TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Security Policies in Pervasive Systems

Design of a Modular Security Policy Framework for Semantic, Multi-Domain, Service-Oriented Pervasive Systems

Julian Hendrik Schütte

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter	
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Claudia Eckert
	2.	Univ.-Prof. Dr. Stefan Katzenbeisser, Technische Universität Darmstadt

Die Dissertation wurde am 20.12.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.04.2013 angenommen.

Acknowledgements

This thesis would not have been possible without the support and encouragement by many people. First, I want to thank Prof. Dr. Claudia Eckert, head of the Fraunhofer Institution AISEC and the Chair for IT Security at the Technical University of Munich. The trust she put in my scientific abilities was a great motivator. She gave me the freedom to work on this thesis and was always available for constructive feedback. Also, I want to thank Prof. Dr. Stefan Katzenbeisser, head of the Security Engineering Group at the Technical University of Darmstadt, for kindly offering a second opinion on this thesis.

I further have to thank Dr. Kpatcha Bayarou and Mario Hoffmann. They gave me the opportunity to work in the project HYDRA, funded by the European Commission, where I could get deep insights in pervasive systems architectures and understand their specific challenges.
Kpatcha always had an open ear and I enjoyed the interesting discussions with him. Mario was an inspiring sparring partner in the HYDRA project and gave me the chance to finish this thesis despite a heavy workload at the department.

Writing this thesis in parallel to my work at Fraunhofer would not have been possible if I were not motivated by my great colleagues and friends at the institutes in Darmstadt and Munich. I am glad for the time we shared and thankful for their support during the time of writing up this thesis.

During the process of writing I further became increasingly thankful for Donald Knuth (TEX) and Leslie Lamport (LATEX) to create the greatest typesetting system on earth and the LYX development team for making it a joy to use.

My parents receive my deepest gratitude for their unconditional love and support. Without their firm believe in my abilities I would never have finished my studies nor this thesis. Last but certainly not least I sincerely thank Carolin for her understanding and continual encouragement, her love, and endless patience.

Kurzfassung

Unter *Pervasive Systems* versteht man Softwarearchitekturen für verteilte Systeme, in denen Dienste dynamisch miteinander kombiniert werden können. Solche Architekturen bilden die Grundlage für sog. “intelligente Umgebungen” – also Anwendungen, die dem Benutzer kontextabhängige Dienste anbieten, indem sie in der Umgebung vorhandene Sensoren, Aktoren und Dienste miteinander vernetzen. Mit der wachsenden Anzahl und der immer größeren Leistung eingebetteter Systeme nimmt auch deren Vernetzung und damit der Bedarf an Pervasive-Systems-Architekturen zu. Die Komplexität dieser Architekturen stellt jedoch nach wie vor eine Herausforderung dar, insbesondere was die Einhaltung von Sicherheitsanforderungen betrifft. Da in sich dynamisch verändernden Architekturen eine manuelle Konfiguration von Sicherheitsmechanismen nicht mehr möglich ist, müssen diese Systeme in der Lage sein, sich autonom zu adaptieren, um Sicherheitsanforderungen einzuhalten. Um die Definition solcher Anforderungen von der eigentlichen Implementierung des Systems zu entkoppeln, werden Sicherheitsrichtlinien, oder *Security Policies*, angewandt. Ein sog. *Policy Framework* stellt hierbei die Funktionen zur Auswertung und Durchsetzung der Richtlinien zur Verfügung und muss entsprechend in die Systemarchitektur integriert werden. Existierende Frameworks sind jedoch nicht hinreichend für die speziellen Anforderungen von Pervasive Systems geeignet. Deren anwendungsunabhängige Systemarchitektur erfordert, dass Policy Frameworks auf nahezu beliebige Sicherheitsmodelle adaptiert werden können, um den Anwendungen gerecht zu werden, die auf der Systemarchitektur aufbauen. Darüber hinaus müssen Regeln über a priori unbekannte Ressourcen und Subjekte definiert werden können, Richtlinien müssen autonom zwischen mehreren Parteien ausgehandelt werden und nicht zuletzt müssen Benutzer die komplexen Regeln auf Fehler hin analysieren können.

Gegenstand dieser Arbeit ist daher der Entwurf eines Frameworks für Sicherheitsrichtlinien in Pervasive Systems. Kern des Frameworks ist ein erweiterbares generisches Richtlinienmodell in Beschreibungslogik, sowie eine modulare Softwarearchitektur zur Auswertung und Durchsetzung der Richtlinien. Ausgehend von diesem Kern-Framework wurden typische Herausforderungen in Pervasive Systems angegangen und Lösungsstrategien vorgestellt.

Zunächst wurde gezeigt, wie die einfachen Konzepte des generischen Richtlinienmodells derart erweitert werden können, um abstraktere und damit verständlichere “High-level Policies” zu definieren. Zweitens wurde eine Erweiterung des herkömmlichen Event-Condition-Action-Musters vorgestellt, dass dem System erlaubt, sich situationsabhängig und automatisch zu rekonfigurieren, um vorgegebene Schutzziele zu erreichen. Drittens wurden sog. *Meta-Policies* modelliert, um Konflikte zwischen mehreren Parteien zu behandeln, ohne dass diese ihre Richtlinien einander preisgeben müssen. Schließlich wurden auf Auktions-Protokollen basierte Strategien zur autonomen Vereinbarung von Maßnahmen untersucht, wobei individuelle Präferenzen der Parteien berücksichtigt werden.

Durch die Umsetzung eines Prototyps für das Kern-Framework wurde die Praktikabilität seiner Software-Architektur, sowie des Beschreibungslogik-basierten Richtlinienmodells gezeigt. Darüber hinaus wurde jeder der o.g. Lösungsansätze in Form von Plugins für das Kern-Framework prototypisch realisiert, und damit belegt dass das Policy Framework typische Herausforderungen in Pervasive Systems lösen kann.

Abstract

Pervasive systems denote architectures of distributed and dynamically orchestrated components and build the basis for "intelligent environments", i.e. context-aware applications which combine ubiquitous networked sensors and services. While the advantages of such loosely coupled architectures are not to be denied, automatically establishing and configuring mechanisms like secure communication protocols or access control is a challenge. Security policies are a way to separate non-functional requirements from the functional implementation of a system. However, most traditional policy models do not take into account the dynamic nature of pervasive systems and fall short of addressing issues like extensibility, reasoning over rules, as well as negotiating and aligning policies across domain boundaries. The contribution of this thesis is the design of a policy framework for pervasive systems which is build around an extensible description logic based core policy model and a modular software architecture. Based on the core framework, we propose and evaluate different strategies to address four typical policy challenges in pervasive systems: first, we show how developers can create increasingly abstract policy models by extending the built-in core model of the framework with additional concepts and relations, and show how to validate model constraints. Second, we propose an advancement of the event-condition-action (ECA) pattern which allows users to write goal-based policies for autonomous reconfigurations of security mechanisms. Third, we show how metapolicies can be used to align decisions from composed policy domains. Fourth, we investigate strategies to allow parties to negotiate policy decisions considering individual preferences. Specifically, we explore the integration of micro-economic approaches like voting and auction protocols into the policy framework. By a prototype implementation of the framework and its add-on modules, we could show that the concept of an extensible, description-logic based policy model is in fact practicable and that it can be used to control security mechanisms in "self-protecting", i.e. adaptive, pervasive systems.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The notion of intelligent environments which react to changing situations and provide context-adaptive user interfaces and services turned from science fiction into a research subject in the early nineties. Since then, terms like *ubiquitous computing*, *pervasive systems*, and *ambient intelligence* have been coined. Although each term focuses on slightly different aspects, they all comprise the vision of environments of embedded computers, working autonomously in the background and recognising situations in order to provide users with adapted tools and information. Mark Weiser was one of the first to formulate this vision in his well-known *Scientific American* article *The Computer for the 21st Century* [193]:

> *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.*

To some extent, this vision has become reality with respect to miniaturisation and capabilities of devices. Mobile phones, homes, and even power grids have become "smart". That is, they are able to interact with their environment and to take "intelligent" decisions based on varying conditions. However, with respect to a universal and autonomous interconnection of such devices, Mark Weiser's vision is still guiding research on self-adaptive systems. Self-adapting distributed systems aiming at an autonomous orchestration of heterogeneous devices and the services they provide are commonly called *pervasive systems,* and are a main pillar of the "smart" applications of the future. Today, pervasive systems still raise challenges in several areas. For example, the degree of autonomy required in order to cope with a constantly changing system configuration has not been achieved yet.

The challenges become especially manifest when it comes to security properties which must be maintained while beforehand unknown devices and services can join or leave the network at any time. Obviously, it is not possible to manually reconfigure the system whenever the environmental conditions or the system architecture itself have changed. Thus, *policies* are indispensable in order to control the security of a pervasive system. Policies can be regarded as a limited programming language for non-functional requirements. That is, they allow users to specify their requirements in the form of machine-readable rules which are then applied by a *policy framework* to the technical components under control. The approach of using policies to detach security configurations from the system's implementation has been in use for decades. For example, firewalls and file systems use policies to control accesses to network resources and files, rather than hard coding access rights into every resource.

While various policy frameworks for distributed systems have been proposed, most of them are not well suited for pervasive systems. For this specifically demanding class of architectures, policy frameworks must be able to deal with devices and services which are not known at the time of writing the policy, must be able to handle conflicts between multiple administrative domains, and must support self-adaptive infrastructures. Further, the actual policy model, i.e. the structure and evaluation semantics of rules is highly application-specific. While for one application, a simple static access control scheme might suffice, another application might put higher demands on the model, such as context-specific reconfigurations of services, policy negotiations across domain boundaries, etc. Thus, building a policy framework upon a single predetermined model is not well suited for pervasive systems. Rather, it should be possible to "policy-enable" the system in advance and leave the choice of the policy model to the application developers using the underlying pervasive system infrastructure.

In this thesis, a policy framework for pervasive systems has been developed. It supports extensible policy models, semantic high-level rules, as well as negotiations and conflict handling across policy domains. The following section will motivate the problem statement in more detail. Section 1.2 points out the main contributions of this thesis to related research areas, and section 1.3 gives an overview of the structure of this document.

1.1. Problem statement and motivation of this thesis

Pervasive systems are a class of distributed computing architectures in which heterogeneous and often mobile platforms are connected in an ad hoc fashion [165]. Such architectures are the basis of upcoming "smart" context-aware and self-adapting applications, often summarised by the term *ambient intelligence*. One of the main paradigms of pervasive systems, without which such self-adaptivity would be impossible is *loose coupling*, meaning that the collaboration between service providers and consumers is not predetermined at design time, but rather services are discovered and bound at run time. While the advantage of loosely coupled architectures and ad hoc connectivity is that they do not require to hard wire component interactions at design time, this flexibility is also what creates difficulties in the context of security mechanisms. Without knowing the exact identity and properties of the components which will be present in the system at a later time, application developers cannot determine which security protocols or authentication mechanisms will be appropriate, i.e. in line with the capabilities of the components, as well as the performance and security requirements of the application.

Different policy languages have been applied for decades in order to abstract away the specification of non-functional requirements from the actual functionality of the system, and nowadays policies are widely used as a kind of limited programming language which allows adapting the behaviour of components without modification of their functionality. Although a plethora of security policy models and languages has been developed in the past, pervasive systems show certain characteristics which make the applicability of existing policy models difficult or insufficient, as we will discuss in this subsection.

Service properties are not known at design time A result of the loose coupling paradigm is that clients and services are not known to each other at design time. As a consequence, clients do not know the properties of a service in advance, like they would do in a traditional

SOA where the service provider would be known and Service Level Agreements (SLA) had been negotiated beforehand. It is therefore necessary to make sure that application developers can formulate demands on the properties of a component at an abstract level at design time and to enforce them on specific service implementations at run time.

Applications with different security models Applications built on a pervasive systems middleware can serve various purposes, ranging from eHealth, over home automation, on to smart grid, and even "cyber-physical" applications. Each application will have its own requirements on how security mechanisms are applied and accesses are granted. As most policy frameworks are based on a certain policy model, such as attribute- or role-based access control (ABAC, RBAC), for example, they might be suited for some applications, but fail to meet the demands of others. This is unfavourable, as it would require users to commit on a specific policy model when designing the system and does not leave room for extensions of that model. In order to make a policy framework sustainable and able to support multiple applications on the same pervasive systems middleware, it is therefore required that the framework allows to add additional features to an existing policy model or even replace the whole model by different one.

Usage of semantic descriptions In many cases, pervasive systems make use of semantic service descriptions, for instance in order to discover matching services. To allow policy authors to refer to such semantically annotated services, it is necessary that the policy language supports semantic descriptions of subjects and resources, and the respective actions to be taken.

Increasing complexity With an increasing number of devices and services and the integration of context-awareness, pervasive systems show greater dynamics and complexity than traditional service-oriented architectures. As a consequence, the task of specifying security policies will become more error-prone and understanding all effects of an existing policies will be even harder than it already is. To remedy this problem, policies should be specified at an abstract level which is closer to the actual intention (i.e., the security model) of the policy author. The policy framework should then automatically refine these high-level policies to enforceable low-level policies. Furthermore, reasoning over policies will be required in order to provide advanced analysis services to users, such as answering "what-if" questions or identifying whether a certain security property will be enforced as desired. Most existing frameworks do not support this feature as they represent policies in a dedicated format which does not allow to apply generic reasoning tasks.

Policy decisions affect multiple domains While most policy models assume that a resource is under the control of a single administrative domains, this is not always the case in a pervasive system. It will be rather be a common situation for a service to be under control of two or more domains and thus different policies, as shown in Figure 1.1. For instance, a mobile phone (or the services running on it) could be subject to policies defined by the phone's owner, as well as to policies defined by the company's network it currently resides in. Another example would be a service that is subject to generic company policies and more specific policies from a subordinate department. In these situations there is no single "best" way to resolve conflicts between the different domains — one could simply prefer the decisions of the most specific domain or let the superior's domain decisions overwrite all others. Also, not in all cases will it be possible to arrange domains hierarchically and thus, the policy framework must provide means to define conflict detection and handling strategies which do not rely on a specific infrastructure or application. Another issue is that policies contain security-relevant information which must not be revealed to outsiders and must therefore be regarded as private. Part of this thesis will thus be an approach to detect and combine conflicting decisions without requiring domains to reveal their policies.

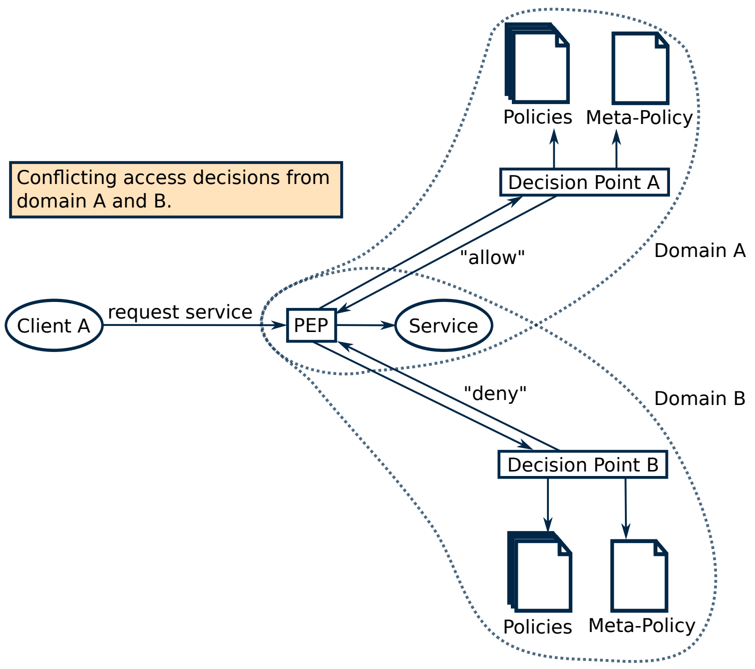


Figure 1.1.: Conflicting access decisions from different domains

So, while pervasive systems and their loosely coupled architectures are predestined to be controlled by policies, current security policy frameworks are not fully suited for the particular requirements of pervasive systems. Neither do they provide a satisfying integration of semantic knowledge, support the specification of high-level requirements, nor do they have the ability to deal with conflicts across domain boundaries.

This thesis aims at closing this gap by developing a framework that allows software developers to "policy-enable" an existing pervasive system. The developed framework will make extensive use of a semantic knowledge base and information inferred from it, provide components for policy decision and enforcement and support the resolution of the above mentioned cross-domain conflicts. Furthermore, it will define extension points for adding additional high-level policy patterns which shall facilitate the specification of policies and make the better understandable for humans. Architectural patterns for orchestrating these components will be proposed, thereby making the solution a *framework*, in contrast to a *library* or a *toolkit*. As part of the evaluation, the applicability of the proposed framework in typical pervasive systems will be tested and the benefits of semantic representations, such as an inherent formalisation and improved analysis features, will be balanced against the expected drawbacks such as higher computing costs.

1.2. Contribution of this thesis

The research carried out in this thesis aims at developing a policy framework which considers the specific requirements of pervasive systems. It is originally based on experiences made during the development of the LinkSmart middleware[1] during the EU-funded project HYDRA [7], where we noted that controlling pervasive systems is hardly possible using existing policy frameworks. While LinkSmart includes in fact an XACML engine, we soon became aware that XACML is not well suited to refer to dynamically composed entities which do not have previously known identifiers or attributes, but merely semantic descriptions. As a consequence, we made first approaches towards a semantically enhanced XACML, One result of these efforts is the semantic XACML analysis tool[2], written by the author of this thesis. However, despite the semantic extensions, a dedicated policy language like XACML turned out not to be the perfect way to policy-enable a pervasive system. From that insight, the research work in this thesis has been derived.

As a guideline for the work carried out in this thesis, the following four research questions subsume the main goals to strive for:

Research question 1 *Which policy models are required in typical pervasive systems and in which way must a framework be constructed in order to support all these policy models?*

Research question 2 *How can policy conflicts, which occur in typical pervasive systems, be handled?*

Research question 3 *How can extensible, analysable and high-level policies be modelled and integrated with existing domain knowledge?*

Research question 4 *What are best practice design patterns for policy frameworks in pervasive systems?*

The aim of the first research question is to identify typically required policy models in pervasive systems and to design a software architecture supporting all of these required models. By answering this question we intent to provide a profound basis for the rest of this thesis.
The second question deals with classifying policy conflicts and deriving possible methods for resolving them. Answering this question is especially of interest as in pervasive systems, administrative policy domains will frequently overlap and policy decisions will get into conflict. Identifying situations in which policy domains overlap (e.g. when a service is under control of multiple PDPs) and researching strategies of resolving such conflicts will be part of answering this question.
Subject of the third question is to investigate ways to address the problems which common policy models have in pervasive systems. The question is on the one hand motivated by the shortcomings of traditional policy frameworks, when they are applied in pervasive systems, such as increasing complexity, static and limited policy models, and scarce support for analysis features. On the other hand, it addresses the increased use of semantic information for service description, discovery, and selection, and the resulting need to make it usable for policy specification. Integrating this information into policy specifications and decisions will facilitate the application of policies in pervasive systems and will allow users to formulate policies at a more abstract level which is closer to their actual intention.
The motivation of the fourth question is to investigate design patterns and technologies which are suited for realising the policy framework architecture developed in this thesis.

[1]`http://sourceforge.net/projects/linksmart/`

[2]`https://github.com/quadriat/XACML-Policy-Analysis-Tool`

The answer to this question is expected to provide best-practice experiences for an implementation which matches the requirements of pervasive systems. In order to validate the results of this investigation, parts of the framework will be implemented in form of proof-of-concept prototypes.

Any research work is supposed to be seen in the wider context of existing work from different areas and its contribution to the state of the art. The specific research areas this thesis is mainly based on are as follows:

1. Software architectures for modular and distributed systems
   As this thesis aims at pervasive systems, we have to consider architectural patterns and typical communication flows in such systems. When designing the policy framework, it must be taken into account that in pervasive systems, services often communicate in an asynchronous and sometimes anonymous way by means of *Publish-Subscribe* and *Whiteboard* patterns, for example.

2. Policy languages and frameworks
   As the result of this thesis will be a policy framework for pervasive systems, it is obvious that existing work on policy languages and frameworks has to be reviewed, compared against the requirements stated in this thesis and improved or complemented in aspects where the requirements are not satisfactorily met.

3. Semantic Web Technology and the underlying description logics
   One part of this thesis is to support reasoning over policies for the sake of refinement and analysis. In this context, technologies from the semantic web community will be investigated and applied, including ontologies, semantic service annotations and reasoners. Further, the properties of the underlying description logics will be analysed and judged in the context of the developed policy framework.

4. Context-aware systems
   One main characteristic of pervasive systems is context-awareness, and thus policies must be able to take changing context information into account. The current state of the art in the area of context-awareness research must therefore be considered during design and evaluation of the policy framework.

During the work on this thesis, the following contributions have been made to these research areas: the development of the framework design reveals insights into patterns and protocols for the interaction of policy-enabling components in pervasive systems, as well as for their integration into reactive and context-aware architectures. The extensible semantic policy model proposed in this thesis is based upon the state of the art in semantic web technologies and description logics and contributes to the field of policy languages and frameworks, just like the proposed approach on cross-domain conflict handling. Based upon the core framework, this thesis proposes some exemplary extension modules which target at specific policy problems in pervasive systems and thereby contribute on the one hand to the state of the art in policies, as well as to the field of dynamic software architectures and pervasive systems.
Figure 1.2 depicts the research tasks and the areas of contributions of this thesis.

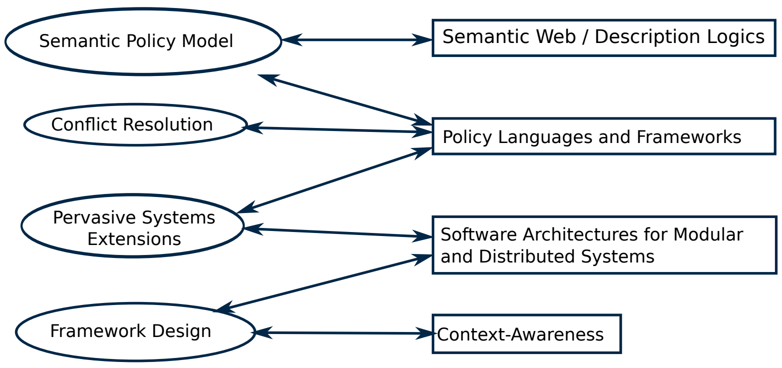

Semantic Policy Model
Conflict Resolution
Pervasive Systems Extensions
Framework Design
Semantic Web / Description Logics
Policy Languages and Frameworks
Software Architectures for Modular and Distributed Systems
Context-Awareness

Figure 1.2.: Research tasks of this thesis (l.) and correlation to research areas (r.)

1.3. Organisation of this thesis

This document is organised as follows:

Chapter 2 introduces background information required for understanding this thesis. Firstly, it provides an understanding of pervasive system architectures, their main characteristics which lead to the challenges we addressed, as well as the differences to other types of distributed systems. Secondly, this chapter gives an introduction into Semantic Web Technology (SWT) in general, and description logics in particular. Readers who have already an understanding of pervasive systems or description logics and the respective notations can thus skip these sections.

Chapter 3 reviews existing work related to the overall goal of this thesis, i.e. the state of the art in policy frameworks and their application to pervasive systems. In addition to this overall state of the art discussion, we will review work related to specific problems, for which we developed solutions in the course of this thesis, where they are relevant, in dedicated sections in chapter 7. From the state of the art analysis, a set of requirements which are crucial to policy-enabling pervasive systems is derived in chapter 4. Subsequently, the approach of this thesis and its intention to address these requirements is sketched.

Chapter 5 introduces the core policy model – basic concepts and evaluation workflows for simple access control and reactive policies, which build an easily extensible basis for more advanced modules, tailored to solve specific pervasive systems problems.

In chapter 6, the overall software architecture of the framework is introduced. Based upon the aforementioned core policy model, the architecture allows to policy-enable a pervasive system and provides extension points which developers can use to adapt the framework to the specific needs of a pervasive system application. Further, we give insights into design choices and the prototypical implementation of the framework.

While the core framework's main purpose is to integrate with an underlying middleware and to provide the functional skeletons for evaluating policies based on the model, chapter 7 discusses several extensions of framework which are based on the core model and solve typical pervasive system challenges: by means of a dynamic role-based access control (DRBAC) extension we show how it is possible to abstract from the core concepts and realise increasingly abstract policy models. An extension for situation-based security reconfiguration illustrates how reactive policies can be applied which go beyond the traditional event-condition-action pattern and are closer to the vision of a "self-protecting" system. Furthermore, a section on handling cross-domain policy decision shows how the metapolicy capabilities of the framework can be used in order to make policy domains collaborate at run time, without violating their individual requirements. Finally, the chapter presents an extension for taking individual preferences during multilateral negotiations into account, thereby illustrating how pervasive systems can be realised which automatically tune themselves towards an optimal trade-off between the priorities of multiple domains. Further, the experiences and results gained from prototypical implementations of these extensions are discussed.

Chapter 8 finally concludes this thesis by critically discussing its results and highlighting directions for interesting future research questions.

CHAPTER 2

Background

In this thesis we design a policy framework for pervasive systems which is based on semantic technologies. In this chapter, the reader will be introduced to the required background knowledge on distributed and loosely coupled service infrastructures, as well as Description Logics for semantic knowledge representation.

2.1. Distributed pervasive systems

Distributed systems have been in use for decades and nowadays, they are one of the most important types of system architecture – in fact, only few systems are not "distributed" in some way. So, the term "distributed system" is very broad and comprises various classes of systems which serve completely different purposes and come each with their own challenges. Following the classification from [165], we will point out only the most important classes of distributed systems in order to distinguish them from the pervasive systems considered in this thesis.

2.1.1. Distributed system architectures

High performance computing *High performance computing* (HPC) systems such as clusters and grids aim at interconnecting many individual hosts and exposing them to the user as a single powerful computing resource. Grids mainly differ from clusters in that they are set up on heterogeneous nodes which may be operated by different companies, while a cluster usually consists of numerous identical nodes. However, in both cases, the nodes are explicitly configured to be part of a specific cluster or grid and are connected to a common static network infrastructure. That is, these systems do not aim at a spontaneous connection of new nodes and expect all nodes to provide a predetermined functionality.

Distributed enterprise applications While HPC systems aim at providing a network of computing resources which are independent from any application, the need for distribution at the application layer emerged soon and protocols for Remote Procedure Calls (RPC) have been used to create *Distributed Enterprise Applications*, in which application components

communicate with each other. These systems still use closely coupled components, that is, components which communicate directly and synchronously with each other. However, this close coupling has two main drawbacks: firstly, the failure of a single component will most likely result in failure of the overall application, so closed coupling is counter-productive to fault tolerance. Secondly, closed coupling requires components to be available at static, predefined addresses, so it is not possible to move them across resources, to replace them or even to add new components at run time.
So, distributed enterprise applications take the concepts of Object Oriented Programming (OOP) and simply distribute the individual components, but do not support on-the-fly modifications of the component infrastructure.

Service oriented architectures The advent of *Service Oriented Architectures* (SOA) was an attempt to move from specialised distributed application components to generic services which communicate over standardised protocols and can be orchestrated as required by an application. At the communication layer, SOAs are usually based on web services and the SOAP protocol. Most of the respective specifications by OASIS[1] have received a status of overall acceptance and are supported by various architectural frameworks, so that the SOA claim of interoperable service provisioning across different platforms has become reality to some extent.
By the integration of Enterprise Service Buses (ESB), SOA aims at de-coupling the direct binding between service providers and consumers and to outsource non-functional operations on service messages from the actual service implementation – such as message encryption, signature, encoding or transformation.
Further, SOA originally addressed the issue of service discovery, mainly by the UDDI [33] service directory – often called the "yellow pages" of web services. However, the UDDI specification turned out to be overly complex and not suitable for practical use, as it is focused at a direct interaction between web service consumers and providers and therefore does not take anonymous communication between services into account. While it is still supported by some SOA frameworks like WSO2[2], main players like IBM have officially dismissed UDDI, so it did not achieve a status of overall adoption. Also, the vision of truly reusable and generic services has not yet become reality as most services in a SOA are still application specific.

