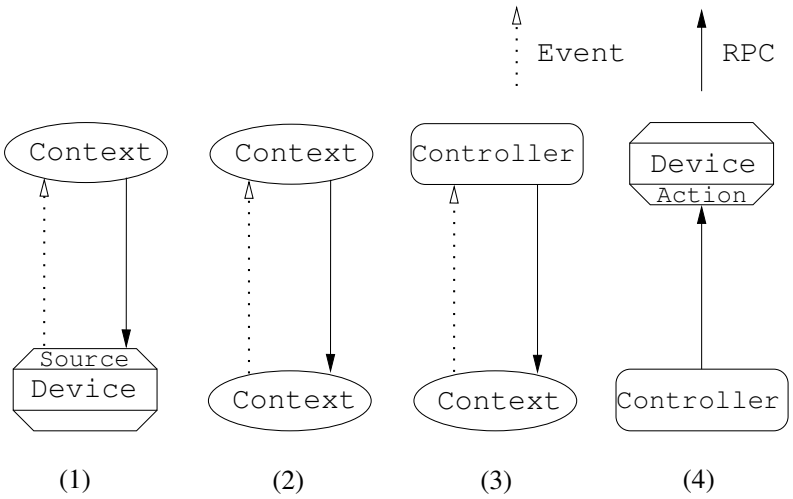

Figure 14: Possible interactions between DiaSpec components

to identify elements that are crosscutting the system design, and thus required to express aspect-oriented programming for DiaSpec. Afterwards, we introduce requirements for aspect-oriented software development (AOSD) at the architecture level. Later in this chapter we use this information to define an aspect-oriented language dedicated to DiaSpec.

Interactions between components

Figure 14 shows the possible interactions between DiaSpec components[1]. The communication is realized by two well-known principles: a synchronous remote procedure call (RPC), allowing a one-to-one interaction, and the publish/subscribe paradigm, in which a publisher sends events to receivers registered for the corresponding event type[2].

The restrictions of allowed interactions between DiaSpec components (Figure 14) are a result of the sense-compute-control (SCC) paradigm that is enforced by DiaSpec (Section 4.1). The allowed interactions are: (1) The source of a device emits events that are received by context components, or context components can retrieve the source value via RPC[3]. (2) & (3) A context emits an event to other context or controller components, or context and controller components can access the context value directly via RPC. (4) Controller components perform actions on device

1 A component is either a device, a context, or a controller.
2 In architecture description languages, the type of interactions between components is defined by a *connector* [36].
3 Actively requesting a value via RPC is later on called *query*.

components via RPC. Note that actions are always of type `void` to conform to the SCC paradigm[4].

The framework generated by DiaGen supports both interaction modes when a connection is specified in the design (Figure 14, cases 1 to 3). The choice of implementation is left to the developer.

Runtime services

Components that enter a pervasive computing system register to DiaEnv and provide information about their capabilities and attributes. This information is used by DiaEnv to provide developers with a service discovery mechanism that filters device and context component instances according to their type and their attribute values. A dedicated implementation of this mechanism is generated in conformance with the DiaSpec description. It corresponds to a typed version of existing service discovery mechanisms.

Security concerns typically alter the way in which components interact, that is, their behaviour. As well, they impact runtime services such as service discovery.

Aspect-oriented programming (AOP)

The decomposition of software into small, meaningful, manageable and comprehensible parts has been a core idiom of software engineering for decades. A proper separation of concerns promotes reusability, traceability, adaptation, and comprehensibility. But, the relevance of concerns may vary with respect to roles: architects, developers and administrators. It may also vary depending on the stage of the software life cycle. Moreover, concerns may be constrained by the implementation language being used. For example, the object-oriented paradigm drives the decomposition of data structures into classes.

Despite the numerous software engineering approaches to system decomposition (*e.g.*, libraries, modules, components, *etc.*), achieving a proper decomposition where every concern is correctly modularized is not possible in practice [84]. Some concerns are then spread out and mixed; these are said to be crosscutting the decomposition of the system.

AOSD is a software engineering approach that focuses on the identification and representation of crosscutting concerns, and their modularization in separate units, as well as their automated composition into a complete system.

4 Controllers are not allowed to retrieve information directly from devices.

AOP [44] is a language independent paradigm where aspects encapsulate crosscutting concerns. In an aspect-oriented language, an aspect associates an advice, the actual code of the concern, with pointcuts that refer to the regions in the base program where the advice is to be applied. An aspect weaver then realizes the coordination of the aspects with the base program, either statically, *e.g.*, through static code inlining, or dynamically, *e.g.*, using the host language's reflection mechanism.

While a significant body of work has focused on the proper decomposition of system architectures [2], it does not explicitly focus on crosscutting concerns. Properties such as security and QoS are inherently crosscutting. Moreover, these properties must be explicitly specified at design time to reason about them.

7.2 IMPROVING PRIVACY

This section introduces two examples of security concerns that crosscut a system design. We revisit these examples in Section 7.3.3 to illustrate the expressiveness and effectiveness of our approach.

7.2.1 *Managing certificates for SSL communication*

Authentication, confidentiality and integrity are key objectives for ensuring privacy. Their effective implementation depends on numerous factors, such as the network type, the level of trust between users, *etc.*

Let us consider the example of the RPC. It is a well-known mechanism to perform remote computations and to exchange messages in distributed systems. RMI is the Java application programming interface that performs the object-oriented equivalent of an RPC. While RMI offers a simple programming interface, it provides no security guarantee: RMI is built on top of the Java remote method protocol, which exchanges serialized Java objects in clear.

In this situation, good practices to achieve a secure design require the use of a secure channel [82]. The secure socket layer (SSL) is a wide spread protocol used to secure transmission in distributed systems. SSL performs authentication and encryption. Operationally, SSL requires participants to store certificates of trusted entities. Fortunately, the RMI API provides support for RMI over SSL. In practice, developers pass a reference to a *TrustStore* containing certificates of trusted entities to the RMI API that transparently performs authentication and encryption.

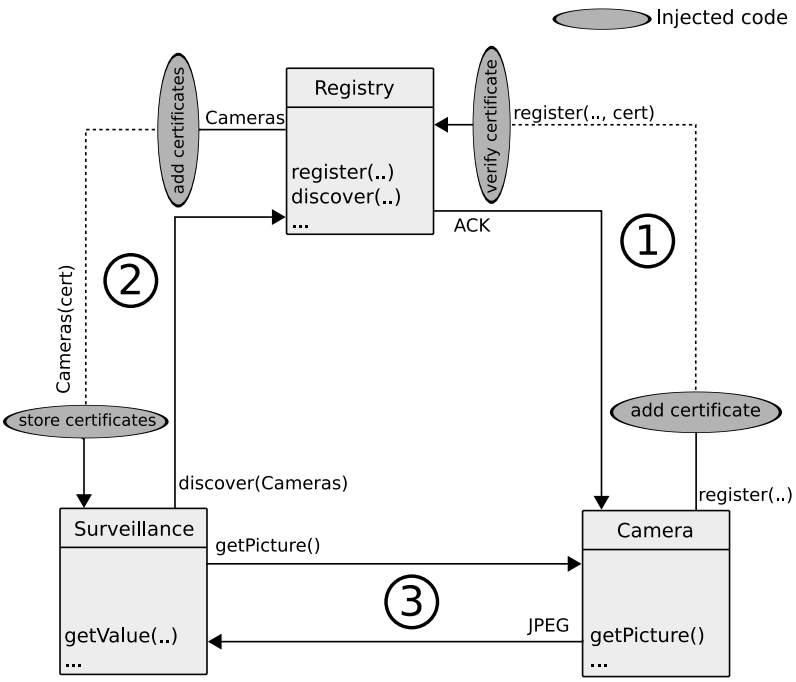

Figure 15: Distribution and verification of certificates

This approach is suitable for the client/server model, where every client has to hold the server certificate and the server to eventually have all client certificates. However, its application to multi agent systems, where entities *(dis)*appear and *(un)*register dynamically, requires additional code to manage certificates.