Cloud Computing Cloud computing can be regarded as an evolution of SOA, with a focus on service provisioning, rather than on service discovery and orchestration. From the frequently cited NIST definition [107], it becomes obvious that the focus of cloud computing is not on specific software architectures or communication protocols, but on a multi-tenant, on-demand provisioning of services and the ability to rapidly scale resources, if necessary. While in the case of a simple Software-as-a-Service (SaaS) cloud computing refers merely to a client-server architecture, scenarios become more complex in the case of so-called *Interclouds* where cloud installations of different providers interact with each other. In Intercloud scenarios, one cloud provider may use the resources of another provider which is either geographically near to the requester or has capabilities and resources the original provider cannot offer. However, at the time of this writing, real-world deployments of Interclouds are still scarce so there are no commonly approved architectural patterns for automatic resource migration, access control and data protection across provider boundaries. Some of these challenges are related to those in pervasive systems, so some

[1]`http://www.oasis-open.org/`
[2]`http://wso2.com/`

results of this thesis might even help to solve challenges in Intercloud scenarios.

Pervasive systems Pervasive Systems aim to overcome the restrictions of closely coupled distributed systems and take into account the requirements arising from context-aware and event-driven applications. Their main characteristic is a loose coupling of components in the sense that connections between components can be *anonymous*, *dynamic*, and *asynchronous*.

- Anonymous means that a service consumer does not have to refer to a service provider by a fixed identifier, but can rather discover and invoke some appropriate service which provides the required capabilities, without knowing anything about the specific implementation of the service. Vice versa, service providers can announce data in the form of events over publish/subscribe mechanisms. This type of communication is usually also supported by Enterprise Service Buses and is sometimes referred to as the *whiteboard pattern*: providers put their data onto a "whiteboard", from where it can be picked up by consumers [127].
- Dynamic means that the coupling between components can change at run time and happens transparently to the components. So, it is possible that while a consumer is connected to some service provider, the provider component gets exchanged by different one which is likewise suited, without the consumer noticing it. Such a dynamic coupling allows for transparently switching to any service which is able to deliver the "best" quality of service, depending on the requirements of the current situation, or to update components at run time, for example. A typical design pattern here is the *connector pattern*, in which a "connector" component serves as the wiring between service provider and consumer and is responsible for connecting the right components to each other.
- Asynchronous coupling refers to application level multicasting by event-based and store-and-forward-oriented communication which allows a timely decoupling of service providers and consumers. Data from service providers may be temporarily buffered and delivered as soon as matching consumers appear in the system.

  This loose coupling of components requires obviously for some way to add descriptions to services by which they can be found and integrated into an application. Addressing components in a straightforward way by using URLs or any other static identifier is not possible in pervasive system — or at least it cannot be the only way of referencing components. Instead, a pervasive system middleware must provide means to add meta-information to services, as well as discovery mechanisms which allow service consumers to find appropriate service providers based on the type of required information, attributes like the service's location, or different quality of service criteria. Often, such meta-information is either provided in form of semantic annotations which are attached to an existing service description (e.g. by SAWSDL [54], as used by the LinkSmart middleware [7]) or by external service description knowledge bases (e.g., WSMO [146, 185], or OWL-S [187], as used in the universAAL middleware [167]). Other pervasive systems rely on a simple syntactical matching of consumer's requirements and provider's capabilities.

Cyber-Physical Systems A currently ongoing evolution of pervasive systems are Cyber-Physical Systems (CPS). The term CPS denotes the interconnection of complex distributed systems and is thus often referred to as a "system of systems". Cyber-Physical systems aim at a strong integration of physical objects with computational resources and can thus be understood as an evolution which is based on pervasive systems, sensor networks, and applications from the Ambient Intelligence area. Although CPS does not strictly refer to any specific architectural patterns, it is related to pervasive systems in the notion of multiple collaborating domains, adaptivity, and context-awareness. In contrast to pervasive systems however, CPS refer to even larger scaled architectures and an integration of various application areas like Smart Grids, eHealth, mobility, etc.

As the area of CPS is still evolving and no clear-cut architectures and protocols have been agreed on so far, it would be difficult to design a policy framework specifically for CPS. Nevertheless, as CPS reuse most concepts of pervasive systems, the policy framework developed in this thesis will very likely be applicable to CPS as well, as we will discuss in the concluding remarks in section 8.3.2.2.

2.1.2. Characteristics of pervasive systems applications

Applications running on top of a pervasive systems middleware are often subsumed by the term "intelligent environment" and refer to use cases like smart homes, or ambient assisted living, which are characterised by context-awareness, ad hoc service orchestration, and heterogeneous platforms.
We will first clarify the terminology around context-awareness and then look at the security challenges arising from pervasive systems applications.

2.1.2.1. Context-awareness vs. situation-awareness

Context-awareness is an often-used, though unclear term, especially when it is confused with similar terms like "situation-awareness". One of the most important definitions of context-awareness is the following, given by Dey and Abowd*:*

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves. [43, 42]*

On the one hand, this definition is in line with the common understanding of context and explicitly states that context is not only information about the system itself, but also about any physical object or place that is relevant to it. On the other hand, however, the definition is also very broad and it would be hard to identify information which cannot be considered as context according to it. It is thus more helpful to refer to Dey's and Abowd's understanding of a *context-aware application*, which is a bit more narrow. According to them, an application is context-aware, if:

> *it uses context to provide relevant information and/or services to the user, where relevance depends on the user's task. [43]*

So, just managing context information does not make an application context-aware. Rather, it should use this information in order to adapts its services and outputs to the user's current needs.

As opposed to context-awareness, the term situation-awareness originally stems from the research area of psychology and cognition and has only later been applied to computer science. It has initially been coined by Endsley who has done significant research on situation-awareness:

> *Situation-Awareness is the perception of elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future. [52]*

The difference to context-awareness is here that a situation is bounded in time and space and that situation-awareness aims at a greater understanding of the situation-specific information: it does not only strive for a "comprehension of their meaning", but even for a projection of the "status in the near future" — a goal that is lacking in the definition of context-awareness.
Moschgath [111] tries to differentiate between the terms *context*, *situation*, and *environment*, in the context of ubiquitous computing and states that they vary only in their perspective: context focuses at actions and decisions, situation focuses at the surroundings and its implication to actions and decision, while environment focuses at the surroundings only.

> *Bei dem Begriff „Kontext" steht der betrachtete Text, die Handlung bzw. die Entscheidung im Mittelpunkt, beim Begriff „Situation" die Umgebung, welche jedoch Einfluss auf das Kernstück Text/die Handlung/die Entscheidung nimmt. Die Übersetzung des englischen Begriffs „Environment" betrachtet nur die Umgebung und schließt damit das Kernstück aus, während der in der Kunst anzutreffende eingedeutschte Begriff explizit den Betrachtenden wieder mit einbezieht.* [111]

However, according to Moschgath, this differentiation is not relevant in the field of ubiquitous computing and thus, the terms could be used interchangeably:

> *Da der Betrachtungswinkel bei der Verwendung der Begriffe „Kontext", „Situation" und „Environment" im Ubiquitous Computing in den wenigsten Fällen eine Rolle spielt und die Unterschiede in der Bedeutung marginal sind, können diese Begriffe synonym verwendet werden.* [111]

Another differentiation between context and situation is made in the complex event processing community, which determines a situation as some generic event pattern which can be instantiated by a series of specific events, for example the pattern *command to door X, error from door X*[3]. These situational patterns would however only be relevant in a specific context, e.g. between 9 and 5 o'clock. So, according to this understanding, situation and context are defined independently of each other and a situation could be detected in various contexts.

In this thesis, we will use both terms *context* and *situation*, where we will base our understanding of context on the definition by Dey and Abowd above. So, *context* is a set of attribute values which describe an entity, whereas the entity can be part of the application or the underlying middleware, but can also refer to some physical object or place, as long as it is of relevance to the application. A *situation*, in contrast, is bounded in time and describes a state — either of the system or again of some physical entity. So, in contrast to a context, a situation has a defined start and endpoint in time.

2.1.2.2. Security challenges in pervasive systems applications

Context-awareness, ad hoc service orchestration, and heterogeneous platforms of pervasive systems applications pose certain challenges when it comes to controlling the security of such applications. While a pervasive systems middleware might provide mechanisms for authorisation, confidentiality, non-repudiation, integrity, and authenticity, it cannot determine how and when to apply them, as it lacks information about the security requirements of the application. At the application layer, however, the services which will be available at run time and the devices on which they are hosted are not known in advance. As a result, it is not possible to hard-code security mechanisms neither into the services themselves,

[3] *http://www.eventprocessing.eu/wiki/index.php?title=Situation_vs._Context*

nor into the application layer. The heterogeneity of platforms further makes it difficult to decide on an appropriate trade-off between security and performance already at design time. So, one challenge pervasive system applications are facing is that the application layer must be able to communicate non-functional requirements to the middleware layer and that the middleware must be able to choose and dynamically apply suited mechanisms in the system architecture at run time.
While policies as a way to specify non-functional requirements have been in use for decades, one special challenge in pervasive systems is that the choice of a suitable policy model depends on the application: some applications might only require a traditional access control scheme, while others might require for some context-specific application of protection mechanisms, taking the specifics of the underlying heterogeneous platforms into account. So, the challenge here is that a policy framework should not only foster a single policy model but should allow applications to register their own models and apply them to the services managed by the pervasive system middleware.

The dynamic of pervasive systems adds another challenge to that: in static system architectures, security policies are often directly attached to specific services and thus only reflect the requirements of the service provider. An example is the WS-Policy standard which allows to add non-functional requirements to a WSDL service description. In a pervasive system however, services might be replaced at run time while the requirements on them remain the same. The challenge is thus to bind non-functional requirements only to a specific class of services or users, without relying on a specific endpoint or identity.

So, the additional degrees of freedom of pervasive systems come at the price of more complex mechanisms for managing security mechanisms according to high-level policies. It is the aim of this thesis to tackle this problem and to make a contribution in form a generic and semantic-based policy framework that provides the means to deal with these challenges.

2.2. Semantic web technologies

Semantic Web Technologies (SWT) are another major building block of this thesis. The formal structure of the policy models introduced in chapters 5 et seqq. is based on Description Logics and the software architecture of the policy framework includes components for ontology parsing and reasoning. In this section, we will therefore give an overview of the most important concepts in the context of SWT, and give references to relevant literature, respectively.

It should be noted that some terms are used interchangeably: in the field of Description Logic, one refers to *concepts*, *individuals*, and *roles* or *relations*. In the context of SWT and OWL (c.f. 2.2.3), the terms *classes*, *instances*, and *properties* and commonly used. Except for fine subtleties, these terms bear the same meaning, respectively, and are therefore used interchangeably in this thesis. The same applies for the terms *complex concept* and *class expression* — both refer to concepts constructed from logical expressions over atomic concepts, once from the DL perspective, once from the SWT perspective.

2.2.1. Description logics

Description Logics (DL) is a knowledge representation formalism which has been introduced in the eighties and began to gain popularity with the advent of semantic web technology (SWT) [20]. As ontology languages and reasoners from the SWT field are based on Description Logics and the policy framework developed in this thesis aims at a close

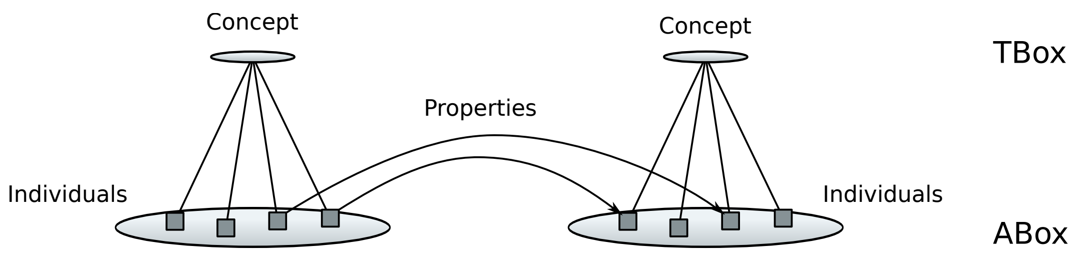


Figure 2.1.: TBox and ABox of a semantic knowledge base

integration of SWT and security policies, the idea of leveraging Description Logics for modelling policies as well as for providing knowledge about the application domain to the policy decision engine is not far fetched. This section will introduce the most important concepts of Description Logics which are required for understanding the rest of this thesis, above all chapter 5 which uses DL to describe the policy model. Readers who already have a reasonable understanding of Description Logics can skip this section and go directly to the discussion of the state of the art in chapter 3.

Description Logics is not a single language but rather a family of decidable logic language dialects which share the same underlying concepts but have different expressivity, depending on the supported *constructors*. Also, slightly varying notations can be found in literature. The work carried out in this thesis adopts most concepts and notations from [12] and [13].
A set of DL statements models a *knowledge base* $KB = \langle \mathcal{T}, \mathcal{A} \rangle$, consisting of a *TBox* $\mathcal{T}$, defining the *terminology* and an *ABox* $\mathcal{A}$, defining *assertions*. The *TBox* comprises *concepts* and *roles*. Concepts represent sets of individuals and are denoted by capital letters. Roles put concepts into relations and are denoted by lower case letters. In the ABox, individuals are defined, assigned to concepts and put into relation with each other using the roles defined in the TBox (c.f. 2.1).
The specific DL family used by a knowledge base is determined by the set of used *constructors*. Every constructor is named by a single letter symbol, so that a specific language dialect can be named by the concatenation of all supported constructors. For instance, Table 2.1 lists the constructors supported by the $\mathcal{SROIQ}(\mathrm{D})$ [74] dialect, which is mainly used in this thesis.

The *ABox* defines *individuals*, assigns them to concepts and defines relationships between them using roles. Formally, it is described by an terminological *interpretation* $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$, consisting of a terminology domain $\Delta^{\mathcal{I}}$ and an interpretation function $(\cdot)^{\mathcal{I}}$, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. With the notion of an interpretation, we can now denote the semantics of DL operators as follows:

$$\begin{array}{rcl}
(\neg A)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & = & \mathcal{C}^{\mathcal{I}} \cap \mathcal{D}^{\mathcal{I}} \\
\forall R.C & = & \left\{a \in \Delta^{\mathcal{I}} \mid \forall b.\,(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\right\} \\
(\exists R.\top)^{\mathcal{I}} & = & \left\{a \in \Delta^{\mathcal{I}} \mid \exists b.\,(a,b) \in R^{\mathcal{I}}\right\} \\
(C \sqsubseteq D)^{\mathcal{I}} & = & C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \\
(C \equiv D)^{\mathcal{I}} & = & (C)^{\mathcal{I}} = (D)^{\mathcal{I}} \\
a : C & = & a^{\mathcal{I}} \in C^{\mathcal{I}} \\
(a,b) : R & = & \left(a^{\mathcal{I}}, b^{\mathcal{I}}\right) \in (R)^{\mathcal{I}}
\end{array}$$

Further and more detailed explanations of the DL semantics of the $\mathcal{SROIQ}(D)$ logic can be found in [112].

Concept Constructors		Name	Symbol
$C \rightarrow$	A	Atomic concept	$\mathcal{AL}$
	$\top$	Universal concept	
	$\bot$	Bottom concept	
	$C \sqcap D$	Intersection	
	$\forall R.C$	Value restriction	
	$\exists R.\top$	Limited existential quantification	
	$\neg C$	Complement	$\mathcal{C}$
	$\{o\}$	Nominals	$\mathcal{O}$
	$\geq n\,R.C$, $\leq n\,R.C$ $= n.R.C$	Qualified number restrictions (Cardinalities)	$\mathcal{Q}$
Role Constructors		**Name**	**Symbol**
$R \circ S \dot{\sqsubseteq} R$ and $S \circ R \dot{\sqsubseteq} R$		Role inclusion	$\mathcal{R}$
R^{-}		Role inverse	$\mathcal{I}$
Datatypes		**Name**	**Symbol**
	$\forall R.d$, $\exists R.d$,	Data type value, data type exists	(D)
		Data types used are n (integer), s (string), f (float)	
		$\mathcal{ALC}$ with transitive roles	$\mathcal{S}$

Table 2.1.: DL constructors

2.2.2. Common reasoning services

Knowledge representations allow to reason over statements in order to derive new facts. So, the support of reasoning and the ability to draw conclusions which are not explicitly stated in the model distinguishes a knowledge base from a database. A number of reasoning engines is available today — most of them based on the tableaux algorithm which allows efficient TBox reasoning. In contrast to these, the KAON2 reasoner[4] maps semantic knowledge bases into a datalog representation which allows to answer ABox queries much faster than the tableaux based variants.

Howsoever the reasoning algorithm is realised, there is a set of standard reasoning services which are provided by most engines and will be used throughout this thesis for the sake of evaluating and analysing policies:

Subsumption checking The axiom $C \sqsubseteq D$ states that D *subsumes* C, i.e. that D is more general than C. This is given if $C^\mathcal{I} \subseteq D^\mathcal{I}$ holds in all interpretations for the chosen terminology. Subsumption checking returns true if C is subsumed by D or returns an explanation if this is not the case.

Satisfiability checking Concept satisfiability is a special instance of subsumption checking where $C \sqsubseteq \top$. It checks if a concept C can be instantiated at all, i.e. if there is an interpretation $\mathcal{I}$ s.t. $C^\mathcal{I} \neq \emptyset$. We then write $\mathcal{I} \models C$.
A knowledge base is called satisfiable if all its concepts are satisfiable.
If the concept (or knowledge base) is satisfiable, this reasoning service returns true. Otherwise it returns the unsatisfiable concept with a respective explanation.

ABox consistency A knowledge base is consistent if facts do not contradict each other, i.e., if there exists an interpretation $\mathcal{I}$ for the chosen terminology s.t. $C^\mathcal{I} \neq \emptyset$ for all concepts $C \in \mathcal{T}$. This reasoning services returns true for a consistent knowledge base or returns explanations for inconsistent facts.

Classification For an individual a, this services returns all class expressions C which describe the individual, i.e. if there is an interpretation $\mathcal{I}$ with $(a)^\mathcal{I} \in (C)^\mathcal{I}$.

Instance checking This reasoning service returns true if a given individual a is an instance of a given class expression C, i.e. if there is an interpretation $\mathcal{I}$ s.t. $(a)^\mathcal{I} \in (C)^\mathcal{I}$.

In contrast to first-order-logic, these reasoning tasks are decidable in all DL dialects. This property makes DL attractive to be used for policy decisions, as a decision process based on reasoning over a DL-based knowledge base is guaranteed to return a valid result, while a FOL-based decision, in contrast to that, could end up in a non-decidable problem. Yet, the worst-case complexity of common reasoning problems over DL is ExpTime or even NexpTime, depending on the language dialect. For a detailed discussion of DL reasoning complexity, we refer to [200].

Most reasoning engines are able to produce so-called *explanations* for non-satisfiable concepts or inconsistent knowledge bases. An explanation is a counter example in form of a sequence of logical deductions derived from the knowledge base, which lead to the detected flaw. In the context of this thesis, explanations will help policy authors to detect inconsistencies in their policy or support them in analysing a policy.

2.2.3. Representation languages

Description Logics is not only a formal way to express knowledge but is also supported by a number of representation languages and reasoning engines, so that it can be easily integrated for practical use. The most common way to write down a DL knowledge base in a machine-processable form is in form of an *ontology*, written in the *Web Ontology Language* (OWL). OWL has been designed as an extension layer to the *Resource Description Framework* (RDF). RDF is a language to represent facts in the form of so-called triplets,

[4] `http://kaon2.semanticweb.org/`

which often relate to the structure *subject-predicate-object* (e.g., `:john :hasAccessTo :crm`), and is widely used nowadays. While RDF is based on a formal graph-based semantics[5] and efficient reasoning engines like Sesame[6] are available, RDF is in many cases only used as a way to represent structured information by triplets, without making use of its semantics, for example to represent meta-information about software components, to serialise news feeds, or in desktop search engines.
OWL adds further semantics to RDF so that it becomes possible to express Description Logic models with OWL. While in the beginning, OWL comprised the three increasingly expressive dialects OWL-Lite, OWL-DL, and OWL-Full, of which OWL-DL related to the $\mathcal{SHOIQ}(D)$ logic and OWL-Full was not decidable in all cases, the current version OWL 2 fully complies with the semantics of the $\mathcal{SROIQ}(D)$ logic [112]. So, denoting a knowledge base in OWL 2 is the most straightforward way to ensure it complies with the underlying formal semantics and in fact, OWL is the standard way of representing ontologies. However, the XML-based syntax of OWL has been designed to be processed by machines and is hardly readable for humans. Especially for large knowledge bases, directly writing OWL syntax is overly complex and cumbersome. On the one hand, numerous tools for graphically modelling of ontologies have been created and are practically used by ontology authors, such as Protegé[7] or Topbraid Composer[8]. However, on the other hand, experienced users often prefer textual representations over graphical models as they are more dense and faster to write. As a consequence, various representations of writing ontologies in a more human-friendly way have been proposed and nowadays, each of them seemed to have found its nice where it can play out its strengths:

The *OWL functional-style syntax* (OWL FS)[9] is part of the OWL 2 specification and is fully compatible with the original XML-based representation of OWL, i.e. each OWL/XML document can be converted to OWL FS and vice versa without any loss of semantics. While it is easier to read for humans, it is still very verbose and only of limited use for developers who need to rapidly write ontologies. The following snippets illustrate the same statements in OWL/XML and in OWL FS:

```
<ClassAssertion>
  <Class URI="&ex;Scientist"/>
  <Individual URI="&ex;Alice"/>
</ClassAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty URI="&ex;collaboratesWith"/>
  <Individual URI="&ex;Alice"/>
  <Individual URI="&ex;John"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty URI="&ex;collaboratesWith"/>
  <Individual URI="&ex;Alice"/>
  <Individual URI="&ex;Bob"/>
</ObjectPropertyAssertion>
```

```
ClassAssertion(a:Scientist :alice)
ObjectPropertyAssertion(a:hasPermission  :alice           :reportPermission)
ObjectPropertyAssertion(a:hasAction      :reportPermission :read)
ObjectPropertyAssertion(a:hasResource    :reportPermission :dailyReport)
```

[5] `http://www.w3.org/TR/rdf-mt/`
[6] `http://www.openrdf.org/`
[7] `http://protege.stanford.edu/`
[8] `www.topbraidcomposer.com/`
[9] `http://www.w3.org/TR/owl2-syntax/#Functional-Style_Syntax`

The *Manchester syntax*[10] [73] aims to provide an even better readability than OWL FS by being more compact and frame-based (as opposed to the axiom-based OWL FS). It covers large parts of the OWL 2 semantics, but in contrast to OWL FS, there are some ontologies which can not be translated into Manchester syntax without loss of semantics. The following snippet illustrates how the same ontology fragment from before looks when written in Manchester syntax. As can be seen, the language is much better suited for users who need to need to read or modify large ontologies, as opposed to OWL/XML.

```
Individual: reportPermission
     Types: Permission
     Facts: hasAction    read,
            hasResource  dailyReport
```

Most interesting about the Manchester syntax is however its variant for writing class expressions, *Manchester DL*[11]. This syntax allows to write class expressions ("complex concepts", in DL speak) in almost natural language and can be used either as a query language or in order to define new (complex) concepts from ontology fragments, as shown in the following example:

```
User THAT hasPermission SOME Permission THAT hasAction VALUE {read} AND
  hasResource {dailyReport}
```

The *Terse RDF Triple Language* ("Turtle")[12] is a language for writing RDF triplets in a compact and readable format and can be used within the SPARQL query language to refer to triplet patterns. In large parts, it is similar to Notation3[13]. A valid Turtle document starts with a prefix definition block to specify namespaces for the document and then refers to triplets in the form `p:subject p:predicate p:object`. For users who need to write a large ABox by hand, Turtle is very attractive due to its grouping and nesting features that allow to specify multiple properties for a single individual at once and to specify objects anonymously, i.e. "inline", as in the following example:

```
@prefix : <http://linkality.org/apollon/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
:Alice rdf:type :Scientist ;
          :hasPermission [
                         :hasAction :read;
                :hasResource :dailyReport ] .
```

Besides these ontology representation languages, a zoo of query languages exist: *SPARQL*[14] is the most prominent and widest used query language, whereas other languages like *Terp*[15] or *SerQL*[16] are supported only by specific reasoning engines and mainly aim at facilitating the fairly complex SPARQL notation for non-trivial queries. As SPARQL is RDF based and thus follows a different semantics than OWL-DL, DL based alternatives like *DL Query*[17] and *SPARQL-DL*[18] have been proposed and are available for productive use. Further, as OWL does not support rules, but Description Logics itself can be combined with so-called "DL-safe rules", i.e. rules which do not go beyond the expressivity of DL,

[10] `http://www.w3.org/2007/OWL/wiki/ManchesterSyntax`
[11] `http://www.co-ode.org/resources/reference/manchester_syntax/`
[12] `http://www.w3.org/TeamSubmission/turtle/`
[13] `http://www.w3.org/DesignIssues/Notation3`
[14] `http://www.w3.org/TR/rdf-sparql-query/`
[15] `http://weblog.clarkparsia.com/2010/04/01/pellet21-terp/`
[16] `http://www.openrdf.org/doc/sesame/users/ch06.html`
[17] `http://protegewiki.stanford.edu/wiki/DLQueryTab`
[18] `http://www.derivo.de/en/resources/sparql-dl-api.html`

different languages are required when rules need to be added to an ontology. While N3 provides some means to write rules [19], the most commonly used language today is SWRL[19], a combination of the former RuleML and OWL, which can be directly integrated in OWL/XML documents.

So far, OWL/XML in combination with SWRL has established as the standard way of writing machine-interpretable ontologies, whereas a variety of different languages is used when it comes to human-readable representations or queries. The prototypes developed in this thesis thus work on OWL/XML, while most manual modifications are made in Turtle. In this document, we will mostly use Manchester DL- or Turtle-style pseudo code to denote ontology fragments.

[19]`http://www.w3.org/Submission/SWRL/`

CHAPTER 3

State of the art in security policy frameworks

Policies and policy frameworks are not a new research area and it is thus no surprise that a number of such systems already exists. This section reviews the most important policy frameworks, discusses to which extend they are suited to be applied to pervasive systems, and identifies the gap between the current state of the art and the policy framework to be developed in this thesis.

We will begin with a review of some standard concepts upon which most policy-based systems rely, and then continue with a discussion of existing frameworks which have their strengths in at least one of the three main aspects this thesis will contribute to: semantic representations, extensibility of the policy model, and integration into existing systems.

3.1. Standardisation efforts

Policies in various styles have been in use for a long time and a number of standards has been created around them. We review here only the most important standards on generic policy concepts which coined the key terms in this field — although implementations of these standards themselves are rarely found in practice. Various additional standards for specific policy languages like XACML [123], WS-Policy [188], and WS-SecurityPolicy [113] exist, but will not be discussed here. We will rather sketch their concepts as needed in the subsequent chapters.

The Common Information Model (CIM) [44] is a standard for management interfaces of distributed systems and defines data structure for describing interfaces and exchanging information in such systems. In addition, a policy-specific extension, the Policy Core Information Model PCIM [110, 109, 47], has been created. Both, CIM and the PCIM extensions are kept as generic as possible and are independent from any vendor, platform, protocol, or implementation. In fact, the authors of PCIM leave application-specific models up to possible future extensions of PCIM and claim the concepts of the PCIM model itself as "*[...] sufficiently generic to allow them to represent policies related to anything. However, it is expected that their initial application in the IETF will be for representing policies related to QoS (DiffServ and IntServ) and to IPSec*" [110]. Also, PCIM does not recommend any language representation of the model. Apart from an exemplary mapping to store the PCIM model

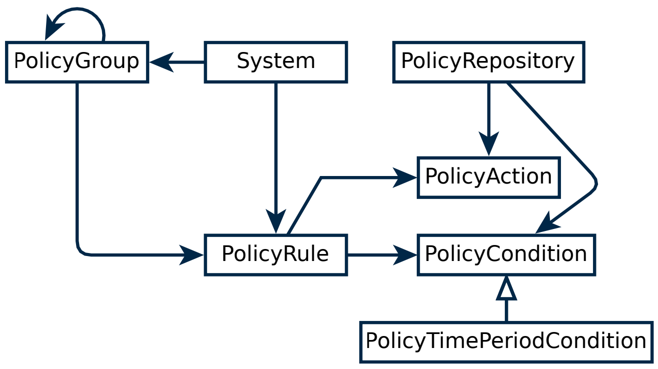


Figure 3.1.: PCIM policy model

in an LDAP directory service, it is left up to the implementer to decide on an appropriate representation of the model. The policy structure defined by PCIM is roughly depicted by Figure 3.1. Most notably, a rule consists of a *PolicyCondition* and a *PolicyAction* part. However, PCIM does not declare when such rules shall be evaluated and leaves it to the implementer to trigger the evaluation when appropriate. So, as opposed to the commonly used Event-Condition-Action (ECA) pattern of reactive policies, PCIM does not consider the event part but rather concentrates on conditions and actions only.
While CIM and PCIM are the only relevant standards defining abstract policy models, practical implementations are scarce. Reasons for this might on the one hand be the level of abstraction, which puts high demands on implementers who have to design appropriate data structures, representations, serialisations, etc. in order to make PCIM usable. On the other hand, PCIM is not well suited for some policies and the expressivity it provides might not outweigh the effort of implementing it. While some research has been done on the basis of PCIM (e.g., [104]), nowadays most policy frameworks use some basic concepts of CIM/PCIM but do not fully comply with the model.

The Common Open Policy Service (COPS) [26] is a protocol for exchanging policy information between a Policy Decision Point (PDP) and its clients, called Policy Enforcement Points (PEP). It has originally been specified to outsource policy decisions to a PDP in the context of RSVP[1], a protocol for router configurations. While this specific use case is not of high relevance in this thesis, COPS is still worth mentioning, as it is a frequently cited protocol which makes use of the architectural components PEP and PDP, which have originally been specified in [196]. An infrastructure of PDP, PEP, Policy Information Points (PIP), and policy repository components has become widely accepted and is used by most frameworks nowadays. Apart from the *outsourcing* interaction scheme, where PEPs request the PDP for a policy decision, COPS-PR [30] defines a *provisioning* or *configuration* model, where the PDP pushed configurations to the PEPs, thereby avoiding the need for online requests for every policy decision. This interaction scheme is however not suited for policies in dynamic environments, where each policy decision requires the evaluation of different information source, and therefore the majority of pervasive systems sticks to the outsourcing scheme.

[1]RFC 2205

3.2. Frameworks with a focus on semantics

At first, related work on frameworks using logic representations of policies will be reviewed. Logic representations, especially semantic web technology (SWT), might be well suited to support users in analysing [182] and writing [174] high-level policies, so the most relevant contributions to this aspect will be discussed in this section.

3.2.1. Protune

Protune [142, 156, 116] is an extensible framework for access control based on *trust negotiations* [195] in the Semantic Web and has been developed as a joint work by L3S Research Center, Hanover and Naples University within the REWERSE [2] project. Trust negotiation denotes the process "iteratively disclosing certified credentials" [21] executing actions as required by the communication partner (e.g., information like certain certificate attributes or actions like registering at a web site) with the goal of deciding an access request. In Protune, policies are written as a Logic Program on top of Prolog and can refer to instances from external ontologies for describing *provisional actions*, i.e. actions which must be executed as part of a trust negotiation. Metarules are used for specifying which actor is obliged to execute an action (`actor:self` or `actor:peer`), as well as for declaring if predicates may be revealed to the peer (`sensitivity:private` or `sensitivity:public`). The special predicate `in` further allows to refer to external functions for evaluating conditions, such as an SQL statement being tested against an external RDBMS, whose functions are bundled in an external package and can be referred to as `rdbms:query("<SQL Statement")`, for instance.
Besides the evaluation and enforcement of policies, Protune also supports policy explanations to inform users about the reasons why an access request has been denied or which information would be required in order to grant it ("how-to", "why/why-not" and "what-if" queries). These explanations are basically Prolog proofs, where predicates like "access(X,Y)" can be annotated with user-friendly explanations like "access by Y to resource X". The following simple policy is given as an example in the Protune documentation[2] and states that the actor is willing to grant access to a resource, if the peer (the other actor) has sent a declaration (identified by the `#Declaration` action in the ontology) containing one of the matching username (`Uid`) / password (`Pwd`) pairs.

```
execute(access(resource)) :- declaration(Uid, Pwd), password(Uid, Pwd).
password(uid1, pwd1).
password(uid2, pwd2).

access(_)->type:provisional.
access(_)->ontology:<www.L3S.de/policyFramework#Access>.
access(_)->actor:self.

declaration(_, _)->type:provisional.
declaration(_, _)->ontology:<www.L3S.de/policyFramework#Declaration>.
declaration(_, _)->actor:peer.

password(_, _)->type:logical.
password(Uid, Pwd)->sensitivity:public :- ground(Uid), ground(Pwd).
```

The approach of Protune is closely related to PeerTrust [115], a Logic Programming based trust negotiation language proposed earlier by some of the authors of Protune and in fact, the PeerTrust engine has been can be used in the Protune architecture instead of the original Protune engine.

[2]http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/?pagename=Protune+in+a+nutshell

Discussion

Protune is one of the most comprehensive policy frameworks for trust negotiations that are currently available. In general, the fact that Protune is moving away from static identifiers for entities, and rather describes entities by means of attributes allows it to refer to beforehand unknown entities – an important requirement in pervasive systems. Although it does not use specific semantic web technology, it can be considered as a "semantic" framework, as it models policies in a Prolog-based knowledge base and provides means to link attributes to external knowledge bases. Also, the possibility to extend Protune by external packages for evaluating conditions or executing actions should be looked upon favourably and will be adopted for the developments in this thesis. Further, although no in-depth evaluation of Protune's usability are available, the authors took care of a facilitated policy specification by providing of an explanation mechanism.

A limitation of Protune is that it only built upon the model of trust negotiation, i.e., it assumes the negotiation scheme described in [34] to be executed before an access request is decided. Thus, the framework does not include event policies which are required for reactive systems and in order to trigger reconfigurations of system components. Further, Protune assumes a negotiation only between client and server and does not consider multiple policy domains whose decisions need to be aligned.

3.2.2. Rein

Rei, whose development started as part of the Ph.D. thesis of L. Kagal, is a policy language for the semantic web [85, 83]. It uses a combination of OWL [106] and Rei-specific enhancements [84] in order to allow using variables in rules[3]. Rei further supports advanced concepts that are not found in many of its competitors, like obligations, metapolicies for conflict handling and "speech acts" (e.g. the concept of policy delegation). In addition, Rei comes with a Prolog-based reasoning engine that can be used for analysis policies and deciding access requests.
Rein (Rei + N3) [97, 87, 86] is a semantic web policy framework based on Rei and the RDF-based rule language N3[4]. In contrast to Rei, it uses an engine based on the forward-chaining reasoner *cwm* to make policy decisions and claims to be independent from any specific policy language (as long as the used policy can be represented using the Rei ontology). The main focus of Rein is on representing and processing policies using semantic web technologies in order to improve cross-domain interoperability of policies.

Discussion

An advantage of Rei is that it integrates advanced policy concepts like delegation and revocation by the means of "metapolicies". The approach of applying semantic web technologies like ontologies and reasoners for representing policies is also very promising. Especially in decentralised environments with a-priori unknown entities, ontologies allow to establish a common vocabulary and platform independent policy layer. In this sense, the idea of Rein could theoretically also be adapted to pervasive environments in which similar requirements apply. Yet, Rein still has some limitations and is not fully suited for pervasive environments:

1. Rein is only focused on access control and thus specifying general security requirements for devices and networks is not possible

[3]Variables are not supported by OWL itself
[4]http://www.w3.org/2004/12/rules-ws/paper/94/

2. Reactive policies, triggered by external events, are not supported by Rein.
3. Given two policies, the Rein engine can detect possible conflicts and resolve them using metapolicies. However, the resolution of conflicts across domain boundaries is not addressed in Rein.

3.2.3. KAoS

KAoS [178, 179], mainly developed by J. Bradshaw at IHMC, Florida, is a semantic policy management framework including a policy language, an engine for policy decisions, enforcement points and a graphical tool for assisting the user in policy specification. KAoS has originally been designed to be used in agent-based networks but has also been applied to semantic web applications. Just as in Rein, policies in KAoS refer to a subject, a resource and an action and specify positive and negative authorisations as well as obligations. However, the policy representation in KAoS is somewhat different from that in Rein: KAoS represents rules as instances in an ontology, i.e. even the policy itself is represented in description logic (i.e., in DAML, the antecedent of OWL). As description logic alone does not support the definition of rules and variables, KAoS additionally uses extensions to the Java Theorem Prover[5] (JTP) to implement a reasoning engine that is used for policy decisions. Yet, using DAML for representing policies results in verbose policy representations as can be seen in this example of a very simple KAoS policy:

```
<daml:Class rdf:ID=ExampleAction">
<rdfs:subClassOf rdf:resource="#EncryptedCommunicationAction" />
   <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#performedBy" />
      <daml:toClass rdf:resource="#MembersOfDomainA" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasDestination" />
      <daml:toClass rdf:resource="#notMembersOfDomainA " />
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
<policy:PosAuthorizationPolicy rdf:ID="Example">
  <policy:controls rdf:resource="#ExampleAction" />
  <policy:hasSiteOfEnforcement rdf:resource="#ActorSite" />
  <policy:hasPriority>10</policy:hasPriority>
  <policy:hasUpdateTimeStamp>4237445645589</policy:hasUpdateTimeStamp>
</policy:NegAuthorizationPolicy>
```

Discussion

KAoS represents policies in description logic, i.e. as ontologies. The advantage of this approach is that reasoning over policies becomes possible and can be used to create user-friendly modelling tools with support for explanations. In addition, existing reasoning engines can be used to check for possible policy conflicts or evaluate access requests.
However, writing policies purely in Description Logic limits the expressiveness of the language. Although the authors of KAoS have partially worked around the problem by integrating JTP, e.g. in order to support variables, KAoS cannot use external mechanisms

[5]http://ksl.stanford.edu/software/JTP/

for evaluating conditions. Further, KAoS is limited to access control decisions and does not thoroughly cover the issue of cross-domain conflicts.

3.2.4. SicAri

The SicAri project[6] focuses, among other aspects, on a security policy framework for pervasive computing environments. The project adopts the notion of *policy patterns* to derive specific rules from high-level protection goals made by the policy administrator [161, 160]. The patterns have been formalised and are represented in knowledge bases which act as support for the user during the policy specification process. After the user has specified the security requirements based on these predefined patterns, a policy generator transforms the high-level specifications into an XACML policy. This policy can then be deployed to the application [145]. Additionally, SicAri takes workflows into account: An approach based on temporal logic is proposed to find the appropriate position in a workflow where a pattern should be applied [144].

Discussion

The most important results of SicAri are in the field of *policy refinement* where SicAri aims at reducing the complexity of policy specification. Users are provided with "best practises" in the form of policy patterns for translating high-level protection goals to more specific, enforceable rules. Thereby, policy specification shall be possible even for non-experts and common pitfalls shall be avoided. Yet, SicAri's approach requires that workflows and entities are known at design time in order to apply the policy patterns and translate them into XACML rules. Dynamic environments in which devices join and leave a network and policies have to be changed at runtime are not considered by SicAri. For policy representation, XACML is used and thus policies in SicAri are limited to the access-control capabilities provided by XACML. Further, the architecture of SicAri's policy framework follows the traditional centralised IETF COPS [26] design and does not take multiple domains into account. So, as cross-domain conflicts and dynamically changing policies will be the rule in pervasive environments rather than the exception, these are still open areas that have not fully been addressed by SicAri.

3.3. Frameworks with a focus on scalability and extensibility

The second relevant aspect that will be reviewed in this chapter is scalability and extensibility of a policy model. It is a main characteristic of pervasive systems that various applications with different security requirements can be set up on top of a middleware layer and that therefore, the middleware needs to support multiple policy models in order to support the applications. In this section, related work with the aim to provide scalable and extensible policy models will be discussed.

3.3.1. Cassandra

Cassandra [15] is a system for role-based trust management [24] and access control in large-scale distributed systems. It features dynamic role assignment based on predicates which are provided as attributes in certificates. Using a trust negotiation approach, similar to PeerTrust [115], the Cassandra system retrieves the attributes required to evaluate a

[6]http://www.sicari.de/

policy from remote domains, decides whether a certain role may be assigned to a user, and applies the access rights of that role. For instance, if a predicate *canActivate* (*Alice*, *Students*) can be deduced from the policy, Cassandra allows user *Alice* to activate the *Students* role.

The policy language of Cassandra is based on a Datalog representation and can be extended by a *constraint domain* of first order formulae. This allows to scale the expressiveness, and with that, the computational complexity, of the language in order to adopt the Cassandra language to the requirements of an application. Respectively, the language can become undecidable if the constraint domain is not chosen restrictive enough. With only a few predicates, Cassandra supports writing RBAC policies with dynamic role activations, delegation of roles and enforcement of separation of duty constraints. For example, the fact that administrators are allowed to delegate their `Adm` role, restricting the length of the delegation chain to *n*, can be written as follows:

```
canActivate(x, DelegateAdm(y,n)) ← hasActivated(x, Adm(z,n))
canActivate(y, Adm(x, n)) ← hasActivated(x,DelegateAdm(y, n)), 0 ≤ n < n
```

Discussion

Interesting about Cassandra is its well-defined semantics based on a Datalog representation and its scalable expressiveness. The authors of Cassandra have recognised that applications have different requirements on the complexity of a policy language and that instead of providing the most expressive languages, it makes sense to allow extensions of the language, so that users can choose the right trade-off between computational complexity and the requirements of their application.

While Cassandra might be suited for some pervasive systems-based applications, it features a single policy model for access control. For example, there is no possibility of reusing Cassandra's predicates in other models, such as for usage control or self-management policies. Further, certificate attributes have to correspond to policy predicates and reasoning about other external information sources is not part of Cassandra's concept.

3.3.2. OPL (ORKA Policy Language)

OPL is a policy language developed as part of the ORKA project and has been motivated by the need for RBAC-based access control patterns in workflow-oriented architectures. Existing policy languages were reviewed during the project and found to provide insufficient support for patterns like role hierarchies, delegation, or separation of duty. Also, the lack of a formalisation of commercial policy languages like XACML and WS-Policy was considered a drawback. The ORKA Policy Language features a number of modules, each providing a certain policy pattern which enriches the RBAC model by additional functionality. Modules can depend on each other so that it is possible to build more complex access control patterns on the basis of existing ones. As the semantic of each module has been specified in Z-Object, OPL policies rest upon a comprehensive formalism.

Discussion

Positive about OPL in the context of pervasive systems is the extensibility of the language's expressiveness by means of additional modules, as it supports scalability and allows to add further high-level patterns which are easier to write and maintain than large sets of individual subject-resource-condition rules (like in XACML). However, OPL focuses on role-based access control and does neither take into account arbitrary attribute-based policies, nor semantic information about services and security properties as it is available in the scenarios considered in this thesis. Therefore, while the modularisation paradigm could be applied to the policy framework developed in this thesis, further research on the integration of semantic information and a more general policy model is still required.

3.3.3. XACML

The eXtensible Access Control Markup Language (XACML) is a policy language and architecture specification for access control. In contrast to the other languages discussed in this chapter, it receives comprehensive support by the industry in form of the XACML OASIS Technical Committee. XACML has been released as an OASIS standard and a number of open source implementations is available. The access control model featured by XACML is attribute based (ABAC), that is, any kind of attribute can be used to describe the subject which has initiated an access request, the resource which is to be accessed and the action which is requested to be taken on the resource. Conditions over these attributes can be formulated using a set of predefined XACML functions. Which types of attributes have to be understood by the PDP is not determined by XACML – this decision is left to the implementer of a specific PDP. Yet, as XACML mainly targets at web service based infrastructures, it provides support for using attributes provided by SAML assertions in form of a SAML *profile* [122]. XACML profiles are extensions of the core specification and define attributes and functions for specific use cases of XACML, such as role-based access control (RBAC) [119], privacy extensions [121] or health-care-specific extensions. Profiles make use of the various extension points of XACML. For example, XACML allows to add custom implementations of attributes and functions, as well as custom attribute resolvers, which are responsible for retrieving the value for an attribute so it can be evaluated against the values contained in the access request.

These extension points provide a great degree of flexibility so that XACML can be adapted to many different access control scenarios. Nevertheless, there are some drawbacks when using XACML in pervasive systems:
at first, XACML only focuses on access control only. That is, its design assumes that a policy decision can only be triggered by an access request and as a result of the decision, the access is either granted or refused. Reactive policies are not supported by XACML. Obligations can be specified as part of the decision, however XACML only states that obligations have to be executed "in conjunction with the authorization decision" [120, line 1806] and thus treats obligations rather as *provisions*, which have to be fulfilled in order for an access to be granted. Usage control obligations, in terms of Pretschner's definition of "requirements on the future" [135] are not covered by XACML. Also, although the XACML specification is not very specific about the usage of PEPs, it is not intended by the specification that PEPs are used at the client side, so it would not be possible to prevent clients from sending out sensitive information or to amend an outgoing access request with descriptive meta data.
Secondly, the XACML syntax is quite verbose and due to the fine-granular attribute based model, the number of rules quickly grows with the number of entities in a system and often results in complex policies which are hard to maintain by the user. Especially the fact that the actual *security model* cannot be expressed in XACML makes it difficult for users to understand and maintain a policy. Writing an XACML policy requires authors to transform the overall security model they have in mind into a set of specific rules, which can then be written in XACML. However, there is no link between the high-level requirements stated in the security model and the low-level policies expressed in XACML. As a consequence, there is no way of checking whether the low-level rules actually enforce the high-level requirements and worse, any change in the rules is likely to break the overall security

model, as there is no mechanism to ensure their consistency with respect to the model. Thirdly, XACML has not be specified in a formal way. Although a number of formalisations has been provided by different authors with hindsight [92, 91, 90, 28, 80], most of them do not comprise the full specification and are based on the respective author's perception of the informal XACML specification. Tools for static (i.e., at design time) change-impact of XACML exist, such as XACML-DL[7], Margrave [60], or the Prolog-based analysis tool by the author of this thesis[8], but they rely on translating a limited subset of XACML into a logic representation and therefore do not allow a comprehensive analysis based on a formal semantics.

3.4. Frameworks with a focus on easy integration into pervasive systems

Most work on the aforementioned frameworks concentrates at the policy model itself, ways to represent, and to extend it. A practical integration of these frameworks into an exiting distributed system was however not in the focus. However, in order to be applicable in practice, this aspect should not be neglected as it will raise its own challenges. In this section, we therefore review policy frameworks which have been designed with practical integration into an existing system in mind.

3.4.1. Proteus

Proteus [173, 172] is a semantic policy framework that features context-sensitive policies. By modelling OWL classes, users can specify positive and negative authorisation rules, as well as event-action policies (called *obligations* by the authors). Each authorisation rule refers to a so-called active context, which can be modelled in OWL likewise, so that the rule becomes only applicable if the appropriate context has been "activated" before. Contexts are modelled as conjunctions and disjunctions of sets of attribute-value pairs, whereas each attribute must have a predefined semantic and must be evaluable by the Proteus framework (e.g., Proteus must be able to determine what the value of an attribute `Scheduled_Calendar_Slot(?meeting)` is). Users can explicitly declare contexts to be incompatible with each other, which guarantees that they cannot be be activated at the same time by the Proteus framework. As an example, the authors mention contexts referring to different locations of the same user, which can obviously not be valid at the same time. Modelling contexts and obligations, Proteus uses not only Description Logic, but also rules similar to Horn clauses, which are evaluated under the closed-world assumption.

A main feature of Proteus is the adaptation of policies, actions and contexts. By *policy adaptation*, a policy can be kept active although its context is no longer valid. For this purpose, users have to specify additional LP rules which determine under which conditions a policy can be kept active. By *action adaptation*, the selection of alternative actions in case the action which has originally been stated in an event-action policy cannot be executed, for example because an authorisation policies refuses it. In that case, users can define alternative actions, again using LP rules. *Context adaptation*, finally, enables Proteus to autonomously active contexts in which a currently requested action will be permitted. For that purpose, when an action is asked to be executed, Proteus searches for all contexts which do not conflict with the current context and whose activation would lead to a

[7]http://clarkparsia.com/
[8]`http://linkality.org/software/xacml-policy-analyser/`

policy permitting the action. Proteus then activates these contexts, if possible, so that the permitting rule becomes enabled and the action is permitted.

In [171], the authors have further worked towards advanced representations of context data and its quality. They consider the classes *precision*, *freshness* and *correctness* as critical for assessing the quality of context data, and therewith, its trustworthiness when used in security policies.

Discussion

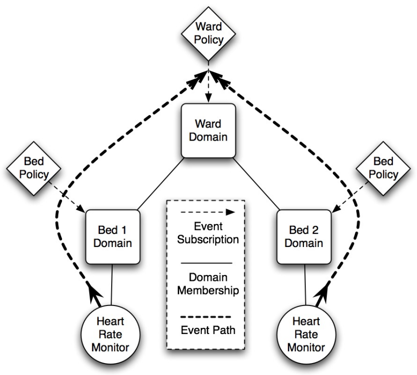
Promising about Proteus is the concept of representing policies on the basis of semantic web technology – just as in Rein and KAoS. In contrast to these two, Proteus further puts its main focus on modelling context and context-sensitive policies, which takes into account the dynamic characteristic of pervasive systems. However, an universal definition of how context should be modelled is not possible. Rather, which model is suitable depends on the application, as stated in [42, 103], and pre-defining a context model within the policy framework, as done by Proteus, could therefore restrict the applicability of the framework.

In addition, Proteus is based on a policy model which may not be suited for all applications. As no mechanisms for extending the expressivity of the policy model are given, the set of use cases in which Proteus is both expressive enough and still usable might be limited. It may further be expected that the usability (in terms of keeping the set of policies maintainable and understandable) of Proteus is decreased by features like adaptation, which autonomously activate and deactivate contexts and policies, trying to find a constellation which permits a certain action. This contradicts the usual deny-default mindset of policy authors, restricting all actions by default and then granting some actions in dedicated contexts. Further, also the issue of resolving conflicts between multiple policies is not covered.

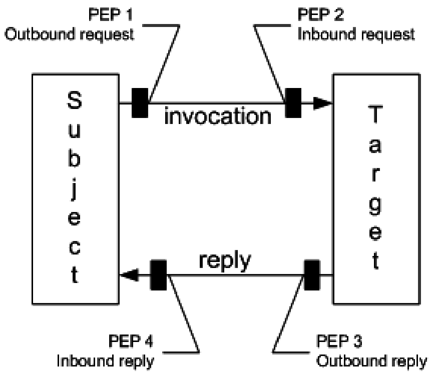
3.4.2. Ponder2

The project *Ponder*2 [4, 176], mainly developed at Imperial College, London by Prof. M. Sloman aims at providing a generic policy framework for controlling so-called "managed objects", hosted within a "self managed cell". Managed objects can either be located within the same Java Virtual Machine or be hosted on remote platforms and connected by protocols like RMI or SOAP. Managed objects are assigned to hierarchically organised domains and communicate with each other over an event bus. The event bus complies to the hierarchical domain concept so that events can only be propagated up the domain hierarchy but not to lower or adjacent domains. This way, the amount of events in a single domain is kept relatively low while it is still possible to catch all events generated in the system by listening to the root domain. Figure 3.2a depicts the hierarchical event model of Ponder2.

Using the *PonderTalk* language, two types of policies can be specified for self managed cells: Authorisation policies and obligation policies. Authorisation policies describe access-control rules for the methods provided by managed objects. An authorisation rule is defined by the subject requesting for access, the action (the method the subject tries to invoke) and the target (the managed object that provides this method). Further, Ponder2's authorisation policies support a parameter "`focus`" for specifying the PEP that shall enforce a policy. There are four possible PEPs for every request: two on the subject's side and two on the target's side, as depicted in Figure 3.2b. So, an exemplary authorisation policy in Ponder2 might look as follows:



(a) Hierarchical event model



(b) Four possible PEPs for each request

Figure 3.2.: Domain and policy concepts in Ponder2

```
(newauthpol subject: root/personnel/nurse/ward1/nurse1
        action: "getrecord"
        target: root/patient/ward1/patient1
        focus:"t").
```

In many cases, contradicting policies will lead to modality conflicts, i.e. one rule would permit an action while another one would deny it. To deal with this situation, Ponder2 provides the following conflict resolution strategies: Usually, more specific rules will be prioritised and overwrite other more generic rules. For cases in which this behaviour is not wanted, Ponder2 further allows to specify global rules that explicitly override all other rules, regardless of their specificity.

Besides authorisation policies, Ponder2 supports obligation policies. These are basically event-condition-action patterns defining actions that shall be executed by managed objects. An example for an obligation policy in Ponder2 is the following rule:

```
policy := root/factory/ecapolicy create.
policy event: root/event/monitor;
condition: [ :value | value > 100 ];
action: [ :monitor :value |
root print: "Monitor " + monitor + " has value " + value
];
active: true.
```

Discussion

Ponder2 is one of the most mature policy frameworks of the related projects presented in this chapter. It is publicly available as open source and is backed by some larger research projects[9]. It is suited for pervasive environments and has successfully been tested in different applications. However, there are still some points in which Ponder2 differs from what is envisioned in this thesis:

1. Ponder2 does not support any semantics. Neither events, nor policy subjects, targets or actions can be described semantically and thus it is not possible to reason over

[9]IST TrustCoM (EU-FP6), IST Emanics (EU-FP6), Allow Project (EU-FP7), EPSRC AMUSE Project

Ponder policies or to include facts inferred from external knowledge bases into Ponder rules.

2. Although authorisation policies in Ponder2 take into account the concept of a "domain", this refers only to the hierarchical domains that are used to organise the managed objects. Ponder2 does not provide any mechanisms for cross-domain conflict resolution between different self-managed cells.

3. Obligation policies in Ponder2 are merely a simple event-condition-action mechanism that triggers managed objects to execute certain actions. As Ponder does not support semantics, the exact data format of events has the be known and referring to any external event processing engine is not possible.

3.4.3. IBM Policy Management Library

The IBM Policy Management Library (PML) is a software library with a focus on policy based management of networked systems. Besides components for policy decision- and enforcement points, PML contains a distributed policy repository, as well some tools for policy analysis and transformation. PML supports authorisation policies based on Access Control Lists (ACL), as well as policies for system constraints, configurations, and Event-Condition-Action rules. They comply to a predetermined, non-formalised model and can either be written in Groovy[10] or in CIM-SPL [45], a simple syntax for the Policy Core Information Model (PCIM [110, 109]) which has also mainly been developed by IBM. Users can use the provided web based management tool to write policies using a template mechanism and apply different analysis tasks. PML claims to support "dominance checks, coverage checks, simultaneous applicability checks and conflict detection / resolution"[11]. Dominance checking tests whether one rule is subsumed by another one, coverage checking tests if sets of input values are covered by a certain rule, and checking for simultaneous applicability tests whether two rules could potentially conflict with each other, as they are triggered by the same input values. These tasks are based on algorithms for solving linear optimisation problems, solution trees for solving boolean expressions, and an "AutoSelect Analyzer" which attempts to apply appropriate algorithms for analysing policies with mixed conditions, containing numeric and boolean variables. Conflict detection and resolution refers to conflicts within a policy, i.e. rules which are simultaneously applicable and might result in conflicting decisions. The authors of PML note that statically detecting conflicts between rules is a difficult problem, as it would require the analysis engine to know the complete semantics of each action. Therefore, they decided to leave it up to the user to write metapolicies in order to declare which rules might end up in a conflict. Simple priority based metapolicies can then be used to prefer one rule over the other in the case of a conflict.
Implementation-wise, PML is based on Java, maintains the policy repository in a Derby or MySQL database, and uses MQTT[12] for policy-controlling network entities. A very simple authorisation policy, granting access based on some attribute value would be written in CIM-SPL as in the following snippet:

[10]Groovy is a scripting language for the Java VM `http://groovy.codehaus.org`
[11]`https://www.ibm.com/developerworks/community/alphaworks/tech/wpml`
[12]`http://mqtt.org/`

```
Strategy Execute_All_Applicable;
Policy  {
        Condition  {
                auth.getClient().getAffiliation() == "US"
        }
        Decision  {
                auth.allow()
        }
}:1;
```

Discussion

PML's design makes it well suited for integration into pervasive systems. Apart from the additions like analysis algorithms, the library is lightweight and can be integrated with only a few lines of code. Its static policy model is likely to show good evaluation performance and addresses some of the needs of pervasive systems, such as controlling access to resources, changing system configurations, and detecting unwanted system states.