Figure 15 shows two components of the intrusion module (Chapter 4), the context component `Surveillance` and the device `Camera`. The third component is `Registry` that is part of DiaEnv. In the example, `Surveillance` retrieves an image that is provided by `Camera`. Several steps are necessary beforehand: (1) on entering the pervasive computing environment, devices have to register and pass their certificates. `Registry` must then verify the certificate before proceeding with the registration. (2) Whenever `Surveillance` wants to retrieve an image, it must first obtain instances of `Camera` from the service discovery service that is provided by `Registry`. `Registry` then returns proxies on instances of `Camera` with their certificates. (3) `Surveillance` may now communicate with any of these `Camera` components via SSL.

As illustrated in Figure 15, the management of certificates for SSL communication typically crosscuts multiple aspects of a distributed system, *e.g.*, registration and discovery, in both the component code and the built-in services (DiaEnv).

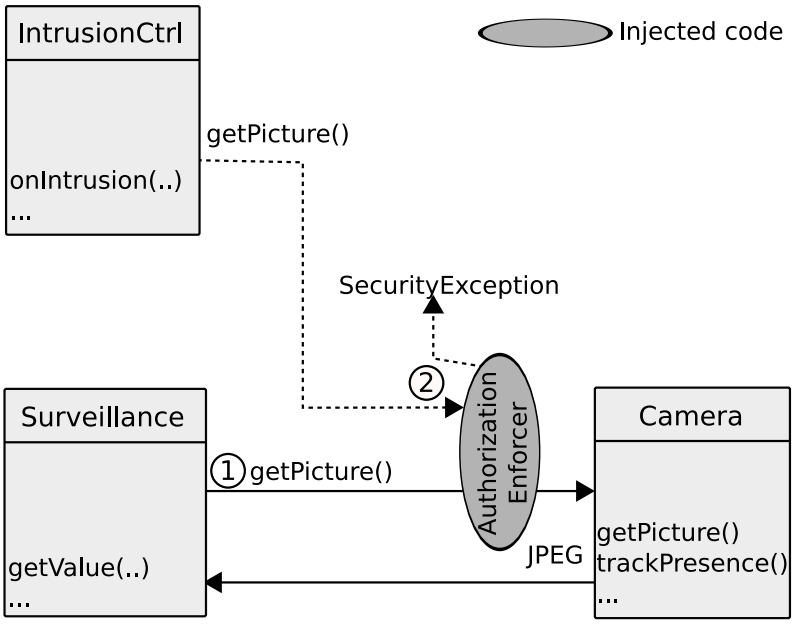

Figure 16: Enforcement of Access Control Lists

7.2.2 *Enforcing access control lists*

The use of SSL connections combined with signed certificates allows the middleware to enforce authentication and encryption of communications in a distributed system. Nevertheless, the security provided by SSL communication is somewhat coarse grained. In a pervasive computing system, every entity must verify each access at a fine-grained level. The fact that an entity is authenticated does not mean that it has access to all resources and information.

For example in Figure 16, `Surveillance` may retrieve images of `Camera` by using the `getPicture` method (1). This interaction has been specified in the design specification[5], and thus is allowed by the authorization enforcer. But, the use of the `getPicture` method should be denied for the `IntrusionCtrl` component (2). This controller component is only allowed to access the `Track` interface of `Camera`. Even if DiaGen generates a programming framework for `IntrusionCtrl` that only exposes a limited view of `Camera` (where only specified interfaces are accessible), a malicious developer might escape the programming framework and craft a request to access every interface exposed by the `Camera` component.

To forbid such uncontrolled access, the system must enforce access control lists upon requests on provided operations. As shown in Figure 16, the access list enforcement logic should be externalized from components [82]. That is, on reception of re-

5 `Surveillance` uses the source `picture` of the `Camera` device (Listing 2, line 11).

quests, the receiver must query the access controlling mechanism to verify that the caller is allowed to request a particular information or action from the callee. Indeed, in Section 7.3.3 we are extending the access control from the previous chapter to include all context components as well. This is done to restrict access to information, and thus to treat information leaks that are a major threat for privacy.

Again, the implementation of the access control enforcement impacts multiple regions of the design; that is, every interface entry point of device and context components.

7.3 THE DIAASPECT LANGUAGE

The previous section presented two examples of security concerns that crosscut the design of a system. To apply AOP techniques to such a system design, we developed DiaAspect, an aspect-oriented language for DiaSpec. This section first describes the join point model. A join point refers to a region in the DiaSpec design description and/or DiaSpec generated programming framework, where aspect code is injected.

Afterwards we present our pointcut language. Pointcuts represent the occurrence of one or more join points. A pointcut is a predicate; it may or may not match the current join point. In addition, a pointcut may expose information specific to the underlying join points at runtime.

Finally, we introduce the DiaAspect language.

7.3.1 *The join point model*

The DiaAspect join point model defines the set of events of interest in a system design description and its associated generated programming framework. AOP events are represented as messages in DiaEnv. For the sake of conciseness, only messages of interest between components of a DiaSpec architecture specification are listed here. Each listed message corresponds to two distinct join points in our model: one at the message emitter and one at the receiver. Figure 17 illustrate our model.

COMPONENT REGISTRATION The `register`, *respectively* `unregister`, message occurs when a component instance notifies a registry of the arrival of a component instance, *respectively* departure, in/from the system. Both the `register` and `unregister` messages have one argument, the signature of the component arriving/leaving. Our model distinguishes the component issuing

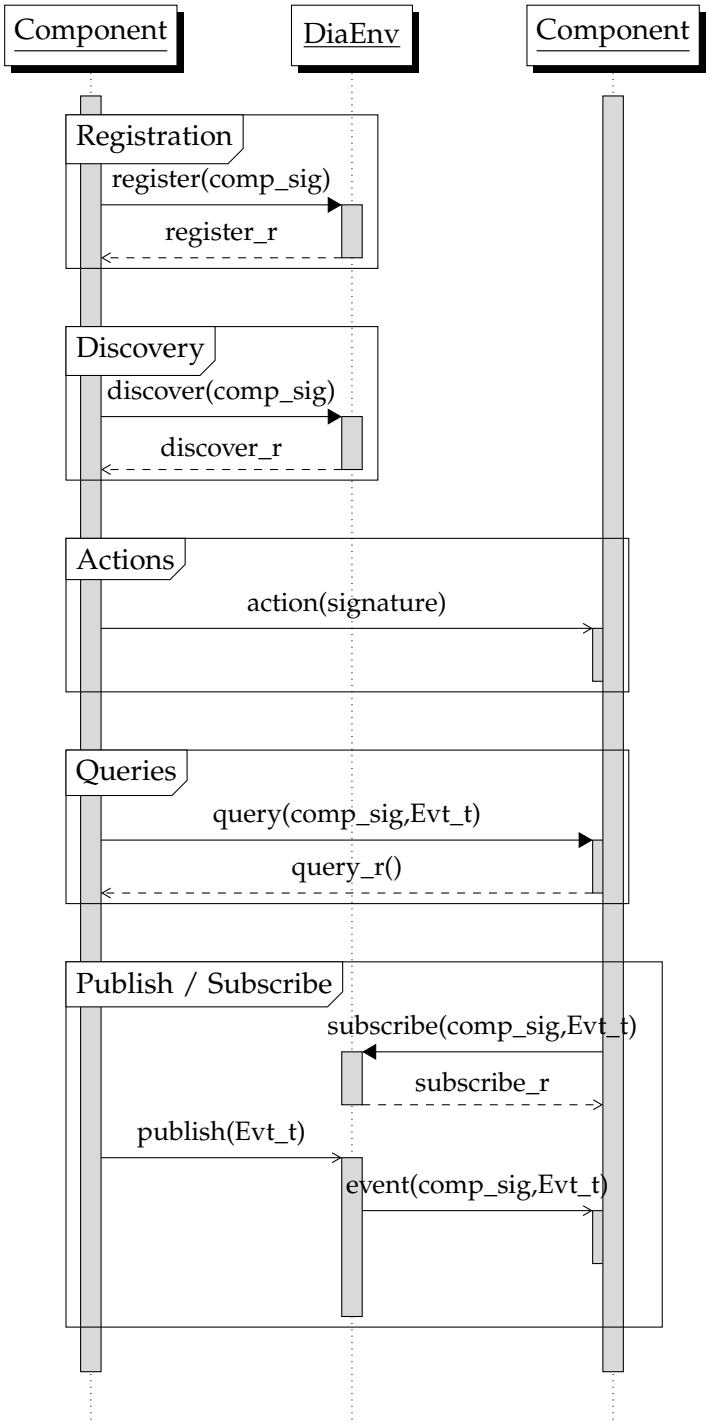

Figure 17: DiaAspect Join Point Model

the (un)register message from the actual component that is arriving/leaving. The `register_r`, *respectively* `unregister_r`, message correspond to the acknowledgment from the DiaSpec registry to a component, after a `register`, *respectively* `unregister`, message. These messages have no argument.