However, there are some drawbacks of PML, the most critical one being that it dictates a specific policy model which cannot be extended and which does not support semantics. For example, authorisation policies are based on a simple ACL scheme and users have no option to apply a different model. This can be a severe limitation if users have to apply a different access rights scheme which is not compatible with the one featured by PML – in such a case, PML would not be applicable. Further, PML lacks semantics for variables, which implies that it is not possible to define high-level policies: all rules have to be written at a "technical" level, i.e. they have to refer to variables as they are provided by sensors in the network and thus, PML policies could quickly become hard to read and to maintain. The lack of semantics also makes it difficult to refer to meta information from the underlying middleware like service descriptions. Such data would be handled as mere Strings variables in PML and it would not be possible to reason over it.Policy Frameworks!PML

3.5. Summary and identification of gaps

While all of the above discussed approaches tackle a specific problem quite well, and some (e.g., Ponder) even strive for an overall approach to "policy-enable" pervasive systems, none of them addresses all challenges satisfyingly, as we will point out in the following.

Only few approaches on consider resources which are concurrently under the control of multiple domains and thus, mechanisms for merging and negotiating policy decisions across domain boundaries are scarce. While the Protune framework is built around a negotiation mechanism for example, it is only focused at trust negotiation, and does not support negotiating obligations, for example. Such negotiations are however essential when aiming for a "self-protecting" system, i.e. a pervasive system which is able to automatically reconfigure itself in order to fulfil protection goals and other non-functional requirements.

Some frameworks like Rein and KAoS are based on semantic web technology (SWT) and thereby allow in general to refer to entity descriptions at different abstraction levels. This is of special advantage in pervasive systems where entities are almost never known by a static identifier but rather by descriptive meta data, often in the form of semantic annotations.

In addition, the logical foundation of SWT makes it possible to analyse policies by reasoning over them and thereby helps users to debug and improve their policies, as e.g. argued by Verlaenen et al. [182]. Other frameworks do not use SWT, but are also based on logic representations, such as Z-Object (OPL), Datalog (Cassandra), or Prolog (Protune). So, building a policy framework on some logical underpinning seems to bee indeed promising and will be considered in framework developed in this thesis as well. An interesting question in this context is to which extend such formalism can be used for automatically enforcing side conditions or in order to automate policy analysis. Literature on the frameworks above in mainly about their functionality, i.e. the policy models they realise, but as information on the analysis features are scarce, this will be an area for further investigation in this thesis.

Further, Rein and KAoS also feature a specific policy model and do not support reactive policies. This is a drawback of all investigated frameworks: although some allow to extend their policy model within limits, none of them is open to the implementation of any application-specific policy model. As a result, the choice of a policy framework currently delimits the options to control a pervasive system the applications which run it – for example, if the framework features a trust negotiation scheme, it is not possible to use it for controlling context-dependent security settings. However, one characteristic of pervasive systems is to provide a generic middleware for distributed and ad hoc connected services, which can be used by completely different applications. So, when policy-enabling such a system, it should be guaranteed that the respective policy model can be adapted to the needs the application – a gap that should be closed by the framework developed in this thesis.

CHAPTER 4

Towards an extensible policy framework for pervasive systems

Based on the state of the art review in the previous chapter, we now derive the requirements for a security policy framework for pervasive systems and sketch the approach of this thesis to tackle these.

4.1. Requirements

The aim of this thesis is to close this gap in the state of the art by developing a policy framework which explicitly takes into account the requirements of dynamic and loosely coupled infrastructures, as they occur in pervasive systems. We will now work out the main requirements for such a policy framework. These requirements will serve as a guideline for the design and implementation of the framework's architecture and service as a benchmark for evaluating the achievements of this thesis. We classify them into the following main aspects: integration into the underlying middleware, control of security functions, the ability to cope with multiple policy domains, extensibility, and analysability. Furthermore, we will discuss some non-functional requirements.

4.1.1. Integration into the underlying middleware

A policy framework is usually integrated into some middleware layer – in our case we are specifically talking about the pervasive systems middleware architectures mentioned above in section 2.1.

Generic abstraction layer design First and foremost, purpose of a framework is to encapsulate often-used functionality and to make it available to the application layer via an API layer. In contrast to a software library, a framework includes a predefined control flow but is still a generic and extensible piece of software. So, it is important that the framework is applicable in as many pervasive systems as possible and does not presume a specific middleware. That is, it has to be designed as a generic abstraction layer that can be integrated into typical pervasive systems, independent of the specific protocols and platforms in use.

Minimally invasive integration into middleware Policy-enabling the middleware will usually require the integration of the framework's components into the control flow of the middleware. This integration must be as minimally invasive as possible because it would counteract the purpose of a framework if users had to spend significant efforts to apply it their systems. So, even in cases where changes in the middleware itself are not possible at all, the policy framework should provide means to policy-enable applications based on that middleware.

Extensibility despite one-time integration As integrating the policy framework into the middleware usually requires manual effort and bears the risk of breaking an application if not tested properly, this should be a one-time task. However, at the same time the framework must remain extensible. This applies to the different extension points of the framework (event adapters, information sources, etc.), but even to the whole policy model. In contrast to static distributed systems, a pervasive system might host applications with totally different security requirements, resulting in totally different policy models which are known at design time (i.e., when the system is set up). It is a drawback of most existing policy frameworks to support only a dedicated model: some feature RBAC models, others focus on trust negotiation, others again provide reactive policies. However, all of these might be required by any application that happens to be installed in a pervasive system at run time and it must be possible to apply these models without manually integrating a policy framework for each of them.

Access to existing domain knowledge Most pervasive systems use some kind of meta information in order to describe service capabilities and requirements. Often, this information is complemented by some representation of "domain knowledge", i.e. of information about users, situations, data sources, and other relevant information. As the security model of an application is often based on such information, it is relevant for policy specification and should thus be accessible by the policy framework. This implies pull access to existing knowledge bases, as well as the ability to receive push notifications about context changes.

4.1.2. Controlling the security of pervasive systems applications

In general, the framework is supposed to provide all mechanisms that are required to policy-enable a pervasive system, that is, specification, evaluation, enforcement, and analysis of policies. In this section, we will discuss the policy models which will be required in order to control these mechanisms.

Achievable protection goals The term "policies" is very broad and can be understood in different ways. In this thesis we will mainly concentrate on security policies, i.e., rules which control the security mechanisms within a system, whereas this does not necessarily exclude quality-of-service policies, for example. The types of policies which need to be supported can be derived from the protection goals the system should achieve, such as *confidentiality*, *integrity*, *authenticity*, *non-repudiation*, and *availability* [49], whereas availability is hard to control by policies and will thus not be in the focus of this thesis.
The two security mechanisms which need to be applied in order to achieve message confidentiality within a middleware are encryption and authorisation. Encryption prevents data from being eavesdropped during transmission or storage, and authorisation ensures that only legitimate subjects are granted access to a certain resource (e.g., a service or a file). For controlling encryption and authorisation in a system, the following policy types are required:

Authorisation policies and obligations *Authorisation rules* determine access rights in a system by means of positive (permit) and negative (deny) authorisations. On the basis of such rules, the framework must support the implementation of common access control models like RBAC [56], Chinese Wall [27], Separation of Duty, Bell-LaPadula [16], etc. Further concepts in the context of authorisation policies which should be supported by the framework are *obligations*, *provisions*, *delegation*, and *refrain policies*. Obligations, in general, are actions which must be executed in the context of an access request decision. They can either refer to accompanying actions which are triggered after the decision has been taken, such as logging or monitoring in the context of usage control (e.g., like in OSL [71]) or for realising audit logs, for example. Or they refer to *provisions*, which are executed as a prerequisite before the final access decision is made [135]. Delegation is the concept of transferring access rights from one subject to an other and refrain policies are negative authorisations which restrict outgoing access requests from a subject, for example in order to protect it from sending out private information.
The application of encryption protocols can either involve just a single entity, for example when storing encrypted data within the database of a certain service, or affect any number of entities, for example if data must be encrypted before it transmitted over unicast or multicast channels, and all concerned entities need to agree on a common protocol and common parameters. While authorisation policies are suited for stating that access shall only be granted if appropriate encryption mechanisms are in place, they are not suited for enforcing *provisions* which must be applied before access to the resource is requested. In [110], this functionality is subsumed by the term "usage policies" which "[...] control the selection and configuration of entities based on specific 'usage' data". In order to support this type of control, the framework must support the specification of provisions in security policies.

Controlling mechanisms for achieving integrity, authenticity and non-repudiation does not require for additional policy models. Data integrity and authenticity are traditionally achieved by applying digital signatures, mostly in combination with a Public Key Infrastructure (PKI) which attests the trustworthiness of the sender's public key. Non-repudiation further includes the usage of authentic and trusted time stamps in order to proof the creation of data packets at a certain point in time. Yet, the involvement of entities and their interaction is equal to the way how encryption can be controlled and thus authorisation policies with obligations and, specifically, provisions, can also be used to control mechanisms for integrity, authenticity, and non-repudiation.
The protection goal of availability focuses on safety, rather than security and will thus not be considered in this thesis.

Reactive policies Although all of the considered protection goals can be controlled by the policies required so far, it is often necessary to enforce a policy decision not only once, when a service is invoked, but to ensure a continuous enforcement. This aspect is called policy *continuity* and is especially important in usage control scenarios. Furthermore, we are dealing with dynamic software architectures and context-awareness systems, where changing conditions might demand spontaneously for a policy evaluation.
The previously introduced authorisation policies and provisions are synchronously evaluated as a consequence of an access request from a subject to a resource. This is sufficient for controlling all of the above-mentioned protection goals, but it does not take into account the dynamic environments of pervasive systems where the required security level depends on the current context and system architecture and may change as the operating conditions of the system vary. For example, while only trusted (i.e., identified and authorised) devices are present in the network, a relaxed and performance-optimised security configuration might

be acceptable, a stricter configuration will be required as soon as unknown devices enter the network. Implementing such asynchronous reactions is difficult using the previously introduced policies. Instead, event-triggered policies are required which specify actions that must be carried out upon the occurrence of certain events. According to its structure, this type of policy is often called Event-Condition-Action (ECA) policy in literature, but the shorter and more general term *event policies* is also used in the PCIM model [110]. Because we do not yet want to predetermine how these policies will be structured, we will simply call them *reactive policies* in this thesis, to underline their asynchronous and event-driven character.

With these policies – authorisation policies with obligations, provisions, delegations and refrain policies, as well as reactive policies – the achievement of protection goals in a pervasive system can be controlled. The policy framework developed in this thesis must thus support each of these functions in form of a *policy specification*, a *decision engine* and appropriate *enforcement mechanisms*.

4.1.3. Handling multiple policy domains

One property which distinguishes pervasive systems from other distributed systems is the existence of multiple administrative domains, which do not only co-exist, but can rather overlap each other. That is, a single resource might be under control of two or more different domains, which poses new challenges for policy frameworks to merge decisions from these domains and handle possible conflicts.

Allow grouping of resources into administrative domains As a first requirement, the policy framework must support the creation of domains at all, i.e. there must be some mechanism to assign resources to domains, where it must be possible for a single resource to reside in multiple domains at once.

Merging of cross-domain decisions If a resource is controlled by multiple domains, the policy framework must provide mechanisms to align decisions from these domains so that none of the involved policies is violated. The framework must further allow policy authors to state conditions for the merging of policy decisions in the form of “meta policies”. So, the framework shall not dictate a single algorithm for the combination of policy domains but rather leave it up to the user to adapt the strategy and define her own criteria for the combination of different domains.

Negotiations between domains Given that two domains generally agree on a policy decision, there might still be different preferences on its specific enforcement. This can refer to different equally applicable obligations, which fulfil the decision but show different characteristics in terms of performance, security, battery consumption, or other non-functional properties. Another example is trust negotiation protocols which require parties to concurrently reveal attributes one after the other. The framework must thus provide extension points at which such negotiation protocols can be plugged in and used by the decision engine.

4.1.4. Extensibility

A framework could hardly be called as such if it were not extensible. In fact, providing a extension points and easy ways to adapt its functionality is the main characteristic of a framework, so these mechanisms have to be designed carefully.

Functional extension points Evaluating a policy will require different components to be orchestrated: information needs to be retrieved from the middleware layer, policies and decision requests must be parsed and evaluated by some strategy, and enforcement points must be able to put decisions into effect. In order to make the framework applicable to a wide range of applications, it must provide extension points for all of these components. However, too much extensibility would render the framework arbitrary and not be of any help to the user. It is therefore important to find the right trade-off between usefulness and extensibility and provide a predefined functionality which can be used in many pervasive systems without the need to undertake extensive adaptations.

Extensibility of the policy model As identified in the state of the art review, most policy framework are centred around a predefined policy model and do not allow modifications or even replacements of it. In a pervasive system, it is however crucial that applications can use the policy model which matches their requirements, so the policies should be adjusted to the application's security model, and not vice versa. Instead of integrating one policy framework per model, a single generic framework should thus support extensions and replacements of the policy model in use. Ideally, the model would allow users to reuse existing concepts in order to write high-level policies which are then automatically broken down into enforceable low-level rules.

Support different policy representations The representation of policies should not be dictated by the framework. This requirement is in contrast to many existing framework which rely on a specific policy language. However, a specific language comes at the cost of limited extensibility – for example if the model is extended and the language does not provide the vocabulary to cover the additional concepts. Further, developers should be able to choose the appropriate way the write policies, like some Domain Specific Language (DSL), or a graphical editor.
Thus, the policy framework should be model-based and support different representations.

4.1.5. Policy comprehension and consistency

Analysability One of the greatest challenges for policy authors is the maintenance of rule sets and the anticipation of their implications to the application. Policies tend to become complex very quickly and sets several hundreds rules are no exception. Therefore, the framework should support users in understanding and analysing their policy and investigate its potential effects under different conditions. This could for example be realised by querying the policy model or even reason over it.

Constraint checking Policies can be regarded as "programming languages for non-functional behaviour", and just as programs, they can contain flaws, unintended redundancies or incorrectly handled border cases. As stated before, the requirement on analysability will help authors to investigate a policy and to actively search for such issues. However, in order to ensure that certain requirements are met, the policy model should support the declaration of constraints which are automatically tested so that it is not possible to write policies violating these constraints.

4.1.6. Non-functional requirements

The requirements discussed so far aim at the functionality of the framework. However, also some non-functional requirements must be kept in mind when designing the architecture:

Performance and scalability Performance of the policy framework should not be neglected. Although it is not the goal of this thesis to implement the most efficient policy framework, nor do we aim at reducing storage- or memory footprint as far as possible, it is however important to keep in mind that the cost at which a policy framework comes will determine its practical applicability. It should thus be evaluated what the limits of the framework are and by which means possible bottlenecks could be circumvented.

Security Finally, as the purpose of the policy framework will be to control the security of a pervasive system, it must be ensured that the framework itself does not impose any vulnerabilities and is protected against attacks. On the one hand, the framework must not dictate any security mechanisms if that would affect its ability to integrate into different middleware architectures. On the other hand, it must be possible to apply typical protection mechanisms to the framework, as required by the application. As a minimum requirement, it must be guaranteed that only authorised users are able to modify the policies. Thus, it must be possible to apply authorisation mechanisms for all administrative interfaces and authenticity to all control messages in the framework. Other security features like confidentiality of control messages or auditable proofs of policy decisions might be required in some situations, but should not be dictated by the framework.

4.2. Overview of the approach

We now introduce the framework design at a birds-eye perspective and show how it addresses the aforementioned requirements by anticipating some of the concepts which will be explained in detail in the subsequent chapters.

4.2.1. Framework architecture

Loosely coupled components As stated before, the framework should be easy to integrate into the loosely coupled architecture of a pervasive middleware. One of the main design paradigms for the framework architecture is thus to stick to the idea of loosely coupled components and adopt it as well for the policy framework. In general, the framework should make use of existing inter-component communication protocols which are already provided by the middleware instead of adding its own communication layer, in order to keep its footprint low. So, the framework will consist of a set of loosely coupled components which communicate over an abstraction layer on top of the underlying middleware's inter-component communication (ICC).

Integration into synchronous and asynchronous messaging Of course, just running the policy framework in parallel to other applications on top of a pervasive system middleware is of little use but rather, the framework has to integrate into the services used within an application. Above, we required that this integration should be minimally invasive and should not have to be repeated at a later time. For this reason, the framework provides only two generic ways to plug into the middleware communication: asynchronous event adapters and synchronous enforcement points.

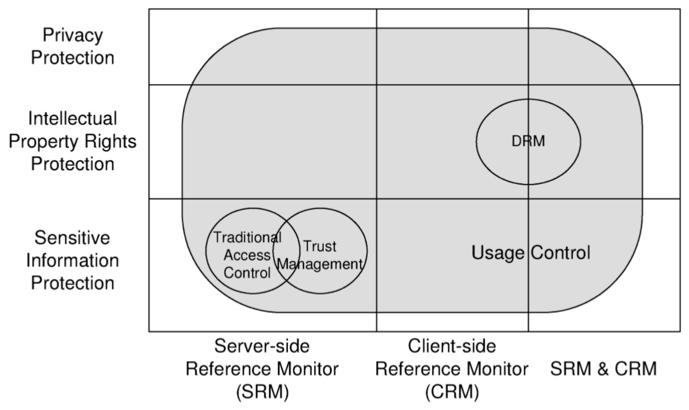

Privacy Protection
Intellectual Property Rights Protection
Sensitive Information Protection
DRM
Traditional Access Control
Trust Management
Usage Control
Server-side Reference Monitor (SRM)
Client-side Reference Monitor (CRM)
SRM & CRM

Figure 4.1.: Scope of the UCON model (acc. to [129])

Event adapters have to be integrated at the event messaging channel of the middleware – this can either be a simple publish/subscribe mechanism or a fully fledged ESB.
Enforcement points are handlers for outgoing and incoming messages from all services. They can either be realised in form of message handlers, proxies or hooks. Which of these patterns is most appropriate depends on the middleware and cannot be dictated by the framework.

4.2.2. Policy models

The policy framework should be suited to control the security of a pervasive system. As the term "security" is very broad, this refers in the first place to controlling all middleware mechanisms which aim at achieving the protection goals integrity, authenticity, confidentiality, and non-repudiation. Eckert [49] mentions further protection goals like availability and anonymity, which are not explicitly excluded from the policy framework's control capabilities but will play a minor role in this thesis, as they usually require for fundamental design choices and cannot dynamically be enforced in an existing architecture.

So, the policy model will be designed in such a way that it supports the aforementioned protection goals and at the same time allows to make policies as abstract as possible, so they are easy to understand, even for security laymen.

The approach we will be following in this thesis is therefore to abstract from concrete policy models and rather base the framework on a minimal "core" model which consists of some basic concepts and evaluation methods and to extend them by increasingly abstract models. A good starting point for deriving generic requirements is the $UCON_{ABC}$ model [129] which claims to be a superset of most existing schemes for information protection (c.f. Figure 4.1) and highlights *mutability* and *continuity* as key aspects of a policy model. Further, Pretschner refers in [137] mainly to requirements on software architectures for usage control models and states that beside a *negotiation mechanism* for exchanging information on usage restrictions, a *reference monitor* for supervision of legitimate use is required.
These statements will be taken into consideration for the design of the framework and a generic core model will be described, including entities of the system, such as subjects, and resources, upon which simple access control and reactive policies can be built. However, the strength of this basic design is to re-use existing concepts and evaluation methods and to combine them to more high-level policies. On the one hand, the model-based approach will ensure that policies are based on a clear semantics and can be analysed and queried.

On the other hand, it will serve as a generic policy representation so that the framework does not depend on a specific policy language.

Further below, in chapter 7, we will show how the basic concepts of the core policy model can be reused to create more advanced policy models on top of them

4.2.3. Cross-domain policy handling

The state of the art review above revealed that most policy frameworks assume a single administrative domain but do not focus at the combination of such domains. On the other hand, different authors have proposed policy merging mechanisms to combine multiple policies into one overall policy which fulfils every individual domain's constraints. While we also aim at fulfilling the requirement of supporting multiple and overlapping policy domains, the framework proposed introduced in this thesis will not follow these policy merging approach for mainly three reasons.

First, merging policies from different domains requires them to be based upon the same model and to use a common representation. As we stated in the previous subsection, however, the framework shall allow each domain to foster its own policy model and representation, so that a direct merge of them is not feasible. In contrast, merging policies of completely different models may turn out to be impossible or result in unwanted effects.

Second, approaches on policy merging require all policy domains to fully reveal their policies. However, policies may contain sensitive information that should not be disclosed, especially not to entities outside the administrative domain. We therefore consider policies as private and strive for some solution which allows to combine domains without requiring them to reveal their policies to each other.

Third, policies in pervasive systems are expected to depend on highly dynamic context information. As a result, when creating a unified policy for multiple domains, context conditions would have to be merged as well, which would require a formal semantics of the context modelling and an algebra for the merging operations. This contradicts however the demand for independent policy models and functional extension points. The latter will for example include different mechanisms for extracting context information from the middleware and for formulating conditions over them. As these extensions shall be application specific and can be provided at run time only, predefining a merging algorithm is not feasible.

Rather, the framework introduced in this thesis will aim at combining policy decisions, instead of the policies themselves. This allows each domain owner to still use his own policy model, as long as all domains use the same decision format (which is given anyway by using the framework). Policies can be kept private and only a minimal snippet that is relevant for the current decision is revealed. Further, users are free to use any domain-specific extensions for extracting and evaluating context information.

In order to declare how decisions may be merged with those of other domains, the framework provides a metapolicy mechanism. Metapolicies can be regarded as "policies about policies" and will allow users to declare under which conditions their domain may collaborate with others.

4.2.4. Extensibility and analysability

Extensibility is one of the main characteristics of a software framework and so it became one of the main requirements for the framework design. As the framework's architecture will be based on loosely coupled components for the sake of middleware integration anyway, using this architecture for plug-in based framework extensions is reasonable. The

approach we will be following is to define multiple extension points in the framework architecture to which functionality can be added the form of *plug-ins*. These plug-ins will only provide specific functions, such as the retrieval of attributes about a subject or an evaluation function for conditions, but will usually be part of an overall extension of the policy model. Therefore, they can be bundled in *extension modules*, which are basically a collection of additional functionality and can be loaded and configured in the framework at run time, i.e. without the need to re-deploy any components – especially without having to touch the integration into the middleware.
Apart from plug-ins, these modules can also contain extensions of the policy model, i.e., additional concepts which abstract from the core model and provide application-specific policies.

In order to simplify policy analysis, these models will be based on a logical representation, i.e. the policy model – the "structure" of a policy – is represented in logic axioms. In contrast to representations using a proprietary syntax without a logical complement, this allows not only querying, but also reasoning and constraint checking of the policy model. The choice of an appropriate policy encoding will be discussed in the subsequent section 5.1.

CHAPTER 5

Core policy model and mechanisms

Now that we have worked out the major requirements for a pervasive systems policy framework, in this chapter we introduce the core model, comprising generic and extensible concepts. The purpose of the core policy model is to provide the most basic building blocks all further extensions of the model can build upon. As we will see, the concepts of the core model alone can already be used to write simple policies. Nevertheless, their main purpose is to encapsulate all information which is handed to the PDP by an access request or an event, and to act as a building block for high-level policy extensions.

At first, we will motivate the choice of an appropriate *encoding* in section 5.1. By an encoding we understand the way to describe the structure and semantics of a policy model, in contrast to a *language* which also comprises the syntax. The actual syntax and tools for policy specification should not be dictated by the framework so at this point we will focus at the encoding only. The way how policies are encoded determines their expressivity and analysability, the usability of policy specification, and the complexity of the policy decision process. We will discuss different approaches and justify the decision for a hybrid policy encoding which is mainly based on Description Logics and is extensible by proprietary add-ons.

A policy encoding must be combined with an appropriate decision process which resolves the policy into a decision to be enforced by the PEPs. The design of the policy framework includes the definition of a generic decision process, structured into different phases, which we will introduce in section 5.2. While the individual phases are defined by the decision process, the individual tasks which can be carried out within each of these phases are not predefined by the framework but are instead provided by *policy modules*, which can be added to the framework as necessary and extend both the policy model and the decision process by further functionality. Such modules can extend each other's policy model and functionality and thereby build a hierarchical structure, whose root is the core policy model, defining basic concepts like access requests, policies, metapolicies, obligations (and provisions) and decisions. These basic concepts are introduced in sections 5.3 to 5.5, along with evaluation mechanisms which allow the specification of simple attribute-based access control (ABAC) policies event-condition-action (ECA) policies, and metapolicies for annotating decisions in multi-domain environments. Later, in chapter 7 we will then see how these basic concepts can be combined and extended to build more abstract policies which are easier to understand for users and better match the requirements of a specific application.

The author of this thesis has published the model of authorisation policies presented in this chapter in [153] and the model of reactive policies in [152] and [143].

5.1. Policy encoding

When creating a policy model some of the first questions to answer are: how will policies be structured and which expressivity should they have? The choice of an appropriate policy encoding is important, as it determines the expressiveness of the model, the ease of handling for developers, the extensibility, and the degree to which policies analysis tools can be applied in order to test for potential effects or conflicts. On the one hand, it is desired to have a formalised policy model and to represent and evaluate it based on that formalisation in order to avoid ambiguities in the specification. Consequently pursuing this approach would require covering all aspects of a rule by the formal model and to represent policies as a logic program.

A formal underlying defines a clear semantic so that no room for interpretation is left to the implementers of a policy decision engine. Further, based on the formal foundations, policy models can be analysed and checked for invariants, i.e. it is possible to define constraints which the model must adhere to. This is an advantage over schema-based policy specifications (like WS-SecurityPolicy or XACML) which can express certain constraints only informally by descriptive texts. Checking constraints at a logical level is not only a much stronger way of enforcing them, because the whole policy model will become invalid and thus not be applicable when it violates the constraints. It also allows generating explanations for the violated constraints in the form of logic proofs. This is of great help to policy authors, as they are immediately informed that their model is invalid and also receive an explanation of the cause. In section 7.2 below, we will give a practical example of Separation of Duty (SoD) constraint checking directly in the policy model.

On the other hand, handling policies should be as easy as possible for users in order to avoid errors in a policy, caused by an over-complex representation, while at the same time the model should be extensible so it can be adapted to different applications. Most commercially used policy languages, such as XACML, or WS-Policy, put the focus on efficient processing and therefore foster a proprietary representation.

From recent approaches on different policy encodings, three main directions have emerged: Description Logics (DL) based representations, Logic Program (LP) based representations, and proprietary representations. The discussion in the following subsections will show that none of the approaches fully matches the requirements on its own, but that a rather a hybrid approach is most promising.

5.1.1. Proprietary representations

Most commercially used policy languages rely on a proprietary encoding, but also Ponder, whose origins are in research, features such a proprietary representation. These languages represent policies in a custom format and use tailored implementations for their evaluation. For instance, the XACML specification defines an XML schema in form of a document type definition (DTD) to which policies have to adhere and gives informal instructions on how access requests have to be evaluated against such a policy. However, a formalisation is not given, so it is unclear how to deal with ambiguities in the specification. The evaluation process of a proprietary encoded policy language can be realised in any high-level programming language and basically boils down to testing attributes from the access request against conditions formulated in the policy. As conditions do not have to follow any predetermined logic, they can be of any required expressiveness. This is an advantage of proprietary encodings, because it does not limit conditional statements to the expressivity of the underlying formalisation. The downside of proprietary policy representations is that they do not provide any formalism. This is a hindrance to policy verification and analysis – the former aiming at confirming that certain properties hold for a given policy, the latter aiming at explaining the results of a policy in order to identify flaws and possible conflicts. Both functions are however highly demanded by policy authors and the manifold attempts to formalise existing proprietary representations (e.g., [90, 92, 28]) confirm the need for a policy model that is easy to analyse.

5.1.2. Description logics

In an attempt to put policies on a more formal basis, the use of logic languages for policy representation has been investigated by the research community. Description logics (DL) is a family of logic languages which have two main properties in common: they are decidable subsets of first order logic (FOL) and they are subject to the open-world-assumption (OWA). The latter implies that DL features monotonic reasoning and thus, in contrast to logic programs, facts which cannot be derived from the knowledge base are not automatically considered false. Description logics are mainly used for knowledge representation and have especially gained wide spread with the advent of the semantic web and ontology languages like OWL, which is a representation language of the $\mathcal{SHOIQ}(\mathcal{D})$ logic. Different authors have proposed approaches on expressing policies in DL: the policy framework KAoS [178] represents policies in an ontology (i.e., in DL) so that a standard semantic web reasoner can be used for policy evaluation. This ontology-based approach clearly provides a number of benefits:
at first, the structure of policies is inherently formalised due to their foundation in DL, and therewith can be easily analysed. Secondly, the usage of standard reasoning engines for policy evaluation and analysis is an advantage over proprietary policy representations, where it depends on the developer of the PDP to understand and implement a probably ambiguous specification correctly. Thirdly, representing policies in DL will facilitate the integration of entity descriptions from existing knowledge bases. Various ontologies for representing security-relevant information have been created and the ability to directly reference them from a semantic policy removes the need for any costly transformations. Examples are ontologies for describing identity credentials [194], generic security concepts [114, 50, 66], or services [54, 187, 146]. Furthermore, ontologies are increasingly used for model-driven software engineering [130], so that more and more software artefacts come with a semantic representation which can be linked to policies over these artefacts.

However, as pointed out in [170], the expressivity of purely ontology-based policies is limited and needs to be extended for most use cases. For example, the monotonicity property of Description Logic implies that it is not possible to overwrite existing axioms. As a result, it is neither possible to express the notion of order (e.g., when ordering rules by their priority), nor to overwrite one rule by another one. Further, as also stated in [170], the notion of variables is not supported. As an example the authors mention the use case of a "same location policy", granting access to files only where $\mathit{location}(\mathit{user}) = \mathit{location}(\mathit{file})$ to illustrate the need for variables. In KAoS, the authors try to overcome this deficit of DL by integrating the Java Theorem Prover[1] to evaluate parts of the policy. Rein, another policy framework based on description logics, chooses a different approach: although rules

[1]`http://www.ksl.stanford.edu/software/jtp/`

of a policy are represented in DL, the representation follows a subject-predicate-object pattern and is evaluated by dedicated rules, rather than by a DL-based reasoner. Variables are supported in the form of Prolog-like expressions which can be used in policies and are evaluated separately. With this approach, the authors indeed overcome the lack of variable support in DL, while still allowing to integrate domain knowledge from ontologies into a policy. However, due to the way Rein represents policies, the benefits of DL as a formal underpinning cannot be applied to them. That is, a formalisation of policies is not inherently given and reasoning is only applicable to domain knowledge but not for policies. As a consequence, Rein does not support policy verification or analysis, despite a DL-based representation.

5.1.3. Logic programs

Logic Programs (LP), consist of rules which are formed by a *head* and a *body,* where the head comprises a single predicate and the body is a conjunction of any number of predicates: $H \leftarrow B_1 \wedge .. \wedge B_n$. Just as DL, LP can be used for knowledge representations and in fact, they are used by expert systems like Prolog or Jess – even relational databases can be regarded as an instance of LP, as shown in [177]. Yet, there are some differences to DL, which provide advantages, but also drawbacks in the context of policy encodings.

In the previous section, it was stated that the main deficits of DL in the context of policy encoding are insufficient support of variables, lack of n-ary relations, missing support of negation-by-failure and the strict monotonicity property which prevents expressing rule priorities. All these features are fully supported by LP, as shown by the following Prolog examples. Horn clauses support any number of variables in predicates, in the head as well as in the body of a rule. The aforementioned "same location policy" can therefore easily be expressed by the following clause:

$grantAccess(user, file) \leftarrow locatedIn(file, x) \wedge locatedIn(user, y) \wedge sameLocation(x, y).$

For the same reason, expressing n-ary relations, such as the relation between a user, a resource and the assigned access rights is also not a problem in LP:

$hasAccessRights(user, resource, [rights]).$

In contrast to DL, Logic Programs are further based on a closed-world, rather than an open-world-assumption. The result is that in LP the absence of a fact is treated as its negation. For example, assuming an LP-based knowledge base, containing a single clause $grantAccess(john, patientRecord)$, the following query would return `false` as an answer, while in a DL-based model the result would be undetermined:
$? - grantAccess(bob, patientRecord)$

This behaviour of LP is one the one hand advantageous, as it complies with the expectations most policy authors will have when querying for non-existing facts, but on the other hand, it implies that inferred facts may change as further axioms (for example, $grantAccess(bob, patientRecord)$ are added to the knowledge base. Compared to that, facts inferred from a DL-based knowledge base are guaranteed to persist even if the knowledge base is extended at a later time. This non-monotonicity of Logic Programs, which is caused by the closed-world-assumption, also allows to express priorities of rules by ordering them. For instance, a common policy pattern is to create a generic rule and then override its effect for specific cases. As shown by the following Prolog example, expressing this pattern in LP is possible by ordering the clauses appropriately and selecting the first of all possible solutions, as opposed to DL where it would be impossible to express such behaviour.

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{decision}(\mathit{bob}, \mathit{patientRecord}, \mathit{grant}).} \\ \mathit{decision}(x, \mathit{patientRecord}, \mathit{deny}) & :- & \mathit{User}(x). \\ ? & :- & \mathit{decision}(\mathit{bob}, \mathit{patientRecord}, \mathit{grant}). \\ \texttt{true} & & \end{array}$$

Although it appears that LP can overcome the limited expressivity of DL, there are still problems which make a pure LP-based policy encoding difficult or even impossible. The first and foremost reason is that Logic Programs are undecidable. Consider the simple Prolog statement $x : -x.$ and a query for x. This program will never terminate as Prolog's forward reasoning engine will never finish to enumerate its infinite number of solutions. Undecidability is highly unwanted in the context of policy encoding as it could result in the policy decision point to stall forever when asked for the evaluation of an access request. Another argument against LP for policy encoding is that it does not feature a separation of domain knowledge and the actual policies. While splitting up a LP-based knowledge base into different files, one representing domain knowledge and the other representing the policy, is technically no problem, the underlying closed-world-assumption and the resulting non-monotonicity property of LP cannot guarantee that previously valid facts remain valid when further axioms are added. Therefore, maintaining the consistency of a policy encoded in LP will become a challenge if the representation is split up into several hierarchical layers, as it is envisioned in this thesis, in order to facilitate the process of policy specification for the user.

5.1.4. Conclusion and choice of encoding

From the discussion of the three main approaches on policy encoding in the previous sections, we will now derive the hybrid encoding approach which has been applied in this thesis. For this purpose, we need to consider the extensible nature of the envisioned framework and the complexity of pervasive systems. As each approach has its own strengths and weaknesses, we will review them by the following aspects, which are relevant for the envisioned policy framework, as we will explain in the following.
A policy encoding which is suited for pervasive systems ...

1. ... should be decidable
2. ... allows an efficient evaluation of policies
3. ... allows to separate policies into different layers of abstraction
4. ... allows to reuse concepts and pattern
5. ... supports extensions of the policy model
6. ... should be analysable
7. ... should be easy to handle for users
8. ... supports variables and rule priorities
9. ... allows the integration of dedicated evaluation mechanisms

Regarding decidability (1), not much explanation is needed. The evaluation of policies should be a decidable problem, so as to guarantee that every evaluation request will result in a decision. DL is a decidable subset of FOL and LP is in general undecidable, as mentioned before. A proprietary encoding is assumed to be designed in a way that it returns a decision for every access request in finite time.

What an "efficient evaluation" (2) is depends of course on the needs of an application and the expectations of the user. Nevertheless, for a practical application of the framework, it is important that decisions requests can be answered in a reasonable time span. In general, proprietary encodings will always be more efficient than generic logic engines. Also, LP tends to be slower than DL due to its greater expressiveness, as also discussed in [175]. However, significant efforts on performance optimisation have been undertaken for both LP and DL engines and despite the complexity of DL-based (up to NexpTime [169]) and LP based reasoning, the performance of reasoning tasks strongly depends on the size and structure of the knowledge base. In practice, many operations on knowledge bases of reasonable size are sufficiently fast, as shown in [25]. Therefore, and because evaluation time does not only depend on the encoding along, but also on the software architecture, we partly approve DL and LP for this criterion.

Being able to structure a policy into different layers of abstraction (3) will have two main benefits: firstly, considering the high dynamism of pervasive systems, the parts of a policy which refer to the entities of the system (e.g., devices, services, and users) will be subject to frequent changes. Thus, the domain model and low level policies must be adapted to varying conditions, while the high level policies will remain constant over a longer period. They reflect the overall security model which is unlikely to change frequently. Thus, partitioning the overall policy model into different parts will help to keep high level policies intact when modifying the more dynamic parts of the model.
Secondly, setting up a policy model requires expertise of different areas and might thus be done by different people. Separating the model into different parts fosters a separation of concerns so that every user can maintain the part she is responsible for, without inferring with the other parts. So, it is desirable that the policy encoding allows to split the model into different layers of abstraction which can be modified independently from each other. Description Logics and ontologies are predestined for such partitioning, as they firstly separate knowledge bases into TBox and ABox and secondly allow to extend a knowledge base by importing external ontologies. Splitting LP-based knowledge bases into different parts is possible but more difficult, as LP does not distinguish between structure and facts, for example. Also, the closed-world assumption makes it more difficult to keep the knowledge base consistent when parts of it are modified independently. Similarly, proprietary encodings often allow to include external files, however, they do not ensure that facts specified in one file are not overwritten by another one. So, LP and proprietary encodings approve only partly for this criterion.

Along with the previous requirement comes the demand for reusable concepts and patterns (4). If several users are working collaboratively on a policy, it is highly desirable that concepts predefined by one user can be combined to more complex policies by another one. This allows for example a system architect to model aspects of the system and a security officer to refer to these concepts when setting up the security policy. Both, in DL and LP, axioms can be reused throughout the knowledge base once they have been specified. Proprietary languages do not support this in general, although some languages, allow to reference elements which have been defined elsewhere in the document.

In contrast to most existing policy frameworks, we intend to design a framework whose policy model is extensible, i.e. can be adapted to the specific needs of the application so that the framework can be integrated in an existing middleware, independent from

Approach \ Criterion	1	2	3	4	5	6	7	8	9
Proprietary	X	X	(X)	(X)	–	–	X	X	X
DL	X	(X)	X	X	X	X	X	–	–
LP	–	(X)	(X)	X	–	X	–	X	–

Table 5.1.: Evaluation of encoding approaches against the stated criteria

the applications running on top of it. So, the encoding must allow to extend a policy model in a modular manner (5). The challenge here is in keeping the model consistent when adding additional modules, i.e. we must ensure that the semantics of the existing rules is not broken when adding extensions to it. Description Logics and ontologies have been designed with extensibility in mind. That is, they allow to split knowledge bases into different namespaces and lack a unique name assumption (UNA) so concepts and individuals with same name are not automatically inferred to be semantically equal. Further, DL is monotonic and lacks the negation-by-failure feature. This means that adding axioms to a DL knowledge base will not alter already inferred axioms and the semantics of an existing policy model would remain intact when it is extended. In contrast to that, LP reasoning is subject to the closed world assumption. That is, extending the knowledge base by additional facts may alter the semantics of an additional policy model. Also, most proprietary encodings, especially those used by commercial policy languages, have not been designed to support extensions of their underlying meta model.

To support analysability (6), not only the rules of a policy should be modelled in a knowledge base, but also the structure of the policies themselves. It has already been stated in section 5.1.1 that the lack of an underlying formalisation in many proprietary encodings has caused numerous attempts by the research community to formalise at least some parts of the existing specifications. Yet, as long as policies are represented in a proprietary encoding, reasoning over policies themselves is obviously not possible, as opposed to DL- and LP-based representations.

Judging what "easy to handle" (7) means is in the end up to each individual user. Yet, DL has been designed as a knowledge representation for end users and different graphical modelling and analysis tools like Protégé and Topbraid Composer are available. LP, in contrast, is mostly focused on logicians and experts, as it is mainly used in expert systems like Prolog or Jess and graphical modelling tools are scarce. Therefore, LP was not considered "easy to handle", as opposed to proprietary and DL-based encodings.

Variables and rule priorities (8) have been discussed above. They are not supported by DL, but can be realised with LP. As the expressivity of proprietary encodings is not limited per se, both features are assumed to be supported.

Integrating dedicated algorithms (9) as part of the policy evaluation process is required in order to connect policies to the information sources of a system. If an access request contains attribute values which must be evaluated against system-specific data, such as time, the number of currently active users, or the user logged into a certain session, mechanisms must be available to retrieve the respective information and carry out the comparison. This can easily be supported by proprietary encodings, however, for DL- and LP-based encodings the information would have to be added to the knowledge base before it could be taken into account. Although there are different ways to extend DL- or LP-based representations by custom extensions which serve exactly this purpose (for example, Prolog can be extended by predicates as Java functions), this would already be a hybrid approach and pure DL and LP do not fulfil this requirement. Table 5.1 summarizes

Choice of a policy encoding From Table 5.1 it becomes obvious that combining a DL-based approach with a partially proprietary encoding will cover all of the required features. The policy encoding used in this thesis will therefore be based as far as possible on Description Logics in order to benefit from the possibility to reason over the policies themselves and to provide a clear separation between domain knowledge and security policies. Where DL alone does not provide the required expressivity, for example for expressing priorities, it will be amended by a proprietary decision mechanism. In general, the core policy model supported by the framework will consist of a set of DL classes and properties (the TBox), along with some predefined individuals (the ABox). In order to set up a policy, a user will load a policy module into the framework and populate the contained ontology with individuals, reflecting the application-specific rules. Further below, in sections 5.3, 5.4, and in chapter 7, we will describe some examples of different policy models which address specific pervasive systems challenges.

A description logic knowledge base will typically consist of ontologies, represented in a language like OWL or short-hand notations like Turtle. Apart from explicitly stating axioms, it is also possible to amend an ontology with rules, written in the SWRL language, an extension of OWL. SWRL rules can be extended by so-called *built-in* predicates which allow to apply external functions to instances of a DL knowledge base. For instance, the built-in `swrlb:stringEqualIgnoreCase` predicate can be used to compare strings.
However, SWRL goes beyond the expressivity of Description Logic and is undecidable in general. As already stated above, this property is undesired when evaluating policies. Further, at the time of this writing, support for SWRL built-ins by DL reasoning engines was scarce and therefore requiring SWRL built-ins for policy evaluation would have resulted in practical problems.

Thus, the core policy model proposed herein will be based on description logic, represented in the OWL 2 language. SWRL rules are not used in the core model, but are generally supported by the framework, as long as they stick to SWRL-DL [131], a DL-safe subset of SWRL which does not extend the expressiveness of Description Logic and is therefore decidable. So, SWRL-DL rules can be regarded as "syntactic sugar" of an OWL knowledge base, because they could be replaced by an equivalent set of OWL axioms. Nevertheless, in some cases they can significantly reduce the amount of axioms which would otherwise have to be written explicitly.

Furthermore, for the sake of extensibility and because Description Logics alone may not expressive enough in all cases, the framework will support proprietary extensions of the policy evaluation process. By means of specific policy decision algorithms which may go beyond the expressivity of description logic, the framework allows users to write rules referring to external information sources and conditional functions, written in any high-level programming language. Although, a model amended by such proprietary extensions may not fully comply with the description logic foundation anymore, in some use cases it is important for policy authors to retrieve information from external databases or to evaluate conditions with custom functions.

5.2. Policy decision process

The framework proposed in this thesis evaluates policies by a *policy decision process.* Whenever a decision is required, e.g. because the PDP receives an event or access request, it evaluates the policy and returns a decision to the enforcement points. The evaluation consists of different phases which are invoked according to the process shown in Figure 5.1. Each phase serves a specific purpose and can be extended by plugins.

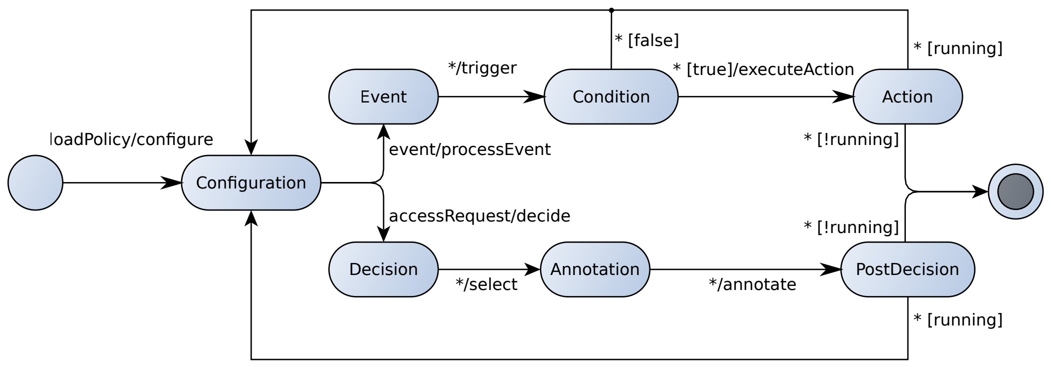


Figure 5.1.: Phases of the Policy Decision Process

During the *Configuration* phase, the PDP is prepared, settings required for the rest of the decision process are loaded, and event listeners are registered. The configuration phase is executed only once after loading the policy and is not repeated for every incoming access request or event. Its result is a set of key-value pairs which are provided as input for the subsequent phases of the decision process. When the Configuration phase is finished, the PDP is ready and listens for synchronous access requests or asynchronous events. Depending on the received input, the subsequent decision process is split into two sub-processes: the first one reacts on external events and triggers the execution of actions which have been defined by reactive policies. The second one evaluates access requests and returns an authorisation decision.

The *Event* phase is initiated by a notification about an external event. During this phase, the PDP transforms the external event into a description logic representation which is compatible with the core policy model, i.e. the key-value event is represented by a class expression with properties and fillers. The PDP then evaluates the event against the set of reactive policies and retrieves a list of actions which must be executed during the next phase. Modules contributing to the Event phase can register *EventAdapters* in order to receive events from different sources.

The next step is the *Condition* phase, in which the PDP evaluates conditional expressions of the applicable reactive rules. *ConditionPlugins* can be registered in the framework for the support of different expression languages and are invoked by the PDP during this phase. When a condition has been positively evaluated, the PDP continues to the next phase, otherwise switches back to *Configuration*.

In the the *Action* phase, all required actions which have been identified in the Event phase are executed. This execution takes place either in the policy decision point itself, or in the policy enforcement points which have been connected to the decision point. Modules can register *ActionPlugins* in order to provide custom actions which can be triggered using Event-Action policies.

Synchronous access requests are at first processed in a *Decision* phase. The access request contains information about subject, resource and access type, as well as various further key-value-pairs which can be set by the PEP. When receiving an access request, the PDP transforms the received information into a Description Logic representation, i.e., subject, resource, and action are each represented as a class expression, possibly containing further extensive descriptions. Then, various policy decision plugins (described in section 6.4.1) process the access request and return a decision which contains either a *deny* or *permit*, along with possible obligations. The final output of this phase is then determined by a *strategy* algorithm which selects one out of the set of applicable decisions.

Then, it is up to the *Annotation* phase to annotate the decision by the result of a metapolicy evaluation. This annotation is required for the resolution of policy conflicts which occur between different administrative domains. An approach of how to create such annotations by means of metapolicies and how to apply it for the purpose of conflict resolution is given in section 5.5.

As a last step, the request is passed along with the decision to the *PostDecision* phase. Plugins which have registered to this phase cannot modify the decision anymore, but they can log details about the decision as a basis for history-based access control or for auditing purposes.

5.3. Basic authorisation policies

One part of the framework specification is the definition of a core policy model, comprising the most important concepts and a simple decision mechanism. The purpose of the core policy model is on the one hand to define generic concepts which can later be extended by additional policy modules in a plugin-manner and on the other hand to allow a simple evaluation of access requests.

For the implementation of access control rules we therefore opt for a model which is as generic as possible and yet easy to extend by more specific modules while covering major concepts like *access requests*, *rules*, *policies*, *obligations*, and *decisions*. The core policy model comprises typical concepts of attribute-based access control (ABAC) models and structures an access request into a subject, a resource and the requested action and describes each of these entities by a set of attributes which is not further specified. It is one of the most generic access control models and therefore well suited for open systems such as pervasive systems or service oriented architectures.

Various research approaches as well as practical applications are based on ABAC. The most important policy language featuring an ABAC model is certainly XACML, which however lacks support for any kind of semantic representation. As the research community has recognised that ABAC would profit from semantic representations in terms of usability and expressivity, different authors have worked towards combining ABAC with semantic web technology in general and description logic in special. In [92], Kolovski proposes a DL-based formalisation of XACML and points out that non-monotonic aspects of XACML such as rule combining algorithms cannot be modelled with plain DL, which is why Kolovski's proposed formalisation makes use of Defeasible Description Logic (DDL^{-}) – a dialect which is currently not supported by semantic web technology like OWL, for example. Nevertheless, the benefits of a DL-based formalisation are pointed out in [91], where analysis techniques for XACML policies using a DL reasoner are shown. Various other approaches focusing at a more practical integration of XACML and semantic web technologies exist, such as in [38], where the authors propose to describe entities by RDF statements in SAML assertions and evaluate them against RDF statements in XACML policies. In [138], the authors propose a similar approach but move away from RDF and instead use SWRL to infer XACML attribute values from an OWL ontology. This approach is taken one step further in [40] where the authors use OWL and SWRL to describe different *contexts,* in which subjects and resources can act. These can be used in XACML policies by means of dedicated context attributes, which are interpreted by a custom XACML *AttributeDesignator*. In [57], the integration of OWL into XACML is used for realising role-based access control on top of XACML's ABAC model. Yet, all of these approaches only use DL as an extension of the proprietary XACML representation and do not propose a DL-based ABAC model.

5.3.1. Modelling Rules

As policies in the ABAC model consist of a sequence of rules, we introduce the rule model first. Each *Rule* is triggered by an access request and resolved to a *Decision*. We model access requests by a *Subject*, *Resource*, *Action* triplet, and decisions as a combination of an *Effect* – which is either *deny* or *permit* – with optional obligations. Obligations are actions which must be executed by the PEP before the access request is granted. In practice they can refer to different implementations such as Java classes or remote service calls. Whether one can rely on the PEP to actually execute this action depends on the trust relationships between PDP and PEP. For the case that the PEP is not fully trusted, it is possible to monitor the successful execution of an obligation by means of a *Monitor*, as we will explain in more detail below. Also, not all actions will result in observable effects. A more detailed discussion of the *Action* model will be given below in section 5.4.
So the model of a *Rule* is straightforward and can be written as follows:

$$\begin{aligned}
\mathit{Rule} &\equiv \forall \mathit{hasSubject}.\mathit{Subject} \sqcap \forall \mathit{hasResource}.\mathit{Resource} \sqcap \\
&\quad\ \forall \mathit{hasAction}.\mathit{Action} \sqcap \forall \mathit{hasDecision}.\mathit{Decision} \\
\mathit{Decision} &\equiv\ = 1\, \forall \mathit{hasEffect}.\mathit{Effect} \sqcap \forall \mathit{hasObligation}.\mathit{Action} \\
\mathit{Effect} &\equiv \{\mathit{deny}, \mathit{permit}\}
\end{aligned}$$

Now, we can model a policy as a sequence of such rules, together with a rule preference algorithm, determining how the effective rule should be selected among several applicable rules.

5.3.2. Putting rules in order

While the modelling of rules is straightforward in Description Logic, first difficulties occur when rules shall be put in order. Usually, rules are ordered by priority so that in case of conflicting decisions (i.e., multiple applicable rules returning incompatible decisions), the decision with the highest priority can be selected. However, modelling the order of rules is not trivial, as DL does not specify an operator for expressing order relations. Instead, a sequence of rules has to be modelled analogue to a linked list in traditional programming languages. In [46], a pattern for creating sequences in OWL is proposed, along with a number of regular-expression-like queries for matching patterns in sequences. While the general approach of modelling sequences as a linked list is reasonable, the proposed approach has two deficits in the context of policy modelling.
Firstly, the authors propose to model each individual in the sequence as a class and create the sequence itself in form of a complex class expression. This would to some extent foil the principle of using the TBox to declare unchanging and reusable concepts, as opposed to the ABox, where application-specific information is modelled.
Secondly, this way of sequence modelling allows the pattern matcher only to answer whether or not a certain pattern is matched by a sequence, but it is not able to retrieve the matching parts in form of either classes or instances. For a policy decision engine, however, it is not only necessary to know whether a policy contains a permitting or denying rule, but it also needs a reference to the specific matching rule in order to retrieve more associated facts from it, such as the requested obligations, for example.

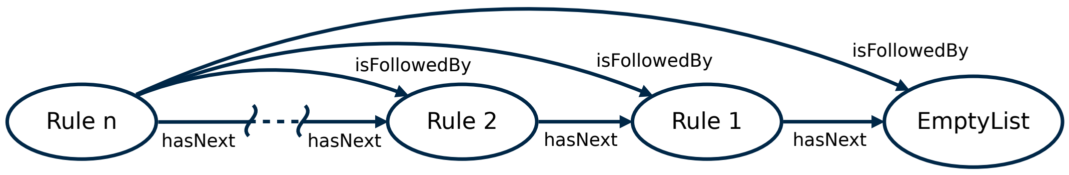


Figure 5.2.: Sequence of Rules in OWL

For that purpose, we slightly modified the approach from [46] and extended it in order to model sequences of individuals (in our case, representing rules), instead of classes. Further, we add helper classes which are useful for policy evaluation. In correspondence with the pattern from [46], subsequent rules are connected by a functional *hasNext* property, and the sequence is terminated by a dedicated *EmptyList* element which allows to identify the last element in a sequence by $\exists hasNext.EmptyList$. As it is easier to query for the last element in the list, as opposed to the first one, and the rule preference algorithms require to identify the first matching rule in a policy, rules are concatenated in descending order, i.e. the first element of the sequence contains the last rule of the policy and vice versa, as shown in Figure 5.2. Further, we model the helper classes *Deny* and *Permit* which comprise all rules with a *permit* or *deny* effect, respectively.

$$\begin{aligned}
EmptyList &\equiv OWLList \sqcap \neg(\exists isFollowedBy) \sqcap \neg(\exists hasContents)\\
Policy &\sqsubseteq OWLList \sqcap \forall hasContents.Rule \sqcap \forall isFollowedBy.Policy\\
\leq 1 hasContents &\sqsubseteq hasListProperty\\
isContentOf &\equiv hasContents^{-}\\
isFollowedBy^{+} &\sqsubseteq hasListProperty\\
\leq 1 hasNext &\sqsubseteq isFollowedBy\\
Deny &\equiv \exists hasEffect.\{deny\}\\
Permit &\equiv \exists hasEffect.\{permit\}
\end{aligned}$$

The helper classes *DenyFollowedByStop* and *PermitFollowedByStop* specify the sequence of permitting or denying rules, respectively, which precede the termination element *EmptyList*. *FollowedByNonMatchingRule* specifies the rules which are only followed by non-applicable rules. The complex concept *MatchingRule* used within this specification is not defined by the ABAC model itself but is rather constructed by the PDP according to the current access request and reflects all rules which match the attributes of the current access request.

$$\begin{aligned}
DenyFollowedByStop \equiv\ & Policy \sqcap \exists hasContents.Deny\\
& \sqcap \exists hasNext.(EmptyList \sqcup DenyFollowedByStop)\\
PermitFollowedByStop \equiv\ & Policy \sqcap \exists hasContents.Permit\\
& \sqcap \exists hasNext.(EmptyList \sqcup PermitFollowedByStop)\\
FollowedByNonMatchingRule \equiv\ & Policy \sqcap \exists hasContents.(Rule \sqcap \neg MatchingRule)\\
& \sqcap hasNext.(EmptyList \sqcup FollowedByNonMatchingRule)
\end{aligned}$$

Rule preference	Evaluation Dl query
first	$\mathit{MatchingRule} \sqcap \exists \mathit{isContentOf}.(\exists \mathit{hasNext}.\mathit{FollowedByNonMatchingRule})$
permit	$\mathit{MatchingRule} \sqcap \mathit{Permit} \sqcap \exists \mathit{isContentOf}(\exists \mathit{hasNext}.(\mathit{EmptyList} \sqcup \mathit{DenyFollowedByStop}))$
deny	$\mathit{MatchingRule} \sqcap \mathit{Deny} \sqcap \exists \mathit{isContentOf}(\exists \mathit{hasNext}.(\mathit{EmptyList} \sqcup \mathit{PermitFollowedByStop}))$

Table 5.2.: Supported rule preferences and DL queries used for evaluation

5.3.3. Defining policies in the ABAC model

Given this ABAC model, users can now write policies by first creating instances of *Rule* and then combining them to a policy by creating a sequence as shown above. Further, each policy is assigned a rule preference, determining which applicable rule should be selected by the PDP. So, a policy could look as follows:

$$\begin{aligned} \mathit{Policy}_x \equiv\ & \mathit{hasRulePreference}.\{\mathit{first}\} \sqcap \mathit{hasContents}.\{\mathit{rule}_n\} \sqcap \exists \mathit{hasNext}.(\\ & (\mathit{Policy} \sqcap \exists \mathit{hasContents}.\{\mathit{rule}_{n-1}\}) \sqcap \exists \mathit{hasNext}.(\\ & \ldots \\ & (\mathit{Policy} \sqcap \exists \mathit{hasContents}.\{\mathit{rule}_1\}) \sqcap \exists \mathit{hasNext}.\mathit{EmptyList}) \end{aligned}$$

The core policy model features three rule preference algorithms which determine how an individual rule shall be selected from the set of applicable rules for an access request. The *first* preference selects the rule which is followed by all other applicable rules, i.e. the rule with the highest priority. The *permit* preference favours those rules which permit the access request and selects the first applicable permitting rule, if existent, otherwise the first applicable deny rule. Finally, the *deny* preference acts alike, but naturally favours rules which deny the access request.