COMPONENT DISCOVERY The `discover` message occurs when a component instance queries a registry for a component. A discovery is parameterized with the partial signature of the component to search for, that is, the component type and the attributes on which to apply a filter. On reply, the DiaSpec registry emits a `discover_r` message containing an array of components.

ACTIONS The `action` message matches the call or the reception of a method defined in an action. The corresponding join point takes the following arguments: the method's name and the argument's types.

QUERIES The `query` message matches the call or the reception of a value request, either on a device or a context component. The corresponding join point takes the following arguments: the component's signature that provides the value, and the type of the value. The providing component answers with a `query_r` message that contains the requested value.

EVENT SUBSCRIPTION The `subscribe` message matches the subscription of a component instance to an event queue. Subscriptions have two arguments, the signature of the publisher and the type of events the subscribers are interested in. Note that the publisher (the parameter) is not the same component as the receiver of the subscription message. Indeed, in DiaSpec, a subscriber subscribes to DiaEnv, not directly to the publisher. The response corresponds to the `subscribe_r` message that has no argument.

EVENTS PUBLISHING AND RECEPTION The `publish` message occurs when a component publishes an event. It is parameterized with respect to the event type and received by DiaEnv. Consequently, DiaEnv emits an `event` message to each subscriber. That message has two parameters, the signature of the publisher and the event type.

```
diaaspect::= (aspect_def | pointcut_def)*;

aspect_def::=
        'aspect' ID ('before'|'after'|'around') '(' signature ')'
        ('returns' type)? throw? ':' pointcut_ref advice;

pointcut_ref::= ID | pointcut_def;

advice::= '{' /* Java code + thisJoinPoint + proceed */ '}'

pointcut_def::= 'pointcut' 'ID '(' signature ')' ':' pointcut ';';

pointcut::= ('send' | 'recv')?
        (
                register
                | unregister
                | discover
                | action
                | query
                | subscribe
                | publish
                | event
        ) (&& from(component_signature)? (&&
            to(component_signature)?
        (&& if (expression))?
        ;

register::= 'register' '(' component_signature ')';

unregister::= 'unregister' '(' component_signature ')';

discover::= 'discover' '(' component_signature ')';

action::= 'action' '(' action_signature ')';

query::= 'query' '(' component_signature ',' event_type ')'

subscribe::= 'subscribe' '(' component_signature ',' event_type
    ')'

publish::= 'publish' '(' event_type ')';

event::= 'event' '(' component_signature ',' event_type ')';

component_signature::= pattern '(' (arg_sig (',' arg_sig)*)? ')';

arg_sig::= (pattern pattern) | '..'; /* AspectJ like */

action_signature::= pattern '(' (arg_sig (',' arg_sig)*)? ')';

event_type::= pattern;

pattern::= /* AspectJ like string pattern */
```

Listing 9: Partial DiaAspect BNF

7.3.2 *The aspect language*

Figure 9 presents the partial BNF of the DiaAspect language. We chose a syntax similar to AspectJ, an aspect-oriented system for Java [45]. The benefits of reusing an existing syntax are well-known from both a user and an implementer perspective. The latter perspective is illustrated in Section 7.4 where AspectJ is shown to greatly simplify the implementation of our language.

Our language associates pointcuts with advice written in Java. An advice holds the implementation of crosscutting concerns and pointcuts select join points where an advice is to be executed. We first present our pointcut language. Then, we introduce the runtime API supporting advice implementation with the information relative to join points matched at runtime.

7.3.2.1 *The pointcut language*

In the aspect-oriented paradigm, pointcuts act as join point selectors. In addition, to capture the occurrence of one or more join points, pointcuts may expose runtime information specific to these join points. The DiaAspect language proposes eight kinds of pointcuts for which we distinguish two categories:

- Design specific pointcuts. These directly relate to artifacts defined by the architect in the system specification. That is, component relationships (connectors).
- Built-in pointcuts. These correspond to built-in services provided by the DiaSpec runtime. That is, component registration, *etc*.

The following lists the pointcut featured in DiaAspect. We previously stated that a join point may catch the emission and/or reception of messages between DiaSpec components. Accordingly, pointcuts in DiaAspect distinguish join points on the emitter and/or the receiver side. If a pointcut is preceded by the `send`, *respectively* `recv`, keyword, it matches only the join point on the emitter, *respectively* receiver, side. If no keyword precedes, join points on both sides are matched.

REGISTER The `register` pointcut (Listing 9, line 27) matches the occurrence of `register` join points and the corresponding `register_r` join points, where the registering component matches the signature given as a parameter. For example, the pointcut `recv register (Device (..))` matches every `register` join point received, where the component instance extends the `Device` component.

UNREGISTER The `unregister` pointcut (Listing 9, line 29) captures the occurrence of `unregister` join points and the corresponding `unregister_r` join points, where the unregistering component matches the signature given as a parameter. For example, the pointcut `send unregister (* (..))` matches every `unregister` join point.

DISCOVER The `discover` pointcut (Listing 9, line 31) matches the `discover` and corresponding `discover_r` join points given a partial component signature (that is, the type of the component requested and the attributes on which the results are filtered). For example, the pointcut `recv discover (Camera (Location loc == "Room 203", ..))` matches the reception of every discovery operation for components of type `Camera` with a specific location attribute.

ACTION The `action` pointcut (Listing 9, line 33) captures all `action` join points for a given method signature. For example, the pointcut `action (OnOff.*(..))` catches every call of methods defined in the `OnOff` action.

QUERY The `query` pointcut (Listing 9, line 35) catches the occurrence of `query` and their respective `query_r` join points, where the providing component signature and the requested value type match. For example, the pointcut `query (Camera (..), JPEG)` matches every image request for cameras.

SUBSCRIBE The `subscribe` pointcut (Listing 9, line 37) intercepts all `subscribe` calls and their respective `subscribe_r` joinpoints for a given component signature and event type. For example, the pointcut `recv subscribe (* (..), Message)` matches every subscription to an event of type `Message`, regardless of the publisher.

PUBLISH The `publish` pointcut (Listing 9 line 39) matches the occurrence of `publish` join points for a given event type. For example, the pointcut `publish (Fire)` matches every published `Fire` event.

EVENT The `event` pointcut (Listing 9 line 41) catches the occurrence of `event` join points for a given event type and a publisher signature. For example, the pointcut `send event (Context (..), *)` matches the sending of any kind of `event` following the publication by a publisher extending `Context`.

Pointcuts select join points that correspond to messages exchanged between component instances. Pointcuts filter join points on their types and the values of their arguments. In our language, an aspect may further filter the messages of interest by specifying additional clauses alongside pointcuts.

FROM/TO As stated before, join points selected by pointcuts relate to messages exchanged between DiaSpec components. Hence, pointcut arguments correspond to the specific content of these messages. In addition to filtering join points according to message type and content, one can further restrict the collected join points by using the `from` and `to` clauses. Given a component signature, the `from`, *respectively* `to`, clause (Listing 9 line 23) restricts collected join points to those emitted, *respectively* received, by components matching the signature in that clause. In `pointcut && from(Device(Location *, ..))`, join point matching is limited to those emitted by components extending Device and with at least one attribute of type Location.

IF To allow aspect developers to further specify the region where to trigger advice, DiaAspect features an `if` clause similar to the one of AspectJ. The advice only executes if the given expression holds true. That expression must be a Java boolean expression that may refer to the DiaAspect runtime API and variables bound in the pointcut definition.

7.3.2.2 *The runtime API*

An advice in DiaAspect is developed in Java. To support developers in writing advice, DiaAspect features a runtime API that exposes similar features to those in AspectJ.

The `proceed` method is similar to the one in AspectJ. It is a virtual method that has the same signature as the pointcut matched, and is used in `around` aspects. Its execution evaluates the join point matched by the aspect. For example, inside an aspect on the sending of a `register` message, the execution of the `proceed` method in the advice resumes the execution of `register`.

As in AspectJ, DiaAspect also provides an advice-visible variable `thisJoinPoint`. It exposes reflective information about the join point that triggered the advice. We extended `thisJoinPoint` to expose the architecture specific information about the current join point in addition to Java semantic information about it. For example, in the case of an aspect on an action call, `thisJoinPoint.getCallerSignature()` returns the signature of the caller component.

Because aspects may be developed independently of the DiaSpec implementation code, developers can not benefit from the support of a generated programming framework when writing advice. Still, to allow developers to publish events and execute actions, we expose the `Processor` API that acts as a front-end to the generated framework. Hence, developers may write an advice

```
1  /* Adding certificates to a discovery request. */
2  aspect around recv discover (* (..)) && to (registry) {
3    RemoteServiceInfo[] rsis = proceed();
4    for (RemoteServiceInfo rsi: rsis) {
5      rsi.setCert(registry.getCertHelper().
           getCertificate(rsi.getID()));
6    }
7    returns rsis;
8  }
9
10 /* Storing the certificates of discovered services locally. */
11 aspect around send discover (* (..) && from (service) {
12   Proxy[] proxies = (Proxy[]) proceed();
13     for (Proxy proxy: proxies) {
14       service.getCertHelper().storeCertificate(
             proxy.getRemoteServiceInfo().getCert());
15     }
16   return proxies;
17 }
```

Listing 10: DiaAspect code managing certificates in a RMI with SSL distributed system (excerpt)

that interacts with DiaSpec components, while still benefiting from runtime coherency checks provided by the framework.

7.3.3 *Crosscutting concerns implemented with DiaAspect*

This section revisits two concerns discussed in Section 7.2 to illustrate DiaAspect and to show that it allows such concerns to be concisely modularized at the design level. We first return to the management of certificates to implement SSL communications. Afterwards, we revisit the enforcement of access control lists.

Managing Certificates for SSL Communications

Listing 10 presents the DiaAspect code that implements the distribution of certificates for SSL encrypted communications. The first aspect augments the behavior of the DiaEnv registry on service discovery to pass the certificates of the discovered services to the requesting component. The second aspect intercepts discoveries to store those certificates locally. These certificates can now be used by the SSL socket factory to encrypt communication. Both aspects use the `CertificateHelper` class to store and retrieve certificates[6].

6 An excerpt of the Java implementation can be found in Appendix B

```
1  /* Checking if an action is authorized. */
2  aspect around recv action (* (..)) {
3    if (getEnforcer().authorize(thisJoinpoint)) {
4      return proceed();
5    }
6    throw new DiaGenSecurityException(thisJoinpoint);
7  }
8
9  /* Checking if a query is authorized. */
10 aspect around recv query (* (..), *) {
11   if (getEnforcer().authorize(thisJoinpoint)) {
12     return proceed();
13   }
14   throw new DiaGenSecurityException(thisJoinpoint);
15 }
```

Listing 11: DiaAspect code enforcing access control lists (excerpt)

Enforcing Access Control Lists

Listing 11 contains the DiaAspect code to enforce access control lists on actions and queries declared in a DiaSpec architecture[7]. It contains a single aspect that intercepts any receiving DiaSpec method call (line 2). We assume that a local object `Enforcer` performs the actual enforcement of an access rule. The advice calls the `authorize` method on the local `enforcer` component (line 3). Depending on the result, the advice either throws a `DiaGenSecurityException`, notifying the caller of the failure, or proceeds with the original method call, as is done for the conflict management. Indeed, the only difference is that we applied conflict management only on actions on devices. The given example in Listing 11 also protects the sources of devices and context components from forbidden accesses (lines 10 to 15). Note that context components are not affected by conflicts. Thus, the ACL of context components does not require to be updated at runtime.

7.4 IMPLEMENTATION

Weaving is the process of coordinating the aspect code with non-aspect code, *i.e.* the base program. In our approach, the aspect code refers to the design description. However, instead of weaving the design description, we inject aspects in the programming framework generated by DiaGen, and DiaEnv. Specifically, DiaAspect aspects are translated into AspectJ aspects that are woven into the implementation code. This approach exploits the design

7 Queries are implicitly declared, when a component is connected to a source or context.

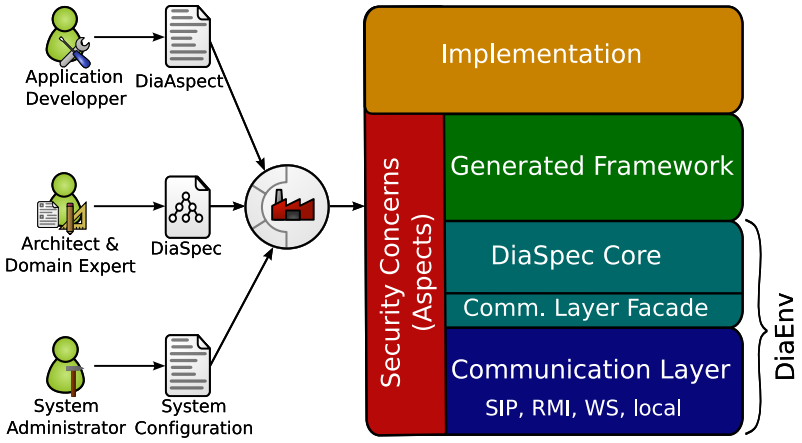

Figure 18: Security concerns crosscut the DiaSpec/DiaGen generated code structure.

description to adapt the weaving process to a given generated implementation support.

Section 4.4 already depicted the structure and organization of the generated Java framework. Figure 18 shows how security concerns crosscut the generated framework and DiaEnv. Given this structure, a received message first passes through the specific communication layer that formats it in a unified form before passing it to the core layer. The core layer unmarshals the message content to extract the sender information, the message type, (*e.g.*, action, query, event, *etc.*), and its content. The extracted information is passed to the generated framework, which in turn dispatches it to the appropriate component proxy and finally to the component implementation. Similarly, method calls and event publishing follow the inverse path. One can note that this particular implementation offers multiple regions to intercept similar join points, as is shown by the `security concerns` box that traverses the stack model vertically (Figure 18). The next section shows how to benefit from this strategy to optimize aspect weaving.

DiaAspect Aspects Weaving

The DiaAspect language describes aspects that coordinate join point regions. These join points refer to artifacts, *i.e.* components, connectors and built-in services, which are defined in a DiaSpec design description and its runtime environment (Section 7.3). An advice is defined purely in Java (exception made of the proceed keyword). The compilation and weaving process must connect the design pointcuts with the generated framework and the implementation code. To do so, DiaAspect aspects are translated

into AspectJ code that is woven into the generated code and DiaEnv.

Translating DiaAspect aspects into AspectJ code amounts to projecting the DiaAspect join point model into the programming framework generated by DiaGen. DiaAspect pointcuts are translated into AspectJ method call pointcuts on the methods of the Java interfaces generated by DiaGen. For example, the DiaAspect pointcut `action (Track.* (..))` intercepts any method declared in the action interface `Track` is translated into the following AspectJ pointcut, `call(* spec.package.name.interfaces.Track.* (..))`.

The translation of the DiaAspect `from` and `to` clauses depends on the pointcut type, *i.e.* `send` or `recv` pointcuts. In the case of `send` type pointcuts, for example in `send action(* (..)) && from(Controller()) && to(Device)`, the `from` clause is translated to limit AspectJ pointcut matching, to callers extending the abstract Java class `Controller`. This is done by using the following AspectJ code

```
this(spec.package.name.components.Controller)
```

and the following invocation

```
this(spec.package.name.proxies.DeviceProxy)
```

It results in restricting the object on which a method is called to `DeviceProxy` classes.