Depending on the rule preference stated in the policy, the decision mechanism will apply one of the queries stated in the following paragraph in order to select the applicable rule and its decision.

5.3.4. Evaluation of access requests

Given an access request containing a subject $s \in (\mathit{Subject})^{\mathcal{I}}$, a resource $r \in (\mathit{Resource})^{\mathcal{I}}$ and an action $a \in (\mathit{Action})^{\mathcal{I}}$, the PDP constructs a class expression describing all applicable rules:

$$\mathit{MatchingRule} \equiv \mathit{Rule} \sqcap \exists \mathit{hasSubject}.\{s\} \sqcap \exists \mathit{hasResource}.\{r\} \sqcap \exists \mathit{hasAction}.\{a\}$$

This class expression *MatchingRule* is then used to evaluate the policy $p \in (\sqsubseteq \mathit{Policy})^{\mathcal{I}}$ according to the rule preference algorithm of p, using one of the queries shown in Table 5.2.

Each of these queries will return at most one rule. If the result is empty, the policy does not contain any applicable rules and the PDP is not able to make any statements about

the requested access. In that situation, it is up to the PEP to handle the access request in a consistent manner. Usually, the default behaviour of the PEP would be to reject the request as if it would have been denied.
If the result contains a rule r, the PDP retrieves its effect e and all assigned obligations O and returns a policy decision d in form of a vector back to the PEP: $d =< e, O >$. The PEP will then execute all obligations and only permit access if each obligation has been carried out successfully and $e = permit$. Otherwise the access request will be denied. The core policy model does only define the structure of obligations, but does not predetermine any specific actions. What a meaningful action is and how it is realised depends strongly on the application and can thus only be defined by a user who is knowledgeable about it, i.e. typically a system integrator or the policy author.

5.3.5. Discussion

The ABAC model described in this section shows that although Description Logics do not support non-monotonic reasoning, they can be used to model policies with rule priorities and it is not necessary fall back to DDL^-, which is neither supported by OWL nor standard reasoning engines. Nevertheless, there is still some proprietary logic outside of DL required, even if it is only for creating the *MatchingRule* concept and selecting the appropriate evaluation query. Also, modelling sequences in DL introduces a considerable overhead and results in a fairly complex model. Especially the recursive definitions of *DenyFollowedbyStop*, *PermitFollowedByStop* and *FollowedByNonMatchingRule* are likely to have a negative influence on reasoning performance.
An alternative to modelling sequences of rules in DL is to simply enumerate rules of a policy. During policy evaluation, the PDP then chooses the *MatchingRule* with the greatest or smallest number, depending on the rule preference. As this alternative removes the need for modelling sequences, along with the *EmptyList* construct, the *hasNext* property and the aforementioned recursive class definitions, it is expected to provide a significant better performance than the sequence-based approach. However, this would be at the cost of loosing an understanding of rule priorities in the DL model and would therefore negatively affect the ability to reason over policies and to analyse possible outcomes.

5.4. Reactive policies

In the previous section, the concepts *access request*, *policy*, *rule*, *decision* and *obligation* have been introduced as part of an authorisation policy model. One property of such models is that they are evaluated only in a synchronous way, i.e. an access request triggers the policy evaluation and is blocked until a decision is returned and obligations have been executed. However, only reacting to access requests is not enough if security mechanisms shall be configured based on the current situation. Policy frameworks dedicated to pervasive systems, such as Ponder [4], therefore make use of *Event Condition Action* (ECA) policies – a simple control pattern that forms the basis of any event-driven architecture. Although existing ECA policies may differ in details, they all define actions to be executed upon the occurrence of events by following the structure ***on*** *event* ***if*** *condition* ***do*** *action*. The policy framework proposed in this thesis has to support an event-triggered execution of actions and must provide core concepts for modelling ECA policies and reasoning over them. The model for these so-called *reactive rules* has been published by the author of this thesis in [152], where we demonstrated its application in a model-based security event management system. Further below, in chapter 7, we will present several use cases, illustrating how these reactive rule concepts can be extended to realise a Dynamic RBAC model (c.f. 7.2), or a "self-protecting" system (c.f. 7.3), for example.

As event-condition-action rules in combination with Description Logic have been investigated by other authors before, it is worth to take a look at the existing approaches. The European Network of Excellence (NoE) REWERSE proposed the r^3 event ontology [9, 1] – basically an ontological representation of the ECA-ML XML scheme[2], developed by the MARS [39] group. For the continuation of the r^3 ontology, the initiatives WIDER [5] and K^{4r} [10] have been started, but were abandoned soon. Although the r^3 ontology looked promising at a first glance, it turned out to be no perfect match when we investigated its applicability for this thesis. It is mainly a meta model for language specific extensions. As an example, r^3 has been used as a meta model for the SNOOP [108] event specification language, for XPath queries, and for the Protune [142] policy language. On the one hand, supporting existing languages is reasonable, as it allows to integrate existing policies and allows users to stick to a language they already know. On the other hand, a language must first be mapped into the predefined r^3 meta model before it can be used. That is users have to correctly model elements of the language like *functors* and *compositional operators*, and then write an *Evaluator* engine for expressions of that language before they can make use of the semantics and reasoning support of r^3.
While the advantage of that approach is that expressions of different languages can be combined with each other due to their common semantic representation, the approach is not practicable for the use cases we envision in this thesis. Firstly, unlike in r^3, which aims at interoperability of rules between semantic web services, a semantic formalisation of existing languages would not provide any benefits in our case. Secondly, requiring users to create a semantic formalisation of an existing language is far too demanding. Apart from that, the resulting ontology would become fairly complex and event the authors of r^3 state that

> *just following the inference path, in order to test and demonstrate the prototypes, is a true challenge [5]*

So, the lesson we took into account from the r^3 ontology is that, in fact, it makes sense to support native languages for events, conditions, and actions, but that these languages should be directly handled by a specific engine and not mapped to a semantic meta model level first.

With the design of reactive rules for our policy model, we mainly at aim at two purposes: first, it must be extensible and support additional actions, event types, and condition statements, even in proprietary languages. Second, ECA rules must be represented in Description Logic so that policy authors can analyse and validate them using the framework's reasoning services.
The actual rule evaluation however, does not have to be done purely in Description Logic but can also integrate CEP engines like Esper[3], ruleCore[4], or Drools Fusion[5]. These engines have been designed to process large volumes of events from different data sources, detect patterns in this event stream and aggregate them to high-level events. Trying to completely replace them by a Description Logic engine is obviously pointless, we designed the ECA model so as to allow developers to choose a trade-off between the richer semantics of a DL representation and the better performance of a CEP engine. Nevertheless, the policy framework does not require any CEP engines but also allows to evaluate incoming events

[2]`http://www.semwebtech.org/languages/2006/eca-ml.dtd`
[3]http://esper.codehaus.org/ and http://www.espertech.com/
[4]http://www.rulecore.com/
[5]http://www.jboss.org/drools/drools-fusion.html

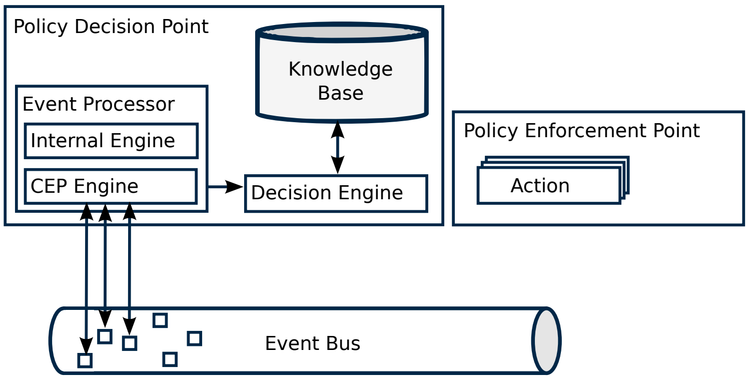


Figure 5.3.: High-level view of framework integration with CEP engines

using instance checking in the ECA model, as depicted by Figure 5.3. As stated above, this approach is however very limited in terms of performance and expressiveness.

5.4.1. Modelling events

The *Event* concept of our model describes complex event patters upon whose occurrence some action has to be taken. As stated in [168], the structure of events and the information they bear depends on the specific application, so predefining a dedicated and universally applicable event structure is not helpful and would result in difficulties for users to adapt the policy model to their needs. So, depending on how they shall be evaluated, events in the core model can be of different types which can be extended at a later time to match the application's requirements and the infrastructures capabilities. The core policy model only distinguishes between *dedicated* and *modelled* events, where the former refers to event types which are directly evaluated by a dedicated evaluation engine and the latter refers to an extensible event description which first needs to be translated into some format that can be interpreted by an underlying event processing mechanism. Further below in this section, we will point out the advantages and disadvantages of these two event representations. Common to all events is that they provide attributes which can be used for further processing of conditions and actions.

Dedicated events Thus, the most generic event structure of the framework is as follows:
$Event \sqsubseteq Thing \sqcap providesAttribute.Attribute$

When an event arrives, the PDP will represent it as a sub-class of *Event*, indicating the *event type* which relates to an evaluation mechanism. For example, the core framework implementation is able to react to simple events which are received by the underlying middleware's event bus. They are referred to by the *SimpleEvent* concept which identifies events by their topic:
$SimpleEvent \sqsubset Event \sqcap topic$

In most cases, this simple event type will not suffice, for example because complex patterns of events need to be detected, and thus it is possible to further extend the model by a *CEPEvent* type which is evaluated by an external complex event processing engine, identified by the *type* property:
$CEPEvent \equiv Event \sqcap hasQuery.String \sqcap providesAttribute.Attribute \sqcap type.CEPEngine$

This way of *dedicated event specification* is closely coupled to the specific event processing engine in the system and requires the policy author to know the CEP engine and its query language. Dedicated events are therefore a good choice when the event processing engine is known, unlikely to change over time, and the user is already familiar with its query syntax.

Modelled events Another way is to use *modelled events*, i.e. to describe event patterns by the model itself instead of relying on a proprietary query language. This brings the advantage that a richer semantics of the event pattern specification is available. Information like the observed event streams or the sensors feeding data into the streams can be modelled and used to formulate more high-level policies on top of it.
One example where such event models provide benefits are SIEM applications which aim at detecting security incidents in complex distributed system. In [152] we describe how the reactive policy model of this chapter has been applied for a security event management system. Based on the work carried out in this thesis, the model has been integrated into the next-generation SIEM engine, developed in the EU-funded project MASSIF [105].

A modelled event is determined by the *ModelledEvent* type, which refers to an *Extractor* and a *Criterion*:

$\mathit{ModelledEvent} \equiv \mathit{hasExtractor}.\mathit{Extractor} \sqcap \mathit{hasCriterion}.\mathit{Criterion}$

An *Extractor* determines the observed event streams and the attributes which shall be received from them. One extractor can refer to multiple event streams and aggregate their attributes by using a *Function*, which operates over a certain *Window* of events. The window can either refer to a period of time, such as “all events within the last two minutes” or to an event count, such as “the last 10 events”. Usually, events contain numerous attributes but only a limited set of them is interesting in the context of a specific event pattern. Therefore, users can determine the set of attributes which shall be available for further processing by means of *Extractors* and their *providesAttribute* property:

$$\begin{aligned}\mathit{Extractor} \equiv\ & \mathit{usesEventChannel}.\mathit{EventChannel}\\ & \sqcap \mathit{function}.\mathit{Function} \sqcap \mathit{window}.\mathtt{String}\\ & \sqcap \mathit{providesAttribute}.\mathtt{String}\end{aligned}$$

When the attributes described by the extractor have been retrieved from the event stream, this does not necessarily mean that the detected event pattern is actually of interest. For example, an extractor would be suited to collect the values of all temperature sensors in a room and aggregate them to a 10 minute sliding average, but it would not be possible to define a threshold for critical temperatures. For this purpose, a *Criterion* concept allows to model such conditions. It operates over the attributes provided by the extractor and consists of a boolean function $\mathit{BoolFunction} \sqsubseteq \mathit{Function}$ along with a set of positionally ordered input parameters, denoted by the index $i \geq 1$ of the $\mathit{hasParam}_i$ property.

$$\begin{aligned}\mathit{Criterion} \equiv\ & \mathit{function}.\mathit{BoolFunction}\\ & \sqcap \mathit{hasParam}1.\mathtt{String}\\ & \sqcap \mathit{hasParam}2.\mathtt{String} \sqcap \ldots \sqcap \mathit{hasParam}_n.\mathtt{String}\end{aligned}$$

Functions, as used in the *Extractor* and *Criterion* concept, consist of an *Operator*, a set of input parameters, and an output attribute, determined by the *providesAttribute* property. Both, input parameters and output attributes, are untyped and single-valued. It should be noted that although the model itself does not enforce sanity checks to prevent output attributes from overwriting existing attributes, such checks can easily be added in form of a plug-in for inspecting the model, as described below in chapter 6.

$$\begin{aligned} \mathit{Function} \equiv\ & \mathit{op.Operator} \\ & \sqcap \mathit{hasParam}_i.\mathtt{String} \\ & \sqcap \mathit{providesAttribute}.\mathtt{String} \end{aligned}$$

As an example, the following listing shows the definition of a modelled event. It monitors a 30 seconds sliding window of syslog events by the *iptables* facility (i.e., the firewall) and calculates the average packet size. When it exceeds 1 KB, the criterion is matched and the respective rule is triggered.

```
:packetSizeEvent
        :hasName "HistoryDBServerAnomaly" ;
        :hasCriterion [ :booleanOp :gt ;
                        :hasParam1 "avgLen" ;
                        :hasParam2 "1024" ] ;
        :hasExtractor [
                :hasEventChannel [ rdf:type :SyslogChannel;
                        :hasField "SRC", "LEN" ;
                        :hasName "iptables" ];
                :hasFunction [
                        :hasParam1 "LEN" ;
                        :providesAttribute "avgLen" ;
                        :hasScope "30 sec" ;
                        :op :avg ] .
```

For comparison, the next listing shows how this complex event would be described in the EPL complex event processing language of the Esper engine. In fact, in the SIEM prototype of the event model which we published in [152], we automatically translate modelled events to EPL queries and register them in the Esper engine. When an EPL query triggers, the respective event model is loaded and can be used by the policy decision point. This way, we combine the efficient event evaluation from Esper with the rich semantics of our model.

```
SELECT  src?,
        avg(cast(len?,float)) AS avgLen
FROM    SyslogChannel.win:time(30 sec)
HAVING  cast(avgLen?,float)> 1024
```

Evaluation In the following, we briefly describe the evaluation semantics for modelled events. Whenever an event *e* is received by the PDP, it is inserted into the current sliding *window* of each extractor, defined by either a time frame or a number of events. Then, all extractors are applied to their current *window* and the extracted values are tested against the *criterion*.
We depict the format of the received event in left column and in the right column the definitions of event, function, and criterion, as it is read from the knowledge base.

```
e rdf:type    Thing
     :hasTopic  topic
     :hasAttribute  attr_1
     :hasAttribute  ...
     :hasAttribute  attr_n
```

```
event rdf:type    Event
     :hasName
     :hasCriterion  c
     :hasExtractor
         :hasEventChannel  ec
         :hasFunction  f
```

```
f rdf:type    Function
     :op  op
     :hasParam  p_1
     :hasParam  ...
     :hasParam  p_n
     :providesAttribute  res
```

```
c rdf:type    Criterion
     :function  b
     :hasParam1  var
     :hasParam  ...
     :hasParam  x_n
```

When the event has been received by the PDP, the following algorithm is applied to evaluate it against the *ModelledEvents* from the knowledge base.

```
# $E_w$ is set of events in current window
$E_w := \{e_i : Event \mid e_i\ in\ window,\ e_i.topic \sqsubseteq EventChannel\}$

# Apply function $f$ to all events in $E_w$
$R := op(E_w, p_1, .., p_n)$

# Check if criterion $c$ is $true$ for all attributes
for each $p \in R$ where $c.hasParam(p)$:
    if not $b(p, x_1, .., x_n)$:
        return false

# If criterion has been matched, continue with evaluation of condition
return true
```

5.4.2. Modelling conditions

After the PDP has processed an event it might need to evaluate whether the system is actually in a state which requires a reaction. For this purpose, users model *Conditions* referring to the current system state – as opposed to events, which denote a point in time. For instance, a typical event could refer to a device joining the network and multiple rules with different conditions can be written re-using the same event definition.

A condition is modelled as a boolean expression of *AtomicTerms*, connected by the concepts *AND*, *OR*, and *NOT*. Atomic terms refer to a query in a specific language, an engine which is able to execute the query, as well as a *Criterion* which is a condition over the query results. So, a condition is written as follows:

$$\begin{aligned}
Condition &\equiv hasTerm.Term \\
AND &\sqsubseteq Term \sqcap =2 hasTerm.Term \\
OR &\sqsubseteq Term \sqcap =2 hasTerm.Term \\
NOT &\sqsubseteq Term \sqcap =1 hasTerm.Term \\
AtomicTerm &\equiv Term \sqcap hasQuery.Query \sqcap hasCriterion.Criterion
\end{aligned}$$

The concept *Query* contains the actual query string, for example an SQL query, a Prolog expression, or a rule engine statement, along with a reference to the evaluation engine, and the set of input and output parameters expected from the query. So the *Query* concept is modelled as follows:

$$\begin{aligned} Query \equiv\ & queryString.\texttt{String} \\ & \sqcap hasEngine.QueryEngine \\ & \sqcap providesAttribute.\texttt{String} \end{aligned}$$

Evaluation Evaluation of a condition c with root term t is straight-forward, as shown in the following pseudo-code listing. Conditional expressions are evaluated in a recursive manner until the *AtomicTerms* are reached. As these refer to external mechanisms, their evaluation is outside of the semantics of the core policy model. This allows for the necessary degree of flexibility, as otherwise evaluations would be limited to the operations defined herein, which would be a significant limitation to the model.

```
evaluate(Term t):
  if t is OR:
    return evaluate(t.hasTerm.$t_1$ $\vee$ t.hasTerm.$t_2$)
  if t is AND:
    return evaluate(t.hasTerm.$t_1$ $\wedge$ t.hasTerm.$t_2$)
  if t is NOT:
    return $\neg$evaluate(t.hasTerm.$t_1$)
  if t is AtomicTerm: #t.hasEngine $eng$, t.providesAttribute $var$, t.queryString $q$
    load $eng$
    var := $eng$.query($q$)
    return var
```

5.4.3. Modelling actions

When the event pattern of a rule has been detected and the rule's condition has been positively evaluated, an action needs to be executed. At the level of the core policy model, an action merely refers to an executable piece of code which can run either locally or remotely and retrieves the rule evaluation results as input parameters.

Further, depending on the actual policies that will be realised on top of the core model, it will be interesting to get information about the successful execution of an action. For example, when an ECA rule refers to an action which simply writes an event to a local log file, it can be assumed that this action is correctly executed and no further measures are required. However, in a scenario where an action has to enforce system settings on a remote system, such as a firewall configuration, for example, the reliability of the execution depends on the trustworthiness of the remote system. In such cases it will be helpful to use *execution monitors* which detect the successful (or failed) execution of an action. With the help of such monitors it becomes possible to write ECA policies which are able to handle exceptions in the execution of actions, such as putting a system into a quarantine network or informing the user if the enforcement of a remote action could not be verified.

Monitors are a pragmatic and generic way to observe the execution of actions and we therefore include them in the core policy model, thereby extending the traditional ECA pattern. Obviously, their drawback is that it depends on the specific system architecture and the monitored actions if and to which extend their execution can be reliably observed. Pretschner [135, 134] distinguishes here between *controllability* and *observability* and states that the former can only be achieved by trust-guaranteeing mechanisms like TPM or DRM,

while the latter provides a weaker statement about the enforcement of an action but is applicable in most cases and can be used for "observable obligations" in usage control policies.
More formal models for monitoring actions are conceivable, but they require additional assumptions which might not be fulfilled by every system and can therefore not be made a prerequisite by the core policy model. In [158, 159], for example, the basic ECA model is extended by pre- and post-conditions for actions ("ECPAP") and a Petri-net-based algorithm for scheduling multiple actions in an appropriate order is proposed. Similar pre- and post-conditions are also used in the framework extension for goal-based policies in "self-protecting" systems, which is explained below in section 7.3 of this thesis. While the benefit of such detailed models is obvious, it will not be feasible in every system to model pre- and postconditions for actions and making this a prerequisite by the core model would thus limit the applicability of the policy framework to only few systems.

Rather, we model actions by a *type*, a set of *parameters*, an optional *target*, *monitor*, and optional events which are triggered on successful or failed enforcement, respectively. The *type* determines how the action shall be invoked (e.g., Java method, OSGi service, REST- or SOAP call, etc.) and relates to an enforcement mechanism which can be plugged into the framework, as described below in chapter 6. Each action may expect a set of parameters, indicated by the *acceptsParameter* property, which are provided by the enforcement mechanism and can be set during the previous evaluation of events and condition. Further, parameters can also directly be set in a policy so as to allow users to set values which are not automatically be provided during the evaluation process. The *hasTarget* property indicates the target for execution of remote actions and therefore depends on the actual action type. For SOAP- or REST actions, the target would specify a URL, for R-OSGi actions, a URL in combination with Java class and method name would be suitable, for example. The optional *Monitor* specified by the *monitoredBy* property determines the execution monitor (in form of a Java class) which is responsible for detecting if the execution of the respective action has been successful or not. It is immediately called before the action execution and is expected to return a *success* or *failure* response within a certain time span. By means of the *onFailure* and *onSuccess* properties, events can be issued which trigger compensating or affirmative actions. These actions should be enforceable as otherwise, as stated in [135], if even the execution of a compensating action cannot be guaranteed, trust in a correct enforcement solely depends on the trustworthiness of the executing platform – and in that situation adding extra likewise untrusted execution monitors would be of little use, obviously.

To summarise, an *Action* in the core policy model is structured as follows:

$$\begin{aligned} \mathit{Action} \equiv\ & \geq 0.\mathit{acceptsAttribute}.\mathit{Attribute} \\ & \sqcap \mathit{type}.\mathit{ActionType} \\ & \sqcap \mathit{monitoredBy}.\mathit{Monitor} \\ & \sqcap \mathit{hasTarget}.\mathit{Target} \\ & \sqcap \mathit{onSuccess}.\mathit{Event} \sqcap \mathit{onFailure}.\mathit{Event} \end{aligned}$$

Action types are not predefined but can be added by plug-ins in the form of individuals of the *ActionType* concept:

$\mathit{ActionType} \equiv \{\mathit{java}, \mathit{soap}, \mathit{rest}, \mathit{r\text{-}osgi}, ...\}$

For each action type supporting remote execution, a target structure should be added in form of a subclass of the *Target* concept, as in the following example:

$\mathit{ROSGiTarget} \sqsubseteq \mathit{Target} \sqcap \mathit{hasUrl}.\mathit{String} \sqcap \mathit{class}.\mathit{String} \sqcap \mathit{method}.\mathit{String}$

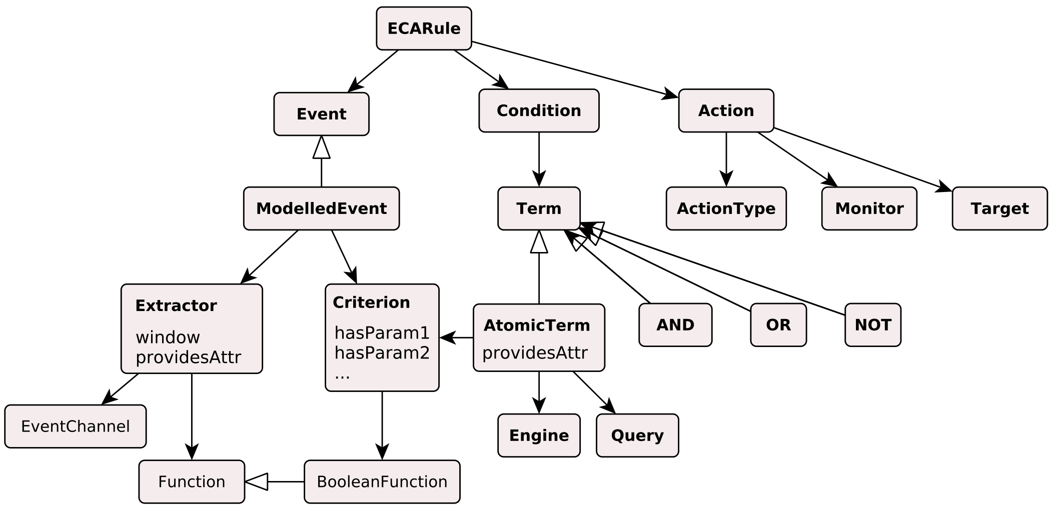


Figure 5.4.: Overview of the reactive policy model

Whenever the PEP is obliged to execute an action with this target, it will contact the remote service indicated by *hasURL* and *class* and invoke the specified *method*. In case the method cannot be called or returns false, the PEP issues a *failed execution* event which can be detected by a Monitor.

5.4.4. Discussion

With these basic concepts, it is now possible to specify policies to react on complex event patterns, and execute actions when a condition is fulfilled. The enforcement can be observed and further actions can be taken when the enforcement has succeeded or failed. Figure 5.4 shows an excerpt of the model of reactive policies with modelled events and conditions. We will now look at an example to understand how the model can be put into practice.

Example To illustrate the usage of a simple ECA policy, we look at the following example rule which requires the IP address of an unknown device to be logged whenever the device joins the network. This rule would require the definition of a *LogAction*, a *newDevice* condition and a *DeviceJoiningEvent*, and can be modelled as in the following example. Here the*LogAction* is an executable method with a single input parameter. Events of type *deviceJoiningEvent* provide an attribute `ip` whose value is used as the input parameter of the action. To ensure that only unknown devices are logged, a *newDevice* condition is added and refers to an external database to check whether the IP address is already known. When the PDP receives an event matching the *deviceJoiningEvent* query it checks the *newDevice* condition by calling the external *SQLEngine* and tests the result against the *isKnownIP* criterion. If it has been positively evaluated, i.e., the device IP is not contained in the DB yet, the action implemented by class `org.example.LogAction` is called and gets passed the IP address as input parameter.
While this rule is a very simple example, it shows the advantage of re-usable events, conditions, and actions which can be combined to more complex rules and have clearly defined semantics so that it becomes easier analyse the rule set, as well as to create tools which help users in defining correct policies.

$$\begin{aligned}
\mathit{LogRule} &\equiv \mathit{hasECAEvent}.\{\mathit{deviceJoiningEvent}\}\\
&\quad \sqcap \mathit{hasECACondition}.\{\mathit{newDevice}\}\\
&\quad \sqcap \mathit{hasECAAction}.\{\mathit{LogAction} \sqcap \mathit{hasParam}1.\{"\mathit{ip}"\}\}\\[1em]
\mathit{deviceJoiningEvent} &\in \mathit{CEPEvent} \sqcap \mathit{hasQuery}.\{"\ \langle\text{CEP query}\rangle\ "\}\\
&\quad \sqcap \mathit{providesAttribute}.\{"\mathit{ip}"\}\\[1em]
\mathit{newDevice} &\in \mathit{Condition} \sqcap \mathit{hasTerm}(\mathit{NOT}\\
&\quad \sqcap \mathit{hasTerm}(\mathit{hasQuery}.\{\mathit{retrIP}\} \sqcap \mathit{hasCriterion}.\{\mathit{myCrit}\}))\\[1em]
\mathit{retrIP} &\in \mathit{Query} \sqcap \mathit{queryString}.\{"\mathit{SELECT\ ip\ AS\ knownIPs\ FROM\ UserDB}"\}\\
&\quad \sqcap \mathit{hasQueryEngine}.\{\mathit{SQLEngine}\}\\
&\quad \sqcap \mathit{providesAttribute}.\{"\mathit{knownIPs}"\}\\[1em]
\mathit{isKnownIP} &\in \mathit{Criterion} \sqcap \mathit{function}.\{\mathit{contains}\}\\
&\quad \sqcap \mathit{hasParam}1.\{"\mathit{knownIPs}"\}\\
&\quad \sqcap \mathit{hasParam}2\,\{"\mathit{ip}"\}\\[1em]
\mathit{LogAction} &\sqsubseteq \mathit{ECAAction}\\
&\quad \sqcap \mathit{hasExecutable}.\{\mathit{org.example.LogAction}\}
\end{aligned}$$

Analysis capabilities Analysis of policies will mainly support users in identifying flaws and understanding possible effects of their rules. A common analysis task is for example to answer *what-if* queries in order to find out how the policy would react in a specific situation. With the ECA model, this can easily be done by retrieving the filler of the *hasECAAction* property for every *ECARule* that refers to a specific event.

Likewise, by reasoning over the model and querying it, one can find out when a specific action would be triggered, or verify that for every action an appropriate execution monitor is in place to monitor its enforcement. Also, if *modelled* events are used (instead of *dedicated* events) it is possible to investigate which event streams (i.e., data sources and sensors) are actually involved in a policy so that missing rules or unnecessary event sources can be identified. By creating hierarchies of event types, it becomes possible to bundle multiple events into one high-level event which can be used to trigger a generic action without repeatedly referencing to the same action in several rules. Further examples of analysing more advanced models based on the ECA model are given for the case of DRBAC in section 7.2, where possible role activations for an event are identified.

Conclusion and applications These simple reactive rules of the core policy model are the foundation of reactive systems and more advanced policy use cases depend on them, such as usage control (c.f. [135, 137, 134, 89]), or adaptive and "self-protecting" systems (c.f. [94, 89, 198] and section 7.3). The core model itself is not intended to be modified, but users of the policy framework are free to extend it so as to match their requirements. Further below, in section 7.2 we will show the extensibility of our code model by introducing a dynamic role-based access control model which extends the reactive rule concepts from this section for the activation and deactivation of roles.

5.5. Metapolicies for conflict handling

Policies, as described so far, result in a decision containing an effect (*permit*, *deny*, $\emptyset$) and a set of obligations, which have to be enforced by the PEP. This is reasonable in scenarios where all entities are under control of a single policy domain, but it does not fulfil the requirements of realistic use cases in pervasive systems. Here, we must consider scenarios where entities are concurrently controlled by multiple domains. One example of such multi-domain scenarios is collaborating enterprises, operating on a common resource. As all enterprises need to make sure that accesses to the resource follow their security policy, conflicts between the policy decisions of the enterprises may occur and additional information about the decisions must be taken into account to resolve the conflict. So, a policy decision does not only have to state *what* has to be done, but also *how it can be combined* with policies from other domains.

Motivation Frameworks like Ponder2 address this problem by creating hierarchies over policy domains and defining rules which prefer the decision of a superior domain over those of its sub-domains, for example. However, domains cannot always be hierarchically structured and thus more flexible approaches are needed.
Other approaches aim at merging the policies from all domains into one unified policy which complies with the requirements of all domain owners. This has two main drawbacks: firstly, it requires that all domains use the same policy format, i.e., both semantics and representation must be compatible with those of all other domains. Secondly, the approach requires all domains to reveal their complete policy to each other.

Both is problematic in the context of pervasive systems: domains will most likely use totally different policy models, so it will not be possible to simply merge policies while preserving their semantics. For example, a personal home automation domain could rely on a simple RBAC model, giving family members access to different digital resources in the household. When medical devices from that domain (e.g., a personal blood pressure meter) are going to be integrated into a value-added service provided by a hospital domain, it is highly unlikely that the hospital will run exactly the same RBAC model. So, merging both policies will not be possible in a straightforward way. Also, it is obvious that neither the personal domain, nor the hospital domain would be willing to reveal their policies for the sake of combing them. The policies contain information about family members, as well as employees and critical resources of the hospital. So, policies have to be considered as private information and in no case it should be required to reveal them to outsiders.

The policy framework developed in this thesis therefore strives for a way to merge policy domains while still leaving the policies private to each domain. This is achieved by *metapolicies* which enrich policy decisions by *annotations*, providing additional information about how to merge them with decisions from other domains. With this approach, the only information revealed to other domains is that contained in the policy decision. In order to merge it into a common decision, all domains must share a common model of obligations, so that it is possible to check whether obligations overlap or conflict with each other. Nevertheless, the policies themselves, i.e. the model and the individual rules do not need to be made public, in contrast to many other approaches. We published a description of the approach presented in this chapter in [149], building upon our previous experiences published in [155].

We will at first clarify what we understand by the terms metapolicy and annotation and review the state of the art in that area. Then, we will point out how metapolicies integrate into the overall core policy model, and then explain in detail the process of annotating and merging multi-domain policy decisions.

5.5.1. Related work

The concept of metapolicies has first been mentioned by Hosmer in [76] and was further elaborated by the author in the two articles [77] and [78]:

> *Metapolicies are policies about policies. They make the rules and assumptions about policies explicit rather than implicit and coordinate the interaction of multiple policies.*

So, Hosmer understands metapolicies as a tool to express how multiple policy domains shall interact with each other (among other purposes) and gives *Separation* and *Precedence* as exemplary strategies for domain combinations. *Separation* denotes the strategy of applying all policies separately and *Precedence* refers to policy priorities, given by an order over policy domains. Although this thesis aims at a greater expressiveness of metapolicies, the general understanding of metapolicies as "policies over policies" is in line with [76, 77, 78]. A slightly different perspective on metapolicies is taken in [11], where the author uses them for "*[...] governing the phases of the policy life cycle*" [11, section 5.2] in a policy-based management system.
Kühnhauser et. al. have applied the concept from Hosmer in an algebraic framework for multiple policy domains [96], which requires predefined *Conflict-* and *Cooperation Matrices*, specifying the cooperation between individual policies. This approach is not well suited to the envisioned pervasive systems use cases of the policy framework proposed in this thesis, as it requires users to know the collaborating policy domains in advance so they can set up the respective Conflict- and Cooperation matrices.
Other authors have taken the precedence strategy from Hosmer as a basis and proposed practical applications of conflict resolution strategies, based on a hierarchy over policy domains, such as in [191] or [148].
In [17], metapolicies are used to specify information about RBAC policies, such as the involved roles, data types, as well as invariants, which must be fulfilled by a policy. Compliance of a policy with this metapolicy is statically examined (i.e., at design time), and attested by issuing a *compliance certificate*. In this approach, metapolicies regarded as "policy interfaces", which help to combine policy domains by checking if their interfaces match. Most of the information the authors propose to represent in a metapolicies is also available in the framework proposed in this thesis: *data types, objects, roles, and functions* in [17] relate information represented in ontologies in this thesis. *Explicit rules* and *Invariance rules* in [17] relate to information which will be represented by metapolicies in this thesis. Yet, there a significant differences, as the approach [17] relies on a specific metapolicy model which is tailored to a hypothetical RBAC policy language and cannot be extended. Further, checking the compliance of a policy with its metapolicy has to be done statically (i.e., at design time) in order to issue the compliance certificate. As a result, checking the compliance of two policy domains in [17] will require human interaction and is not suited for combining policy domains ad hoc, i.e. at run time.
A first approach by the author of this thesis on metapolicies for conflict resolution between multiple policy domains has been published in [155]. Therein, metapolicies are used as invariants which are guaranteed to be enforced, even in the case of conflicts. According to its compliance with the invariant, a policy decision is either classified as *strict* (i.e., it must be enforced), *weak* (i.e., it may be overwritten by another domain) or is *dropped*, if it does

not comply with the invariant. The DL-based model allows to reasoning over metapolicies so as to prove that there are no conflicts in the metapolicies themselves. While the general idea of classifying policy decisions into different requirement strengths is continued in this thesis, the approach from [155] has been modified and its expressiveness has been extended.

In this thesis, we use Defeasible Logic (dl)[6] for describing the combination of policy decisions from multiple domains and thereby adopt an approach from [100], as opposed to the SQWRL-based combination process used before in [155]. A dl-based description is independent from the actual language used for implementation and is thus better suited for a formal description of the process.
The dl-based metapolicies described in [100] are used for the composition of multiple policy domains, where the authors assume WS-SecurityPolicy as the underlying policy language. By specifying the evaluation process in defeasible logic, they are able to consider dependencies between individual rules of different domains (e.g., "if another domain requires X, then I am willing to waive my requirement for Y and accept X instead"). However, as pointed out in the following subsection, the composition method from [100] has some drawbacks: at first, it results in incompatible domains in the case both domains define opposite alternative obligations in their metapolicy. This is of course unwanted, as in that situation, actually both alternatives would be acceptable for both domains. Secondly, [100] is focused on WS-SecurityPolicy only, so it does not allow each domain to foster its own policy model. In connection with this, the approach requires all parties to reveal their policies to each other in order to merge them into a single policy document. This is feasible for WS-SecurityPolicy, as these policies are intended to be communicated as a whole to the communication partner. It is however not suited for pervasive system where domains use different policy models and policies are considered private.
Further, the approach requires to test all subsets of the set of possible solutions in order to find a feasible solution so that the evaluation of a metapolicy is theoretically of 2^n complexity, where n is the number of variables in the defeasible logic theory. In a naive Java implementation, this approach required by average 417 ms[7] for an input set of just 18 variables, suggesting that even after thorough optimisation efforts, the approach will probably soon reach the limits of acceptable computing time, as the number of variables grows. While there is of course room for further optimisation, the approach proposed by Lee et. al. is unnecessarily complex and inflexible.

The approach on metapolicies used in this thesis, is related to the general idea by [100] of using defeasible logic for policy composition. It will however fix the mentioned problems and has been extended to deal with the policy decisions used by the proposed framework, containing access control decisions and obligations.

5.5.2. Basic metapolicy model

Metapolicies allow users to define how policy decisions may be combined with those of other domains. They are evaluated during the Annotation phase of the decision process and result in an annotation that is attached to the actual policy decision. PEPs which are registered in multiple domains can then merge these annotated decisions into one that complies with the metapolicy of all involved domains.

Authorisation metapolicy So, the structure of a metapolicy is not much different from that of a "normal" policy. Again, the framework provides some basic concepts as part of

[6]note that we abbreviate Defeasible Logic by *dl* and Description Logic by *DL*
[7]On Pentium a 4 Dual Core, Oracle Java 1.6.0_24, $\sigma = 169$ ms, $N = 22$

the core policy model and extension modules can add further concepts and evaluation strategies to them. Just as the basic authorisation rules from above, a meta-rules for authorisations are defined as follows:

$$\begin{aligned}
\mathit{MetaRule} &\equiv \forall \mathit{hasSubject}.\mathit{Subject} \sqcap \forall \mathit{hasResource}.\mathit{Resource} \sqcap \\
&\quad \forall \mathit{hasAction}.\mathit{Action} \sqcap \forall \mathit{hasAnnotation}.\mathit{AuthAnnotation} \\
\mathit{AuthAnnotation} &\sqsubseteq \mathit{Annotation} \sqcap {=}1\ \mathit{hasEffect}.\mathit{Effect} \sqcap \\
&\quad {=}1 \mathit{hasStrength}.\mathit{Strength} \sqcap \\
&\quad \forall \mathit{compulsory}.\mathit{Action} \sqcap \\
&\quad \forall \mathit{forbidden}.\mathit{Action} \sqcap \\
&\quad \forall \mathit{alternative}.\mathit{Alternative} \\
\mathit{Alternative} &\equiv {=}1\ \mathit{replaces}.\mathit{Action} \sqcap \forall \mathit{by}.\mathit{Action} \\
\mathit{Strength} &\equiv \{\mathit{strict}, \mathit{weak}\}
\end{aligned}$$

The main difference to the aforementioned *Rule* concept is that *MetaRule* refers to an *Annotation* instead of a *Decision*. Annotations assign a strength to the effect and thereby determine whether it has to be enforced in all cases (*strict*) or may be overwritten by another domain (*weak*). Further, metapolicies are expected to be conflict-free, i.e., there is no order over *MetaRules* defined and it is assumed that there is only a single unambiguous meta-rule for an access request. The core model helps policy authors to detect possible conflicts in their metapolicy by declaring *hasSubject*, *hasResource*, and *hasAction* as *key* for the *MetaRule* concept. Properties which are defined as key of a concept are assumed to uniquely identify that concept. In other words, if two individuals have the same *filler* of a key property, i.e. the same individual assigned by that property, both individuals are considered to be the same. More formally, the key axiom can be denoted as follows, according to the OWL 2 Direct Semantics specification [112]:

For individuals c, c', of a concept C, an individual y, and a property R that is declared as key property for C applies: `if` $(c, y) : R$ `and` $(c', y) : R$ `then` $c = c'$.

So, whenever two possibly conflicting *MetaRules* are defined, the reasoning engine will infer that these rules are equal. The user can then inspect the explanation of this inference to track down the reason of the conflict and to re-arrange the model in order to remove the conflict.
A further possibility would be to declare all individuals of *MetaRule* as different, thereby forcing the model to become inconsistent (c.f. *ABox consistency* on page 33). This way, it would automatically be validated that the metapolicy does not contain conflicting rules. However, as the core policy model serves only as a basis and is supposed to be extended by additional modules, such invariants could prevent the extension modules from applying their own evaluation mechanisms. For this reason, such a strict checking of conflict-freeness has not be integrated into the core model.

ECA metapolicy A metapolicy can also be defined for event-triggered rules. It is structured analogue to the ECA model from section 5.4 above – the only difference is again that a metapolicy does not require specific actions to be executed but rather declares how actions can be combined with those of other domains. The structure of an *MetaECARule* is thus as follows:

$$
\begin{aligned}
\mathit{ECAMetaRule} &\equiv \mathit{hasECAEvent.Event}\\
&\quad \mathit{hasECACondition.Condition}\\
&\quad \mathit{hasAnnotation.ECAAnnotation}\\
\mathit{ECAAnnotation} &\sqsubseteq \mathit{Annotation} \sqcap =0\,\forall \mathit{hasEffect.Effect} \sqcap\\
&\quad =0\forall \mathit{hasStrength.Strength} \sqcap\\
&\quad \forall \mathit{compulsory.Action} \sqcap\\
&\quad \forall \mathit{forbidden.Action} \sqcap\\
&\quad \forall \mathit{alternative.Alternative}
\end{aligned}
$$

Again, these concepts are intended to be extended by additional modules.
Only specifying a metapolicy is however not enough, obviously. It must be defined how metapolicies are processed and how they will finally control the merging of policy decisions from multiple domains. This will be described in the following sections.

5.5.3. Evaluation of metapolicies

The approach to metapolicies in this thesis is to use them as constraints which apply to policy decisions. That is, by means of metapolicies, users can on the one hand define invariants, which are guaranteed to be fulfilled, even in the case of inter-domain conflicts, and on the other hand relax requirements set by the original policy decision in order to support compromises between policy domains. In this section, we will describe the process of annotating a decision based on a metapolicy evaluation, as well as the combination of multiple annotated decisions. The evaluation of metapolicies is part of the *Annotation* phase, as shown above in Figure 5.1 and thus cannot modify the actual decision anymore, but will rather annotate it with further information. Concrete use cases involving metapolicies will be given below in section 7.4.

A policy decision, as stated above, is of the format $dec =< e, O >$, where e denotes the effect of the decision and $O \subseteq (Actionable)^{\mathcal{I}}$ denotes a set of obligations which must be enforced by the PEP.
The result of a metapolicy determines how e and O can be combined with other domains. For this purpose, effects can be classified as *strict* or *defeasible*. Strict effects must be enforced in every case, and other domains are not allowed to overwrite them. In contrast to that, defeasible effects may be overwritten for the sake of collaboration, if the policy of another domain requires so.
As for obligations, the decision of a metapolicy returns three classes: *compulsory*, *alternative,* and *forbidden* obligations. Compulsory obligations $\mathcal{C}$ denote actions which must be enforced by the PEP, regardless of the original policy decision or decisions of other domains. As an example, a metapolicy might require to log every access request, independently from what other domains require. Alternative obligations $\mathcal{A}$ denote alternatives which are may be applied in case the original obligation must not be executed due to the requirements of another domain. An example would be to demand an encryption protocol which is classified as *strong*, but to allow the usage of a protocol classified as *medium*, if the use of *strong* encryption is prohibited by another domain. Forbidden obligations $\mathcal{F}$, finally, denote obligations which must not be applied at all, even if another domain requires them. The result of a metapolicy evaluation can thus be written as $dec_{meta} = \langle e_m, \mathcal{C}, \mathcal{A}, \mathcal{F} \rangle$, where $e_m \in \{strict, weak\} \times \{permit, deny\}$, $\mathcal{C}, \mathcal{F} \subseteq (Actionable)^{\mathcal{I}}$ and $\mathcal{A}$ denotes a set of alternatives for obligations: $\mathcal{A} \subseteq (Actionable)^{\mathcal{I}} \times (2^{(Actionable)^{\mathcal{I}}})$

When classifying effects and obligations this way, the evaluation of metapolicies corresponds to non-monotonic reasoning, as it requires to revise a previously taken decision. Therefore, while metapolicies themselves can be expressed in Description Logic, a pure DL-based evaluation is not possible. We will therefore use defeasible logic [63, 192] to describe the evaluation of policy- and metapolicy decisions in the following.

On this basis, we can now write a policy decision $dec = \langle e, o_1, .., o_n \rangle$ as the following defeasible logic:

$$\begin{array}{rcc} [e_p]: & \Rightarrow & e \\ & \Rightarrow & o_1 \\ & \vdots & \vdots \\ & \Rightarrow & o_n \end{array}$$

If a metapolicy returns a non-empty result dec_{meta} for annotating the decision, the defeasible logic theory is extended as follows:

1. If the metapolicy returns a weak effect, add the defeasible rule $[e_m] : \Rightarrow e_m^{weak}$ and a superiority relation $e_m^{weak} > e_p$ to D.
   Thereby, effect e_m^{weak} overwrites e_p, but is still marked as *defeasible* and thereby allows other domains to overwrite it.
2. If the metapolicy requires the effect to be strictly enforced, add the strict rule $[e_m] :\rightarrow e_m^{strict}$ to D (a superiority relation is not needed at this point).
   This way, effect e_m^{strict} will be definitely provable and cannot be overwritten by another policy domain.
3. For all compulsory obligations $c \in \mathcal{C}$, add strict rules $\rightarrow c$ to D.
   This makes c definite provable and thus a necessary requirement that cannot be overwritten.
4. For all obligations $a \in \mathcal{A}$ acting as alternatives to an obligation o_i, add strict rules $\neg o_i \rightarrow a$ to D.
   This allows a to be an alternative obligation, in case that o_i could not be derived, because it was defeated by the theory of a collaborating domain. The alternatives, in contrast to o_i are written as strict rules and will thus be definite provable so they cannot be overwritten by another policy domain.
5. For all forbidden obligations $f \in \mathcal{F}$, add a strict rule $\rightarrow \neg f$ to D.
   This way, f is definite *not* provable and it is avoided that a collaborating domain can set it as a strictly required obligation.

As a result of this algorithm, the defeasible logic theory D, representing the original policy decision combined with the the metapolicy evaluation, looks as follows:

$$\begin{array}{lrcc} [e_p]: & & \Rightarrow & e \\ [e_m]: & & \rightarrow & e_m^{strict} \\ & & \Rightarrow & e_m^{weak} \\ & & \Rightarrow & o_1 \\ & & \vdots & \vdots \\ & & \Rightarrow & o_n \\ & & \rightarrow & c \\ & \neg o_i & \rightarrow & a \\ & & \rightarrow & \neg f \\ & e_m & > & e_p \end{array}$$

As each obligation refers in fact to a class expression in Description Logic, we need to add some further rules in order to ensure that obligations which are already subsumed by other obligations do not occur twice in the final decision. For instance, considering that one party requires *AES-256-CBC* while the other one requires *Encryption*, the former is a subclass of the latter. In that case, we do not want to apply two obligations, but rather one matching both requirements, i.e. the minimal subset satisfying all decisions. Therefore, for each obligation o_x subsumed by another obligation o_y (i.e., $o_x \sqsubseteq o_y$), rules are added to D so as to ensure that the existence of o_x removes the need for another o_y (i.e., $o_x \rightarrow \neg o_y$) and that forbidding o_y also forbids o_x (i.e., $\neg o_y \rightarrow \neg o_x$). Note that this transformation from Description (DL) into Defeasible Logic (dl) differs from the one in [63], where the author aims at using dl inference to reason over DL.

After the defeasible logic theory has been constructed in this way, it is appended to the policy decision and returned to the PEP. In case the PEP is under control of multiple policy domains and a conflict in the decisions arises, it will request the PDP to create a composed decision from the annotated individual decisions. How this work we will describe in the following subsection.

Composition of annotated policy decisions

The defeasible logic theory from the previous section can be thought of as a policy decision where certain parts are compulsory while others may be overwritten, if necessary. If a PEP is connected to multiple PDP, e.g. from domain A and B, and thus receives two theories D_A and D_B, it will derive the final applicable decision as follows:

1. Rename labels in D_A and D_B to ensure uniqueness
2. Merge the theories from A and B into one: $D' = D_A \cup D_B$
3. Reason over D' and retrieve the set $\mathcal{Q}$ of conclusions. For the sake of readability we will use $\mathcal{P} \subseteq \mathcal{Q}$ to denote provable facts, i.e. facts f which are definite ($+\Delta f$) or defeasibly ($+\delta f$) provable in D', and $\mathcal{N} \subseteq \mathcal{Q}$ to denote non-provable facts $-\Delta f$ or $-\delta f$, respectively.
4. Test if access rights of A and B are compatible:
   If $\mathit{allow} \notin \mathcal{P}$ and $\neg \mathit{allow} \notin \mathcal{P}$, both domains strictly require conflicting access rights and are thus not compatible with each other.
   If $\exists o_i \in D'$ s.t. $o_i \in \mathcal{N}$ and $\neg o_i \in \mathcal{N}$, the required obligations of both domains conflict with each other, for example because A states a compulsory obligation which is forbidden by B.

If the test in (4) fails, the decisions of domain A and B cannot be combined with each other, because both domains strictly require conflicting access rights. In that case, it is not possible to apply both domain's policies likewise and the composed service cannot process the request without violating the security policy of one of the back-end services. As a consequence, access could simply be denied by default, an administrator could be notified about the event and asked for taking a manual decision. Which one of these possible actions is suitable depends on the actual application and has to be agreed between the PDPs before composing their domains. Otherwise, if the decisions are compatible, all obligations $o \in \mathcal{P}$ will be executed and the respective access right enforced.

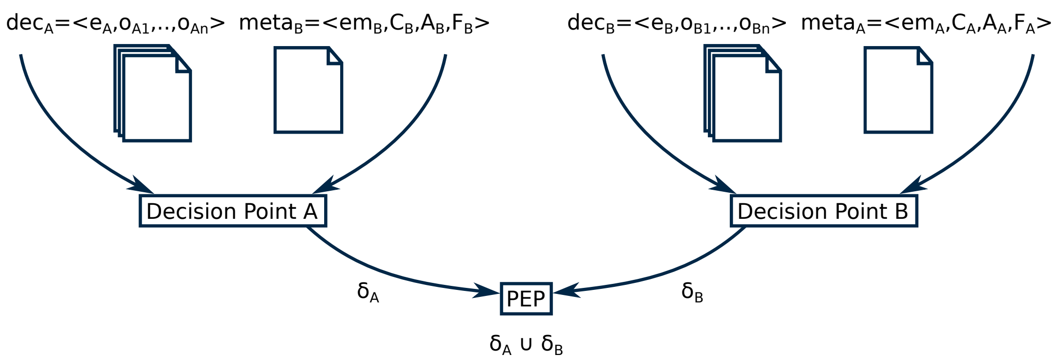

dec_A=<e_A,o_{A1},..,o_{An}>
meta_B=<em_B,C_B,A_B,F_B>
dec_B=<e_B,o_{B1},..,o_{Bn}>
meta_A=<em_A,C_A,A_A,F_A>
Decision Point A
Decision Point B
PEP
δ_A
δ_B
δ_A ∪ δ_B

Figure 5.5.: Composition of annotated policies from different domains

Properties of composed decisions

We will now show that the composition of policy decisions does not violate the constraints set by each individual decision. That is, the composed decision must not require obligations which have been marked as forbidden by one of the domains, it must contain all obligations which have been marked as compulsory and it must contain at least one of the alternatives for a defeated obligation o.

Proposition 1. *Given a defeasible logic theory $D = (F, R, >)$ and $\rightarrow \neg f \in R_s$ and assuming that $D \nvdash +\Delta f$, we can show that $D \vdash -\Delta f$.*

Proof. As we cannot conclude $D \vdash +\Delta f$, according to the aforementioned assumption, there can be no fact f, as otherwise (1.1) would let us conclude $+\Delta f$. So, as $f \notin F$, rule (2.1) is fulfilled. Further, in order to fulfil (2.2), we need to show that for all rules r enabling f the antecedent is not definite provable. This is obviously the case, as otherwise (1.2) would lead to the conclusion, $+\Delta f$ which we excluded in our assumption. So, under the assumption that $D \nvdash +\Delta f$ we can conclude $-\Delta f$, according to (2.1) and (2,2). □

Proposition 2. *Given a defeasible logic theory $D = (F, R, >)$ and $\rightarrow c \in R_s$ and assuming that $D \nvdash -\Delta c$, we can show that $D \vdash +\Delta c$.*

Proof. As $\rightarrow c \in R_s$ and the empty antecedent can always be concluded, (1.2) can be fulfilled and $+\Delta c$ follows. □

Proposition 3. *Given a defeasible logic theory $D = (F, R, >)$ and $\neg o \rightarrow a_i \in R_s$ and assuming that $D \nvdash -\Delta\neg a$, and $+\Delta\neg o$ we can show that $D \vdash +\Delta a$.*

Proof. With $\neg o \rightarrow a_i \in R$ and $+\Delta\neg o$, (1.2) is fulfilled. Further, as we assumed that $D \nvdash -\Delta\neg a$, there are no conflicting definite conclusions and we can derive $D \vdash +\Delta a$. □

Obviously, it is simple to show that the desired properties hold in a merged dl theory. However, the propositions above contain assumptions to ensure that the policy decisions of two domains are actually combinable. Namely, we assumed $D \nvdash +\Delta f$ in order to ensure that for a forbidden obligation f there is no other rule that strictly requires f. Likewise, for any compulsory obligation c we assumed $D \nvdash -\Delta c$, to avoid conflicts with rules which strictly require to not execute c and finally, we assumed that no rule exists that forbids the execution of an alternative obligation, s.t. $D \nvdash -\Delta\neg a$. These assumptions are not necessarily valid and need further discussion:

As dl is a sceptical logic, it does not allow to derive conflicting conclusions but rather marks literals of conflicting conclusions as ambiguous. Theoretically, ambiguities can be resolved by defining priorities over domains. That is, our approach would allow to create hierarchies over policy domains and let the decisions of a parent domain overwrite those of its child domains, just like the Ponder2 framework does. However, this would circumvent the guarantees of forbidden and required obligations and has therefore not been included in our conflict resolution strategy. Nevertheless, in scenarios where domain hierarchies are required, the current strategy can easily be modified by a respective Annotation Plugin for the framework. Here, we rather put up with unsolvable conflicts.