In the case of a `recv`, the pointcut is limited to a caller of type proxy. In addition, the object on which the method is called must extend the corresponding abstract class.

Listing 12 presents the AspectJ code generated for the DiaAspect code from the example on enforcing access control lists. The DiaAspect aspects in Listing 11 are translated into AspectJ aspects. The first aspect is woven around the execution of the `orderReceived` method that is called whenever a DiaSpec action call is received by a component instance providing that method. The pointcut limits the matching to classes, extending the `Device` class. The second aspect intercepts method calls of the `queryReceived` method on all components that extend the `Service` class (the base class for every DiaSpec component). The advice code is left unchanged.

The generated framework is composed of multiple layers (Figure 18). Depending on the aspect code, it is possible to modify the aspect projection (weaving) to shortcut these layers. This is done to optimize performance of the applications. For example, consider a DiaAspect aspect intercepting all receiving calls for a given method. Instead of weaving the aspect in the generated framework layer, we can inject it directly into the communication

```
1  /* Checking all actions that are executed on devices. */
2  Object around(RemoteServiceInfo rsi, Service callee, String
       method) throws DiaGenException:call(Object orderReceived(
       RemoteServiceInfo, String, Object...) throws DiaGenException)
       && target(Device+) && target(callee) {
3    if (callee.getEnforcer().authorize(thisJoinpoint)) {
4      return proceed(rsi, callee, method);
5    }
6    throw new DiaGenSecurityException(thisJoinpoint);
7  }
8
9  /* Checking all queries that are sent to components. */
10 Object around(RemoteServiceInfo rsi, Service callee, String
       method) throws DiaGenException:call(Object queryReceived(
       RemoteServiceInfo, String, Object...) throws DiaGenException)
       && target(Service+) && target(callee) {
11   if (callee.getEnforcer().authorize(thisJoinpoint)) {
12     return proceed(rsi, callee, method);
13   }
14   throw new DiaGenSecurityException(thisJoinpoint);
15 }
```

Listing 12: AspectJ code generated for the DiaAspect aspect from the example on enforcing access control lists (excerpt)

layer (*i.e.* RMI, Web Service, *etc.*). When the advice code does not call the `proceed` method, this optimization avoids unmarshalling the call up to the application layer. Similarly, weaving can be specialized depending on the specific communication layer and of the aspect code.

7.5 SUMMARY

This chapter presented our approach to expressing security properties in system design using an aspect-oriented approach. The specification of these properties, *e.g.*, secure communication, impact every aspect of a pervasive computing system and cannot be properly expressed in the traditional component-and-connector idiom.

To overcome this limitation, we proposed DiaAspect, an aspect-oriented language dedicated to DiaSpec and its runtime support. This allows to define regions, where to apply crosscutting security code, on design level. The design specification of the system is used to inject code into the generated framework, and DiaEnv.

8

EVALUATION

Compared to our approach to conflict handling where it is sufficient to implement a configuration of the already existing mechanism in the form of state components, DiaAspect targets to ease the implementation of these mechanisms. DiaAspect was not integrated into the DiaSuite development process, it is rather used to specify where security aspects are to be applied in a given system design. Obviously, this is done after designing a system, since the design description is needed to describe the pointcuts. The security code that is used within the aspects has to be implemented separately, *e.g.*, the management of certificates is done by the `CertificateHelper` class (Appendix B, Listing 15).

Expressiveness

DiaAspect is expressive concerning the interception points in the system and the type of exchanged message, such as actions or events, for example. In a pervasive computing system, the deployed components and the messages they exchange are the main points of interest for security concerns or non-functional properties in general. In that, DiaAspect provides valuable aid to the developer, since the generated framework is complex. This is due to several abstraction layers that enable the framework to enforce the design description of an application, and to target different communication technologies. Additionally, since the framework is generated, modifications to the framework are lost when it is regenerated. As a result, the developer has to modify the framework generator to add code permanently in certain regions. This is complicated and error prone, and impacts every generated framework.

Lessons learned

The work on DiaAspect had an impact to the DiaSuite project in general. Before, the focus lay on functional aspects of application development. At that point, the DiaSpec language and the generated framework were still evolving. Analyzing the framework concerning the integration of security concerns led to restructures within the framework to ease this task. Indeed,

non-functional properties are entering the spotlight: temporal quality of service (QoS) constraints [29] and error handling [50].

Ease of use

Developers that are familiar with AspectJ will easily adopt the syntax of DiaAspect. The declarations of packages, classes, and methods is no longer necessary, making the pointcut declarations easier to define and more readable. Nevertheless, DiaAspect also inherits the disadvantages of AOP. Projects depend on another technology (*i.e.*, AspectJ) that introduces new potential sources of errors. Additionally, debugging becomes more difficult. For once, because the aspect code is developed separately from the application code. Another reason is that the debugging mechanism of the integrated development environment (IDE[1]) provides less support when multiple technologies are combined (*i.e.*, Java and AspectJ).

Related work

This section briefly reviews approaches related to the modelling of crosscutting concerns in architecture design and in pervasive computing.

Multi-dimensional separation of concerns [84] shows how the software artifacts, corresponding to different concerns (*a.k.a.* hyperslices), can be merged to generate a full application. This is the approach chosen by subject-oriented programming [32], where hyperslices are pieces of code (*e.g.*, partial class hierarchies). Aspect-oriented programming [44] is quite similar but it is asymmetric: it considers the structure of a base program and it provides pointcut languages to specify where another code crosscuts the base program and the corresponding pieces should be woven.

DAOP-ADL [61] is an XML-based architecture description language that integrates aspects as first class entities of architecture description. The interconnection of aspects with components in DAOP-ADL is specified as evaluation rules on the interfaces of the architecture. Similarly, Pessemier et al. [59] integrate aspect entities in the Fractal ADL, by extending the component membrane to support aspect weaving. PRISMA [58] is an ADL that integrates an aspect-oriented approach directly in the component-and-connector approach: aspects are modeled as components, allowing direct reuse of consistency checks and code generation

1 For example, Eclipse http://www.eclipse.org

tools. In comparison to these works, our aspect language DiaAspect is not limited to the artifacts of the ADL but also allows aspects to coordinate with the built-in services, like component registration and discovery provided by the generated framework.

Ren *et al.* [64, 65] present an extension of xADL, an XML ADL, for modeling security at the architecture level. Their extension permits architects to control component instantiation, interface access, and data flow by annotating the components and connectors. In comparison to this work, our approach allows to inject the necessary code to enforce such policies without modifying the system architecture. Moreover, by specifying these changes at the design level, our implementation is reusable with multiple middleware.

Part IV

RESULTS

9 CONCLUSION

In this thesis we have presented an approach to integrating security concerns into the development process of pervasive computing systems. The focus was access control and privacy, both problems of particular interest in this domain. We now summarize the thesis results and draw overall conclusions.

Pervasive computing environments, or smart homes in particular, promise new ways to supporting users in their daily activities. Developing these supporting applications is challenging, because pervasive computing combines elements of several domains, thus inheriting and creating many challenges for the application developer. Besides functional challenges that have to be addressed to make application development feasible, security concerns are of high importance. This is due to the fact that the more technology permeates the user's lives, the more he depends on it, and thus, failures of critical applications can endanger him and his assets. Therefore, they require adequate protection.

We have presented several approaches that are used to develop applications for pervasive computing systems and examined their limitations. Most approaches do not cover the whole development lifecycle, thus neglecting the necessity to integrate security concerns from the start, that is, the design phase. Another problem was that approaches focused either on security concerns or on providing functional support for the developer, though both are essential to develop secure systems.

To tackle these shortcomings, we have presented a domain-specific approach to architecturing conflict handling of pervasive computing resources. This approach covers the development lifecycle of a pervasive computing system. Our approach includes the automatic detection of potential conflicts, their resolution, and their prevention at runtime. We have extended our design language DiaSpec to add information that is required for these three stages of conflict management. This information is used to generate code that guides and supports the implementation of conflict management.

We have introduced new tasks dedicated to conflict management in the development process of a pervasive computing system. In the resulting process, application and conflict-handling code are cleanly separated. Furthermore, our approach to conflict

management is incremental and modular, preserving the independence between applications. This facilitates reuse of applications, and makes the conflict management easier to understand and verify.

To prevent conflicts, our implementation enforces an ACL for each pervasive computing resource. These ACLs are automatically updated based on the current system state.

To integrate security mechanisms and cope with the problem that security concerns typically impact every aspect of a pervasive computing system, we have proposed DiaAspect, an aspect-oriented language dedicated to DiaSpec and its runtime support. We developed DiaAspect on top of both the sense/compute/control idiom that is used by DiaSpec, and common runtime services provided (*e.g.*, service discovery). We have shown that DiaAspect is expressive enough to implement concrete solutions on two widespread security problems: the distribution of certificates over an encrypted network and the enforcement of access control lists in home automation systems.

We have presented the implementation of the DiaAspect aspect weaver. It injects aspect code into the programming framework generated for DiaSpec specifications. We implemented our weaving process by translating DiaAspect code into aspects written in AspectJ, a well-known aspect-oriented system for the Java programming language. We have also demonstrated how our translation scheme allows to modify the DiaAspect pointcuts projection to optimize the woven code according to a design specification and the structure of generated frameworks.

10

FUTURE WORK

Currently, our approach to conflict management treats conflicts for classes of resources. This strategy applies to situations where applications act on all instances of a class (*e.g.*, the emergency application unlocks all doors). We plan to extend this approach by introducing a location (*e.g.*, a room, a floor) in the conflict management declaration of an application. The interaction contracts that were recently added to the DiaSpec language [15] should be a good basis for this extension. Until now, DiaSpec focused on the functionality of applications making it difficult to extract useful information for non-functional properties. This is changing with the interaction contracts that provide great potential for specifying *how* components should interact. Gatti et al. [29] extended these contracts to specify time constraints for components, showing the possibilities they open up on integrating non-functional properties from the very beginning of a project. Other possibilities are specifying the expected ranges for values (*e.g.*, events or method parameters) or providing information to configure other integrated security mechanisms.

For now, our conflict management requires the system administrators to implement system state components in Java (Section 5.2), which is not feasible for smart homes[1]. To address this problem and make this task doable for the end-user (in this case the occupants of smart homes), we plan on leveraging Pantagruel, a rule based, graphical language that has been implemented on top of DiaSpec [22]. Pantagruel provides high level constructs to orchestrate DiaSpec entities in form of rules, exactly what is required for the state components.

We also plan to expand our model to include the access rights of users. Access control is a major problem in pervasive computing environments, since it must handle physical and virtual objects at the same time [71, 86]. A related research direction is to integrate user preferences into our model to resolve certain types of conflicts, as proposed by Shin et al. [78].

Another interesting research direction is to protect the privacy of users by enforcing a more precise access control to information in the system. Currently, all accesses to context and sources that are specified in the design are allowed, without any reasoning about the type of information,*e.g.*, location information about

1 As we mentioned in Section 2.2, the user is not an administrator.

people might be sensitive, while the temperature of rooms is not. Hengartner and Steenkiste [34, 35] propose to use relations between information to derive access rights or change the granularity of the information, *e.g.*, a location service only provide the current building instead of a room. A DiaSpec specification only reveals the data type of the information, nothing about the information itself is revealed nor are any relations given.

Concerning DiaAspect, we plan to extend the pointcut model to capture new join points; currently, our pointcut language only captures join points related to the relationships between components or to the built-in services of the runtime. A number of join points would be of particular interest: component declaration, component instantiation, *etc*.

BIBLIOGRAPHY

[1] Frances Aldrich. Smart homes: Past, present and future. In Richard Harper, editor, *Inside the Smart Home*, pages 17–39. Springer London, 2003. ISBN 978-1-85233-854-1.

[2] João Araùjo, Elisa Baniassad, Paul Clements, Anand Moreira, Awais Rashid, and Bedir Tekinerdoğan. Early aspects: The current landscape. Technical report, Lancaster University, 2005.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2 edition, April 2003. ISBN 0321154959. URL http://www.worldcat.org/isbn/0321154959.

[4] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996. ISSN 0001-0782. URL http://doi.acm.org/10.1145/230798.230809.

[5] Matthew A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201440997.

[6] Matthew A. Bishop. What is computer security? *IEEE Security and Privacy*, 1:67–69, 2003. ISSN 1540-7993. URL http://doi.ieeecomputersociety.org/10.1109/MSECP.2003.1176998.

[7] Julien Bruneau, Wilfried Jouve, and Charles Consel. DiaSim, a parameterized simulator for pervasive computing applications. In *Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems (Mobiquitous'09)*, pages 1–3, 2009.

[8] Muffy Calder and Alice Miller. Detecting feature interactions: How many components do we need? In *Objects, Agents, and Features*, pages 45–66, 2003.

[9] Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of LTL properties - a case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[10] Roy H. Campbell, Jalal Al-Muhtadi, Prasad Naldurg, Geetanjali Sampemane, and M. Dennis Mickunas. Towards security and privacy for pervasive computing. In *Proceedings of the International Symposium on Software Security*, pages 1–15, 2002.

[11] Johann Cas. Privacy in pervasive computing environments – a contradiction in terms. *Technology and Society Magazine*, 24(1):24–33, 2005.

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

[13] Damien Cassou, Benjamin Bertran, Nicolas Loriant, and Charles Consel. A generative programming approach to developing pervasive computing systems. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'09)*, pages 137–146, 2009.

[14] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Architecture-driven programming for sense/compute/control applications. In *Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH'10)*, 2010.

[15] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 431–440, 2011. URL http://hal.inria.fr/inria-00537789/en/.

[16] Harry Chen, Tim Finin, and Anupam Joshi. *Ontologies for Agents: Theory and Experiences*, chapter The SOUPA Ontology for Pervasive Computing, pages 233–258. Springer, July 2005.

[17] World Wide Web Consortium. Web Services Architecture, 2004. URL http://www.w3.org/TR/ws-arch/.

[18] Nicodemos Damianou, Naranker Dulay, Emil C. Lupu, and Morris Sloman. The Ponder policy specification language. In *Proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, pages 18–38, London, UK, 2001. Springer-Verlag.

[19] Nicodemos Damianou, Arosha K. Bandara, Morris Sloman, and Emil C. Lupu. A survey of policy specification approaches. Technical report, Imperial College of Science Technology and Medicine, London, 2002.

[20] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, 2001. URL http://dx.doi.org/10.1207/S15327051HCI16234_02.

[21] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998. ISBN 0764580434.

[22] Zoé Drey, Julien Mercadal, and Charles Consel. A taxonomy-driven approach to visually prototyping pervasive computing applications. In *Proceedings of the 1st IFIP Working Conference on Domain-Specific Languages (DSL'09)*, pages 78–99, 2009.

[23] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Oldenbourg Wissenschafts Verlag Gmbh, 2004.

[24] W. Keith Edwards and Rebecca 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. ISBN 3-540-42614-0.

[25] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 117–129, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. URL http://doi.acm.org/10.1145/336512.336542.

[26] European Parliament and the Council. Directive 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal L281*, 23:31–50, 1995.

[27] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943. URL http://www.worldcat.org/isbn/0321712943.

[28] David Garlan, Daniel P. Siewiorek, and Peter Steenkiste. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1:22–31, 2002.

[29] Stéphanie Gatti, Emilie Balland, and Charles Consel. A step-wise approach for integrating QoS throughout software development. In *Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering (FASE'11)*, pages 217–231, 2011.

[30] Jeremy Goecks and Elizabeth Mynatt. Enabling privacy management in ubiquitous computing environments through trust and reputation systems. Presented at *CSCW 2002 Workshop Privacy in Digital Environments: Empowering Users*, 2002.

[31] Sandeep K. S. Gupta, Tridib Mukherjee, and Krishna Venkatasubramanian. Criticality aware access control model for pervasive applications. In *Proceedings of the 4th International Conference on Pervasive Computing and Communications (PERCOM'06)*, pages 251–257, 2006.

[32] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, 1993. ISBN 0-89791-587-9. URL http://doi.acm.org/10.1145/165854.165932.

[33] Pablo A. Haya, Germàn Montoro, Abraham Esquivel, Manuel García-Herranz, and Xavier Alamán. A mechanism for solving conflicts in ambient intelligent environments. *Journal of Universal Computer Science*, 12(3):284–296, 2006.

[34] Urs Hengartner and Peter Steenkiste. Exploiting information relationships for access control in pervasive computing. *Pervasive and Mobile Computing*, 2:344–367, 2006.

[35] Urs Hengartner and Peter Steenkiste. Avoiding privacy violations caused by context-sensitive services. *Pervasive and Mobile Computing*, 2:427–452, 2006.

[36] Valerie Issarny, Luc Bellissard, Michel Riveill, and Apstolos Zarras. *Systems: Lecture Notes in Computer Science*, chapter Component-Based Programming of Distributed Applications, pages 327–353. Spriner-Verlag, 2000.

[37] Yongming Jin, Jinqiang Ren, Huiping Sun, Suming Li, and Zhong Chen. An improved scheme for delegation based on usage control. In *Second International Conference on Future Generation Communication and Networking (FGCN'08)*, volume 1, pages 74–78, 2008. doi: 10.1109/FGCN.2008.43.

[38] Lalana Kagal, Vladimir Korolev, Harry Chen, Anupam Joshi, and Tim Finin. Centaurus : A Framework for Intelligent Services in a Mobile Environment. In *International Workshop of Smart Appliances and Wearable Computing at the 21st International Conference of Distributed Computing Systems*, pages 195–201, 2001.

[39] Lalana Kagal, Jeffrey Undercoffer, Filip Perich, Anupam Joshi, Tim Finin, and Yelena Yesha. Vigil: Providing trust for enhanced security in pervasive systems. Technical report, University of Maryland Baltimore County, 2001.

[40] Lalana Kagal, Anupam Joshi, Jeffrey Undercoffer, Filip Perich, and Tim Finin. A security architecture based on

trust management for pervasive computing systems. In *Proceedings of Grace Hopper Celebration of Women in Computing*, 2002.

[41] Lalana Kagal, Vladimir Korolev, Sasikanth Avancha, Anupam Joshi, Tim Finin, and Yelena Yesha. Centaurus: an infrastructure for service management in ubiquitous computing environments, 2002. ISSN 1022-0038. URL http://dx.doi.org/10.1023/A:1020385804671.

[42] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, 2003.

[43] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24:779–796, October 1998.

[44] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997.

[45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, 2001. ISBN 3-540-42206-4.

[46] Arun Kumar, Neeran Karnik, and Girish Chafle. Context sensitivity in role-based access control. In *ACM SigOps*, pages 53–66, 2002.

[47] Marc Langheinrich. Privacy by design - principles of privacy-aware ubiquitous systems. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01)*, pages 273–291, 2001.

[48] Marc Langheinrich. A privacy awareness system for ubiquitous computing environments. In *Proceedings of the 4th International Conference on Ubiquitous Computing (UbiComp'02)*, pages 237–245, 2002.

[49] Emil C. Lupu, Naranker Dulay, Alberto Schaeffer Filho, Sye Keoh, Morris Sloman, and Kevin Twidle. Amuse: autonomic management of ubiquitous e-health systems. *Concurrency and Computation: Practice and Experience*, 20:277–295, 2008. ISSN 1532-0626. doi: 10.1002/cpe.v20:3. URL http://portal.acm.org/citation.cfm?id=1345491.1345494.

[50] Julien Mercadal, Quentin Enard, Charles Consel, and Nicolas Loriant. A domain-specific approach to architecturing error handling in pervasive computing. In *Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH'10)*, pages 47–61, 2010.

[51] Mary Moore. The evolution of telemedicine. *Future Generation Computer Systems*, 15(2):245 – 254, 1999. ISSN 0167-739X. doi: 10.1016/S0167-739X(98)00067-3. URL http://www.sciencedirect.com/science/article/B6V06-4037X3B-B/2/aff1f8221ada60b4c04627b3ff87f986.

[52] Erica Naone. Taking control of cars from afar. http://www.technologyreview.com/computing/35094/?a=f, March 2011. (retrieved: 3/15/2011).

[53] OECD. Guidelines on the protection of privacy and transborder flows of personal data, 1980.

[54] OMG. The common object request broker: Architecture and specification. Technical report, Object Management Group, 1995.

[55] Joon S. Park and Ravi S. Sandhu. The UCON usage control models. *ACM Trans. Inf. Syst. Secur.*, 7:128–174, 2004.

[56] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[57] Anand Patwardhan, Vladimir Korolev, Lalana Kagal, and Anupam Joshi. Enforcing Policies in Pervasive Environments. In *First International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, pages 199–308, 2004.

[58] Jennifer Pérez, Nour Ali, Jose A. Carsí, Isidro Ramos, Bárbara Álvarez, Pedro Sanchez, and Juan A. Pastor. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information and Software Technology*, 50(9-10):969–990, 2008. ISSN 0950-5849.

[59] Nicolas Pessemier, Lionel Seinturier, and Laurence Duchien. Components, ADL & AOP: Towards a common approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) at ECOOP'04*, pages 61–69, 2004.

[60] Ruben Picek and Vjeran Strahonja. Model driven development – future or failure of software development? In *18th International Conference on Information and Intelligent Systems (ISS'07)*, pages 407–414, 2007.

[61] Monica Pinto, Lidia Fuentes, and Jose Maria Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, pages 118–137, 2003. ISBN 3540201025.

[62] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *Proceedings of the 3rd International Conference on Pervasive Computing and Communications (PERCOM'05)*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[63] Saeed Rashwand and Jelena Mišić. A novel access control framework for secure pervasive computing. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference (IWCMC'10)*, pages 829–833. ACM, 2010.

[64] Jie Ren and Richard N. Taylor. A secure software architecture description language. In *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics, held in conjunction with the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[65] Jie Ren, Richard Taylor, Paul Dourish, and David Redmiles. Towards an architectural treatment of software security: A connector-centric approach. In *Approach, Proceedings of the Workshop on Software Engineering for Secure Systems, held in conjunction with the 27th International Conference on Software Engineering*, pages 1–7, 2005.

[66] Daniel Retkowitz and Sven Kulle. Dependency management in smart homes. In *Proceedings of the 9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'09)*, 2009.

[67] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1:74–83, 2002. ISSN 1536-1268. URL http://doi.ieeecomputersociety.org/10.1109/MPRV.2002.1158281.

[68] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. SIP: Session Initiation Protocol. Technical report, RFC 3261, 2002. URL http://www.ietf.org/rfc/rfc3261.txt.

[69] Giovanni Russello, Changyu Dong, and Naranker Dulay. Authorisation and conflict resolution for hierarchical domains.

In *Proceedings of the 8th International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*, pages 201–210, 2007.

[70] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[71] Geetanjali Sampemane. *Access Control For Active Spaces*. PhD thesis, University of Illinois, 2005.

[72] Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell. Access control for active spaces. In *Proceedings of the 18th Annual Computer Security Applications Conference (ASAC'02)*, pages 343–352, 2002. doi: 10.1109/CSAC.2002.1176306.

[73] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996. ISSN 0018-9162. URL http://dx.doi.org/10.1109/2.485845.

[74] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–18, August 2001.

[75] Estefanía Serral, Pedro Valderas, and Vicente Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6:254–280, April 2010. ISSN 1574-1192. URL http://dx.doi.org/10.1016/j.pmcj.2009.07.006.

[76] Nigel Shadbolt. Ambient intelligence. *IEEE Intelligent Systems*, 18(4):2–3, 2003. ISSN 1541-1672.

[77] Jeffrey S. Shell. Taking control of the panopticon: Privacy considerations in the design of attentive user interfaces. Presented at *CSCW 2002 Workshop Privacy in Digital Environments: Empowering Users*, 2002.

[78] Choonsung Shin, Anind K. Dey, and Woontack Woo. Mixed-initiative conflict resolution for context-aware applications. In *Proceedings of the 12th International Conference on Ubiquitous Computing (UbiComp'08)*, pages 262–271, 2008.

[79] Morris Sloman and Emil C. Lupu. Engineering policy-based ubiquitous systems. *The Computer Journal*, 53(7):1113–1127, 2010. doi: 10.1093/comjnl/bxp102. URL http://comjnl.oxfordjournals.org/content/53/7/1113.abstract.

[80] João Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the 3rd Working Conference*

on Software Architecture, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. ISBN 1-4020-7176-0.

[81] Vince Stanford. Using pervasive computing to deliver elder care. *Pervasive Computing, IEEE*, 1(1):10 – 13, 2002. ISSN 1536-1268. doi: 10.1109/MPRV.2002.993139.

[82] Christopher Steel, Ramesh Nagappan, and Ray Lai. *Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall, 2007.

[83] Sun Microsystems. Entreprise Java Beans specification, 2007. URL http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[84] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, 1999. ISBN 1-58113-074-0.

[85] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practices*. Addison-Wesley, 2010.

[86] William Tolone, Gail-Joon Ahn, Tanusree Pai, and Seng-Phil Hong. Access control in collaborative systems. *ACM Computing Surveys*, 37:29–41, 2005.

[87] Verena Tuttlies, Gregor Schiele, and Christian Becker. COMITY - conflict avoidance in pervasive computing environments. In *Proceedings of the 2nd International Workshop on Pervasive Systems (PerSys'07)*, pages 332–345, 2007.

[88] Mark Weiser. The computer for the twenty-first century. In *Scientific American*, volume 265, pages 94–104, 1991. URL http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.

[89] Mark Weiser and John Seely Brown. *The coming age of calm technolgy*, pages 75–85. Copernicus, New York, NY, USA, 1997. ISBN 0-38794932-1. URL http://portal.acm.org/citation.cfm?id=504928.504934.

[90] Mark Weiser, Rich Gold, and John Seely Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM Syst. J.*, 38:693–696, 1999. ISSN 0018-8670. URL http://dx.doi.org/10.1147/sj.384.0693.

[91] Wikipedia, 2011. URL http://en.wikipedia.org/wiki/Aspect_oriented_programming.

[92] Raj Yavatkar, Dimitrios Pendarakis, and Roch Guerin. Framework for policy-based admission control. Technical report, RFC 2753, 2000. URL http://www.ietf.org/rfc/rfc2753.txt.

[93] Feng Zhu, Matt W. Mutka, and Lionel M. Ni. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, 4:81–90, 2005. doi: 10.1109/MPRV.2005.87.

Part V

APPENDICES

A

DIASPEC CODE SAMPLES

```
/* Description of the available entities. */
device LocatedDevice {
  attribute location as Location;
}
device Alarm extends LocatedDevice {
  action OnOff;
}
device BreakDetector extends LocatedDevice {
  source broken as Boolean;
}
device Camera extends LocatedDevice {
  source presence as Boolean;
  source picture as JPEG;
  action Move;
  action Track;
}
device Door extends LocatedDevice {
  source status as DoorStatus;
  action LockUnlock;
}
device Keypad extends LocatedDevice {
  source status as HomeStatus;
}
device Logger {
  action Log;
}
device Messenger {
  action Send;
}
device SmokeSensor extends LocatedDevice {
  source smoke as Float;
}
device Sprinkler extends LocatedDevice {
  action OnOff;
}
device TemperaturSensor extends LocatedDevice {
  source temperature as Float;
}
/* Description of the supported actions. */
action LockUnlock {
  lock();
  unlock();
}
action Log {
  logEvent(event as String);
}
action Move {
  role(degree as Integer);
  pitch(degree as Integer);
  yaw(degree as Integer);
```

```
}
action OnOff {
  on();
  off();
}
action Send {
  send(message as String);
  send(picture as JPEG);
}
action Track {
  trackPresence();
}
/* Enumerations and data types. */
enumeration DoorStatus {
  LOCKED, UNLOCKED, OPEN
}
enumeration HomeStatus {
  SECURED, UNSECURED
}
structure Location {
  location as String;
}
structure JPEG {
  name as String;
  content as Binary;
}
```

Listing 13: Taxonomy of the emergency application

```
/* Context components. */
context AvgTemp as Float indexed by location as Location {
  source temperature from TemperatureSensor;
}
context DoorStatus as Boolean indexed by location as Location {
  source status from Door;
}
context Fire as Boolean indexed by location as Location {
    context AvgTemp;
    context SmokeDetected;
}
context Intrusion as Boolean indexed by location as Location {
  context Occupancy;
  source broken from BreakDetector;
}
context Occupancy as Boolean indexed by location as Location {
  source presence from Camera;
  source status from Keypad;
}
context SmokeDetected as Boolean indexed by location as Location {
  source smoke from SmokeSensor;
}
context Surveillance as JPEG indexed by location as Location {
  source picture from Camera;
}
/* Controller components. */
controller FireCtrl {
  context DoorStatus;
  context Fire;
```

```
30    action LockUnlock on Door;
31    action Log on Logger;
32    action OnOff on Alarm, Sprinkler;
33  }
34  controller IntrusionCtrl {
35    context Intrusion;
36    context Surveillance;
37    action LockUnlock on Door;
38    action Log on Logger;
39    action OnOff on Alarm;
40    action Send on Messenger;
41    action Track on Camera;
42  }
```

Listing 14: The emergency application

B

JAVA CODE SAMPLES

```java
public class CertificateHelper {
  public Certificate certificate;
  public String owner;
  private String keystorePath;
  private char[] keyPass;
  private char[] trustPass;

  /** Constructor. */
  public CertificateHelper(ServiceConfiguration config) {
    this.owner = config.getOwner();
    this.keystorePath = config.getKeystorePath();
    this.keyPass = config.getKeyStorePassword();
    this.trustPass = config.getTrustStorePassword();
    this.certificate = this.getCertificateFromKeystore(owner);
  }

  /** Checks if local trust store contains certificate of <alias>.
      */
  public boolean containsCertificate(String alias) { ... }

  /** Returns the local certificate. */
  public Certificate getCertificate() {
    return this.certificate;
  }

  /** Returns certificate of <alias> from the local key store. */
  public Certificate getCertificateFromKeystore(String alias) {...}

  /** Returns certificate of <alias> from the local trust store. */
  public Certificate getCertificate(String alias) {
    KeyStore ks;
    try {
      ks = this.getTruststore();
      return ks.getCertificate(alias);
    } catch (Exception e) {
      DiaLog.error(" -- Couldn't retrieve certificate " + alias +
          " from the trust store.");
      e.printStackTrace();
    }
    return null;
  }

  /** Stores a given certificate under the given alias in the
      local trust store. */
  public void storeCertificate(String alias, Certificate cert) {
    KeyStore ks;
    try {
      ks = this.getTruststore();
      if (ks.isCertificateEntry(alias)) {
```

```
47          DiaLog.info(" -- " + alias + "'s Certificate is already in
                the trust store.");
48        } else {
49          ks.setCertificateEntry(alias, cert);
50          FileOutputStream fos;
51          fos = new FileOutputStream(keystorePath + owner +
                "Trust.jks");
52          ks.store(fos, this.trustPass);
53          fos.close();
54          DiaLog.info(" -- Stored " + alias + "'s certificate into "
                + owner + "'s trust store.");
55        }
56      } catch (Exception e) {
57        e.printStackTrace();
58      }
59    }
60
61    /** Compares a received certificate to the local certificate. */
62    public boolean verify(String alias, Certificate cert) { ... }
63
64    /** Returns the local key-store. */
65    private KeyStore getKeystore() {
66      try {
67        KeyStore ks = KeyStore.getInstance("JKS");
68        ks.load(null, this.keyPass);
69        FileInputStream fis = new FileInputStream(keystorePath +
              owner + ".jks");
70        ks.load(fis, this.keyPass);
71        fis.close();
72        return ks;
73      } catch (Exception e) {
74        e.printStackTrace();
75      }
76      return null;
77    }
78
79    /** Returns the local trust store. */
80    private KeyStore getTruststore() { ... }
81  }
```

Listing 15: The `CertificateHelper` class that manages certificates in trust and key stores.