In case a conflict between annotated decisions cannot be resolved, none of the ambiguous conclusions is derived. The result is that for conflicts between forbidden obligations, where one policy domain strictly requires the execution of an obligation while the other one forbids it, both $-\Delta f$ and $-\Delta\neg f$ will be concluded, with the effect that the obligation will not be executed. The same applies for compulsory (and alternative) obligations, i.e. both conclusions $-\Delta c$ and $-\Delta\neg c$ (respectively, $-\Delta a$ and $-\Delta\neg a$) will be derived. So, by checking the merged dl theory for ambiguous conclusions for any literal in $\mathcal{F}$, $\mathcal{C}$ or $\mathcal{A}$, we are able to detect unresolvable conflicts between the policy decisions of different domains and handle them appropriately, as discussed above.

5.6. Constraint checking

Representing policies in a logical model provides the option to check the model for constraint violations and inconsistencies. As this can be extremely helpful for policy authors, we now discuss how we can leverage the capabilities of the semantic policy model to support such automatic constraint checking.

As we mentioned above, the model is evaluated in a hybrid way, i.e. by semantic reasoning over the model, as well as by involving proprietary plugins where necessary. Similarly, it is possible to validate constraints in two ways: either solely by means of the model's semantics, or by an external validation plugin.

For a validation purely based on semantic reasoning, three standard reasoning tasks come into consideration: *satisfiability checking*, *consistency checking*, and *instance checking*. The basic idea is the same for all three options: set up constraints in the ontology and test the ontology for existence of any instances violating the constraint.
Satisfiability checking tests if the concepts in the ontology are satisfiable, i.e. if the concepts can have any individuals. The *Nothing* concept, sometimes written as $\emptyset$, is by definition unsatisfiable. So, whenever an unsatisfiable concept X has been specified, the reasoner will infer X to be a subconcept of *Nothing*. In the context of policy validation, we can use satisfiability checking in order to reveal structural flaws in a policy. Let us consider the following example. We assume that we want to specify dependencies between obligation modules so the policy framework would be able to load all required dependencies before executing a specific obligation. This could be done by creating a *DependableAction* subconcept of the *Actionable* concept which references dependencies by a *dependsOn* property. As a module depends on itself, of course, we add a *dependsOn*.`Self` statement. Now, an obligation A depending on another obligation B would be written like this:

$$\begin{aligned}
\mathit{DependableAction} &\sqsubseteq \mathit{Actionable} \sqcap \mathit{dependsOn}.\mathtt{Self}\\
A &\sqsubseteq \mathit{DependableAction}\\
B &\sqsubseteq \mathit{DependableAction}\\
A &\equiv \mathit{dependsOn}.B
\end{aligned}$$

If for some reason obligation A and B are incompatible, for example because they serve conflicting protection goals, this could be expressed by making A and B *disjoint*. As a result, the reasoner will infer two disjoint concepts (namely, A and B) to be the same, leading to unsatisfiability of A and B. In the context of our policy validation, this means that the structural flaw in our model is revealed automatically and the policy author can be advised to fix it. For this purpose, an *explanation* in form of the inferred axioms leading to unsatisfiability can be retrieved from the reasoner. In our toy example, it would look as follows:

```
A disjointWith B
B equivalentTo dependsOn only A
B subclassOf Actionable
Actionable equivalentTo dependsOn Self
```

So, satisfiability checking is mainly used to detecting structural flaws in a policy or domain model and will automatically result in an explanation for the policy author, giving her hints on how to trace the defect.

Consistency checking searches for any individuals in the ontology which conflicts with another. Compared to unsatisfiable classes, inconsistency is a stronger violation, as it prevents the reasoner from further processing the ontology. Thus, whenever a policy model will be inconsistent, it will not be possible to load it into the reasoning engine of the PDP. As an example, we look at the Separation of Duty (SoD) constraint which we will discuss in more detail below in 7.2). Let us assume that in an RBAC model, two roles are incompatible – for example, a subject is never allowed to have the role of an *applicant* and that of a *funder*. This can be modelled by creating a class expression describing the unwanted constellation and specify it as a subconcept of *Nothing*:

$$\begin{aligned} Ssod_1 &\equiv hasRole.\{\texttt{applicant}\} \\ Ssod_2 &\equiv hasRole.\{\texttt{funder}\} \\ \emptyset &\equiv SSod_1 \sqcap Ssod_2 \end{aligned}$$

Now, consider that a subject s violating this constraint is specified. In that case, the reasoner will infer s to be an individual of the *Nothing* concept, meaning that an instance of the (by definition) non-instantiatable concept has been found. As a result, the ontology is marked as inconsistent and the reasoner will return an explanation telling the policy author that s was "forced to belong to class $SSod_1 \sqcap SSod_2$ and its complement" (namely, *Nothing*).
So, inconsistency checking can be used to detect individuals violating the constraints set by the structural model. In terms of policy validation, this is especially helpful, as authors can model overall constraints in the policy or domain model and be sure that they cannot be violated at a later time, when the ontology is updated.

However, as said before, an inconsistent ontology cannot be used for reasoning, which can be cumbersome in practice. Firstly, not in all cases, it will be reasonable to completely abandon the functionality of the reasoner, and therewith the PDP, just because one individual (which might not even play a role during policy evaluation) is violating constraints. Secondly, apart from stating which individual is inconsistent, the reasoner does not provide any further information or debugging support, as it cannot process inconsistent ontologies. This is a hindrance for authors who need to trace and fix the problem. For this reason, we converted the problem of consistency checking to a problem of instance checking.

The core model of the policy framework contains a specific *Invalid* concept for which policy modules can define subclasses in the form of class expressions, modelling a constraint violation. Any instance of these subclasses, representing a counter evidence for the constraint, will be detected by the framework using the normal instance checking feature of the semantic web reasoner. For each instance, a describing annotation can be shown to the user, as well as an explanation, which is basically the set of axioms which makes the respective individual an instance of the *Invalid* concept. Coming back to the SoD example above, we simply create the restricting class expression as a subconcept of the *Invalid* concept.

$$\begin{aligned} Ssod_1 &\equiv hasRole.\{\texttt{applicant}\} \\ Ssod_2 &\equiv hasRole.\{\texttt{funder}\} \\ Invalid &\equiv SSod_1 \sqcap Ssod_2 \end{aligned}$$

When the constraint is violated, the reasoner will infer an instance of *Invalid* and report to the user an explaining set of inferred axioms, as we have seen above. This constraint checking is automatically done whenever an updated ontology is loaded and can additionally be triggered by the user via the API. We consider this way of constraint checking as more practicable compared to the inconsistency checking, because even despite violations, the ontology remains valid and can be used for policy evaluation and debugging.

So far, we have discussed ways to check constraints purely based on reasoning over the ontology, i.e. the Description Logic knowledge base. However, reasoning over DL is not suited for a generic "schema" validation and so, some constraints which might be relevant for the user cannot be validated only by reasoning over the ontology. The reason is that DL reasoning is based on the open world assumption (OWA) – the assumption that facts might exist but are not represented in the model. This assumption is very helpful when combining ontologies or, as in the case of our core policy model, extending them by additional axioms, because it guarantees that all previously inferences remain sound and are not altered when adding further axioms to the ontology. However, it excludes the so-called *negation by failure* feature, which is essential for validating some constraints. For instance, let us consider a minimum cardinality constraint on the property *hasAnnotation* of the *MetaRule* class, i.e. the requirement that every meta rule must have at least one annotation. Theoretically, minimum cardinalities are supported by DL, so we can write

$$MetaRule \equiv \geq 1 hasAnnotation.Annotation$$

and consequently, in an attempt to make this a constraint for the policy, one could model a subclass of *Invalid* in form of a class expression `MetaRule and not (hasAnnotation some MetaRule)`, trying to match all meta rules which do not have an annotation assigned. If the reasoner would be able to find an instance of that class expression, this would proof that the constraint is violated.
However, the semantics of cardinalities here is different than what a policy author would expect in that situation. Under OWA, the pure absence of an axiom does not imply its negation – the reasoner simply assumes that that the axiom exist, but is not contained in the model. So, the aforementioned class expression will never have any instances and is not suited to test for violations of the constraint.
Other possibly unexpected effects are caused by the lack of a Unique Name Assumption (UNA). As an example, consider again the *MetaRule* class. It has a *key* constraint declared on the three properties *hasSubject*, *hasResource*, and *hasAction*, thereby making this triple a unique identifier for a meta rule. As an effect, the reasoner infers two meta rules to be the same, if they refer to the same subject, resource, and action. Although this result is intended and makes sense under the lack of a UNA, users might find it cumbersome that "rule 1" is inferred to be the same as "rule 2", for example. So, as long as not all individuals are explicitly modelled as *different*, putting uniqueness constraints on them will not result in an inconsistent ontology but rather in inference of *sameAs* properties.

For these reasons, the framework provides a second way to check the policy model for constraints, using external validation plugins, so-called *Validators*. A Validator is a plugin which has full access to the knowledge base and the reasoning engine and contains a proprietary algorithm for checking constraints in the model, usually by querying the model. Just as with the purely DL-based constraint checking, a validator will either confirm the integrity of the model or return a set of explanations if constraints have been violated. Validator plugins can be contributed by extension modules for the framework, which we will introduce in detail in the following chapter. In contrast to the policy model itself which is bound to open world assumption, the plugins can query the model with a closed world assumption (CWA), using a query language like SPARQL. With the CWA, the above mentioned requirement for a minimum cardinality can easily be checked by counting the annotations of each instance of *MetaRule* and ensuring it is greater than zero. Also, checking that meta rules do not refer to overlapping targets (subjects, resources, and actions), is easily possible by querying for two such instances where the target of the first subsumes the target of the second instance.

So, while the DL model provides some capabilities for constraint checking, only the combination with a closed world reasoning, as it is provided by the Validator plugins, allows to build policy modules which can automatically identify and explain inconsistencies and thereby support users in policy specification.

5.7. Summary

This chapter has introduced the conceptional core of the policy framework which serves as a basis for custom extensions, tailored to the needs of specific applications:

- a generic decision process into which specific policy decision algorithms can be integrated,
- an approach on a hybrid policy encoding based on formal Description Logics, amended by proprietary plugins, whose monotonicity property allows to extend the policy model without breaking its semantics.
- core policy concepts for authorisation (synchronously triggered) and event-condition-action (asynchronously triggered) policies
- a formalism by which policy decisions can be annotated with meta-information and an algorithm to combine such annotated decisions from multiple policy domains.

The generic decision process aims at covering as many policy use cases as possible and has been created after reviewing a variety of policy frameworks and the way they evaluate requests and events. While there is a plethora of policy models in use, ranging from access control, on to usage control, information flow policies, or policy-based management, their evaluation process boils down to receiving synchronous requests or asynchronous events, processing and evaluating them using some policy model and returning a decision. The decision contains information about access rights and might instruct PEPs to carry out certain actions. Also, it might be put into some context, i.e. a decision might have to interpreted in different ways, depending on condition, such as if the policy domain has been combined with another one. This generic process is part of the framework core and each of the individual steps can be adapted to the specific policy model at a later time.

The policy encoding determines the level of expressivity and formalisation of policies. After an evaluation of different options, the choice fell on an ontology-based encoding of

rules, combined with a decision process which evaluates their Description Logic based structure, but can be extended by further proprietary functions. This design has been chosen as it comes with a number of benefits.
Firstly, OWL 2-based ontologies have a formal semantics in Description Logics [112]. This is an advantage over policy frameworks which define a language only at syntactical level, rather than referring to a formal foundation. For example, even in mature languages like XACML, there are border cases which are not completely clear from the specification, and therefore it becomes difficult to foresee which decisions will be taken under certain circumstances. A formal underlying defines clear semantics so that no room for interpretation is left to the implementers of a policy decision engine. Further, based on the formal foundations, policy models can be checked for invariants, i.e. it is possible to define constraints which the model must not violate, as it would otherwise become inconsistent. This is a clear advantage over schema-based policy specifications (like WS-SecurityPolicy or XACML) which can express certain constraints only by informal description texts. Checking constraints at a logical level is not only a much stronger way of enforcing them, because the whole policy model will become invalid and thus not be applicable when constraints are violated. It also allows to generate explanations for the violated constraints in the form of logic proofs. This is of great help to policy authors, as they are not only immediately informed that their model is invalid, but also receive an explanation of the cause. In section 7.2 below, we will give a practical example of Separation of Dtuy (SoD) constraint checking directly in the policy model.
Secondly, Description Logics is monotonic, i.e. it is not possible to override or invalidate conclusions inferred from a knowledge base by adding further axioms to the knowledge base. This is an important property if the policy model shall be extensible, as it prevents an existing model from being overwritten by an extension. This property distinguishes our policy framework from others which rely on closed-world assuming systems like Prolog, for example.
Thirdly, semantic web technology is used in various applications, from resource categorisation in the web and in storage systems, on to the descriptions of networked devices and even sensor nodes [190]. Such already available information can be easily used and referenced by policies so that users will be able to link policies to existing information from OWL-S web service descriptions, or RDF meta data from desktop search engines.

Based on the semantic policy encoding, a "core policy model" has been introduced in this chapter, containing basic concepts for authorisation policies and event-triggered rules. These concepts are deliberately kept simple, as they build the generic structure of any synchronously and asynchronously evaluated policies and are intended to be extended by more specific add-ons to the framework. That is, the information modelled by these concepts will be provided at runtime by the policy framework in the form of class expressions and can be enriched by further, extended models to write rules at a more abstract level. Nevertheless, the core policy model can already directly be used for simple access control policies, assigning access rights on the basis of subject, resources, and actions. As each of these is modelled as a semantic class expression (or "complex concept", in DL speak), subjects, resources, and actions can be directly given by an identifier or described by a set of properties, thereby allowing an attribute-based access control (ABAC) with the core model.
Asynchronously triggered policies comply with the traditional event-condition-action pattern. Events can be modelled in two different ways: either by referring to some external event processing (CEP) or rule engine, or by a specific algebra within the policy model. Both approaches have their advantages and disadvantages and can be used even in parallel with the core policy model. Actions refer to pieces of executable code and can be combined

with Monitors, which are responsible for observing a successful execution of an Action. As the core policy model provides only the generic base concepts, there is no detailed event algebra given and it is left open which specific event processors, rule engines or action execution platforms are supported. Examples for these will however be given in chapter 7below.

Finally, by integrating a decision annotation mechanism and a formalism to merge annotated decisions, the framework provides the basis for combining policy decisions from independent and equitable domains. On top of this mechanism, applications can foster their own metapolicy model to determine a combination strategy. The choice of combining policy decisions instead of the policies themselves takes into account the fact that on the one hand pervasive systems may comprise administrative domains with totally different policy models and representations and that on the other hand, policies need to be considered sensitive information that cannot be frankly revealed to outsiders for the sake of combination.

The framework core introduced in this chapter has been designed to be extended by application-specific policy models in the form of modules. The following chapter explains how the theoretical concepts from this chapter can be put into practice in form of a software architecture which suits the needs of typical pervasive systems and chapter 7 illustrates several extensions of the core concepts discussed herein.

CHAPTER 6

Framework building blocks and software architecture

In the previous chapter, the core policy model supported by the framework has been introduced and the different phases of the decision process have been explained. In this chapter, the architecture of the framework will be introduced and the main building blocks, as well as their extension points will be described. At first, we will discuss the architecture from a high-level perspective and then describe the individual components in more detail, explaining how they interact and how the policy framework is supposed to be integrated into an existing pervasive system. An overview of the framework architecture and its prototype implementation "Apollon" has been published by the author of this thesis in [153].

6.1. Prerequisites

As identified in the requirements chapter, the main purpose of the framework is to "policy-enable" a pervasive system and to allow users to control the "behaviour" of the system in the most easy as possible way, while still providing the degree of flexibility that is required to put various different policy models into practice, depending on the application's requirements. Thus, one of the main aspects when designing the framework architecture was flexibility and extensibility. It was therefore chosen to develop a *framework*, as opposed to a *middleware* or an *application*. Because of the design goal of easy deployment and usage, it was also opted out to design a *library.*

Although there is no clear-cut definition of the term "framework", there is a common understanding that a framework provides functionality to a set of application which have some a common basis, for example web applications, smartphone apps or – in our case – pervasive system applications. Frameworks are thought to be integrated into an existing infrastructure so as to enrich its functionality by that of the framework. In contrast to a library, a framework usually provides all functionality that is required for a certain purpose (e.g., database management, web service creation or policy-based management) and may comprise executable components which can be started on their own on top of a runtime environment. A library is more limited than a framework as it encapsulates only some limited re-usable functionality and needs to be directly integrated into the application

code. It can therefore serve various applications, such as a XML parser library, for example, which can be used all kinds of XML-handling application, but it also requires more manual integration work and aims at application developers, rather than system administrators or end-users. In the context of distributed systems, there is often talk of a middleware, and we thus should clarify what the difference between a middleware and the policy framework is. Object of a middleware is to provide the necessary infrastructure that applications can build upon by adding an abstraction layer over the distributed components and making them accessible to the application in a way that hides the specifics of the distribution mechanism. In a distributed system, typical tasks of a middleware thus relate to accessing the distributed components, including service discovery, binding and invocation, as well as the support of heterogeneous platforms and different protocols. As applications need to be programmed against the interfaces provided by the middleware, there is often a closed coupling between the application and the middleware, while the actual platforms that are interconnected by the middleware can be exchanged without modifications of the application.
In the context of this thesis, this means that we will assume the existence of a pervasive middleware which provides standard functionality, without necessarily requiring any specific middleware system. The policy framework has been designed in such a way that it runs on top of the middleware and accesses its functions. However, as one design goal was to support portability of the framework so as to neither bind it to a specific application nor to a specific middleware system, care was taken to expect standard middleware functionality and to de-couple it from the actual framework implementation as far as possible.

Concretely, the policy framework relies on the following mechanisms to be available:

Service provisioning & discovery As the framework itself will provide services which need to be accessed by the application, it is required that the underlying middleware provides some mechanisms to discover services locally and remotely and to access their methods. This prerequisite will be no limitation, as service provisioning & discovery is a standard functionality which can be expected anyway in any distributed system.

Request interception Further, in order to "policy-enable" a distributed system, it will be necessary to hook into the request chain of services and to be able to intercept or modify requests according to policies. It is therefore required that the middleware layer provides such a mechanism. In many service-oriented architectures, modifying requests is a standard procedure, like for example in web service architectures where SOAP requests are processed by a chain of so-called "handlers" which apply modifications like encrypting, signing or transforming the request in any way. However, there are also more lightweight middleware systems which do not provide request interception out-of-the-box. In this case, integrating the policy framework into an existing system requires more manual work, such as creating service wrappers, for example, which proxy an existing service and make it "policy-aware". Another option has been applied in the prototype implementation of the framework (c.f. section 7.1) which was done on the basis of the OSGi middleware and specifically, R-OSGi for remote services – a lightweight middleware for distributed services which lacks immediate support of request interception per se. However, as the OSGi middleware allows to react on service registration events, it was possible policy-enable services by intercepting their registration event and replacing them on-the-fly with a dynamically created policy proxy.

Event bus In a reactive and context-aware system, communication is not only synchronous but also often asynchronous and anonymous. A common use case is sensor data, where sensors merely publish measurements without knowing the recipient and applications need to react on certain measurements without knowing the specific sensor service which published them. Such communication is typically realised by a publish/subscribe system where services subscribe to certain event types and an event bus mechanism takes care of routing events from the sources to the appropriate sinks. Being able to access the event bus is a prerequisite for the policy framework if any event-driven policy model should be applied. As almost every pervasive system middleware provides an event mechanism, this requirement does not put high constraints on the integration of the policy framework, however it depends on the specific event infrastructure of the middleware to which extend the policy framework will be able to react to events. While some architectures allow to access the "raw" event bus, thereby providing access to every single event that has been published in the system, some architectures requires event listeners to subscribe to specific topics. If there is no hierarchy of topics so it is not possible to subscribe to a root topic, integration of the policy framework requires to manually subscribe the framework to topics which are relevant for policy evaluation.

Component-based architecture A component-based architecture is not an absolute must and the policy framework will also be integrable in monolithic architectures. Nevertheless, many pervasive system middlewares are in fact built component-based and the advantages of such an architecture cannot be denied: components encapsulate contiguous functionality and provide it over interfaces to other components, thereby allowing a loose coupling between components so as they can be replaced or orchestrated in a new way at run-time. While components have been used for decades to describe logic blocks of an architecture during the design phase, reflecting this design pattern by actual implementations has gained popularity in the last few years. Nowadays, some applications like Eclipse or Protégé are using component-based architectures and thereby allow to update or replace certain parts of the application on-the-fly. In the context of pervasive systems, more ambitious approaches on "self-* architectures" exist, which are able to reconfigure not only some settings but even the whole software architecture at run-time. This is obviously an interesting feature, as it allows to re-structure a system driven by policies, so that it becomes for example possible to automatically replace services by a more secure or more lightweight variant, to choose the user authentication mechanism that is most appropriate for the current input device of the user or to automatically switch the communication protocol between services to the one that best matches individual requirements on the performance-security-resource trade-off.
The policy framework itself has been designed in a component-based fashion and will therefore easily integrate into likewise constructed middleware systems. Further, using a component-based middleware will pave the way for one of the most interesting applications of policies in pervasive systems is to control self-* features which require to dynamically load and execute new functionality or to re-structure the existing system architecture.

6.2. Overview

In general, policy-enabling a distributed system requires different level of expertise: on top, there are business or security rules which are often derived from compliance catalogues, legal regulations or contractual arrangements and therefore requires knowledge about

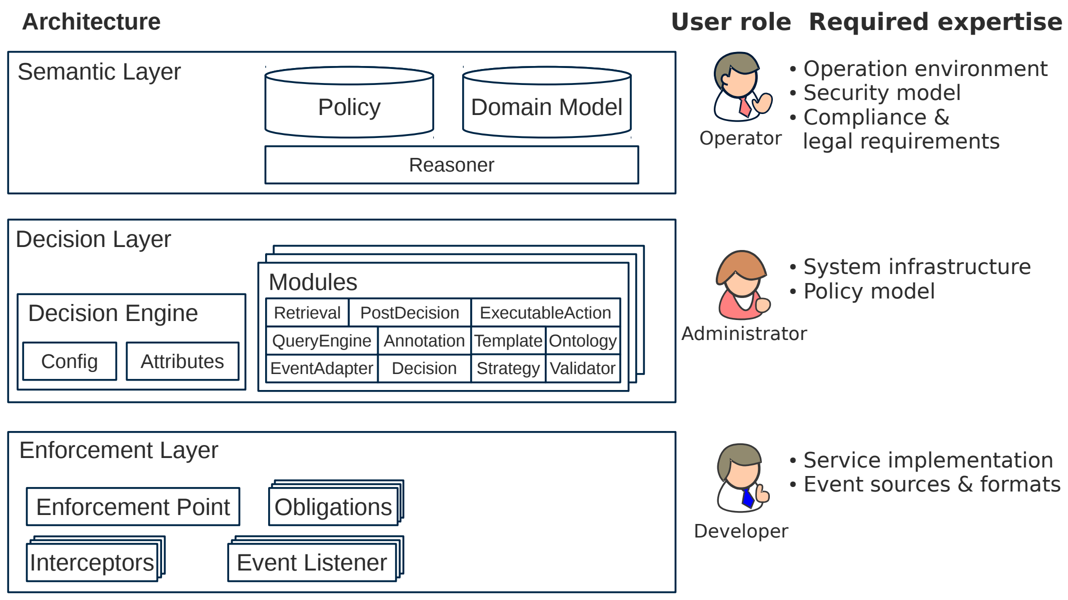


Figure 6.1.: Abstraction layers address different user expertise

these high-level standards.
To put these rules into practice then, they have to be written in a policy language or somehow be modelled in a machine-interpretable way. This requires the user on the one hand to be familiar with the specific policy system that is in place, but also with the architecture of the distributed system so such an extend that she can refer to the appropriate entities, users, access rights and conditions in a policy.
Finally, the distributed system needs to be controllable by the policy framework. That is, all policy-relevant information needs to be collected at various points in the infrastructure and provided to the policy framework, as well as all controllable components needs to be accessible by the policy framework. This requires more or less manual integration work, depending on the underlying middleware, as stated above. In fact, in most cases this integration will have to be done only once during the lifecycle of the pervasive system, but it requires in-depth knowledge about the system architecture and its implementation, and will therefore usually not be done by the user who is in charge of the high-level policy catalogues, for example.

One design goal of the policy framework developed in this thesis was to take into account these different levels of required expertise by providing abstraction layers which separate between tasks of different expertise as far as possible. That is, in contrast to other frameworks which do not provide such an abstraction, writing a policy will not be a task that requires a single user to be knowledgeable about everything: the policy syntax, the details of the system she is trying to policy-enable, and any possible security implications of the rules at the same time. Rather, the framework will provide three different abstraction layers at which users can bring in their different expertises, as shown in Figure 6.1.

Semantic Layer On the top is the *semantic layer* which provides the most high-level interface to a policy-controlled environment. It consists of a knowledge base which contains a domain model and high-level policies. A domain model describes all information that describes entities in an application and that is relevant to the evaluation of policies. This can include information about users, the network infrastructure, properties of devices, or event sources. For example, the sets of users, roles and permissions in an RBAC model is part of the domain-specific knowledge, as they depend on the concrete set-up and use case, in contrast to the policy model describing concepts like User, Role and Permission, which can be re-used in any other instantiation of the RBAC model and is therefore part of the underlying decision layer. As the domain model is completely application-specific it cannot be provided by the policy framework but rather has to be set up by an user who is familiar with the application and the underlying infrastructure and thus knows which information is relevant for policy evaluation. It is also not predetermines when the domain model has to be populated: some parts may be created when the application is deployed, some parts are created as part of the policy authoring process and some parts may even be dynamically updated as new services join the network. Managing this layer requires knowledge about the actual security model which should be implemented, as well as about the deployed application which has to be controlled by policies. A typical user role to interact at the semantic layer would thus be an operator, who is responsible for setting up and maintaining a pervasive system application and must ensure that the application follows certain guidelines derived from compliance catalogues or legal regulations. For this type of user, the actual implementations both of the policy framework as well as the underlying middleware is irrelevant – all he is interested in is to provide an application which behaves in the way that has been specified by the policy.

Decision Layer The next more detailed level is the *decision layer*, which comprises the components of the actual policy framework. As we will explain in more detail below, it contains the decision engine which coordinates different policy modules in order to derive reasonable decisions from input data gathered from the lower layer. This layer is both independent from the application, as well as from the underlying middleware. It merely provides the necessary components to policy-enable a system and determines the policy model, i.e. the rule structure and evaluation procedure. As described in section 5.2, the generic evaluation process for access requests or events is divided into different phases. The actual algorithms carried out during each of these phases are implemented in the components at the decision layer and several of these components can be bundled in one *policy module* which enrich the framework's capabilities. For example, by loading respective modules, the decision layer would provide the basis for a context-aware dynamic RBAC model which could then be instantiated by instances at the policy layer. An initial deployment of the decision layer components requires a user to know the system infrastructure and be able to add new services to the network. Also, the policy model which should be applied to an application has to be configured at this level. It therefore addresses users in the role of a system administrator, who are permitted and able to integrate components into the system architecture but do not need to deal with the actual policies. Once the components at this layer have been set up, there is no further need for interactions unless the policy model itself or any system configuration has to be adapted. As a result, policies can be adapted to possibly changing requirements at the semantic layer without the need to involve any technician.

Enforcement Layer Finally, the enforcement layer is responsible for gathering information about the system at run time and apply policy decisions at the operational level. The main components of this layer are thus event listeners which have to dock onto the system's event bus, as well as policy enforcement points which have to be integrated into the pervasive system in such a way that they are able to intercept service requests and load executable obligation plugins. The realisation of this layer obviously depends on the pervasive system,

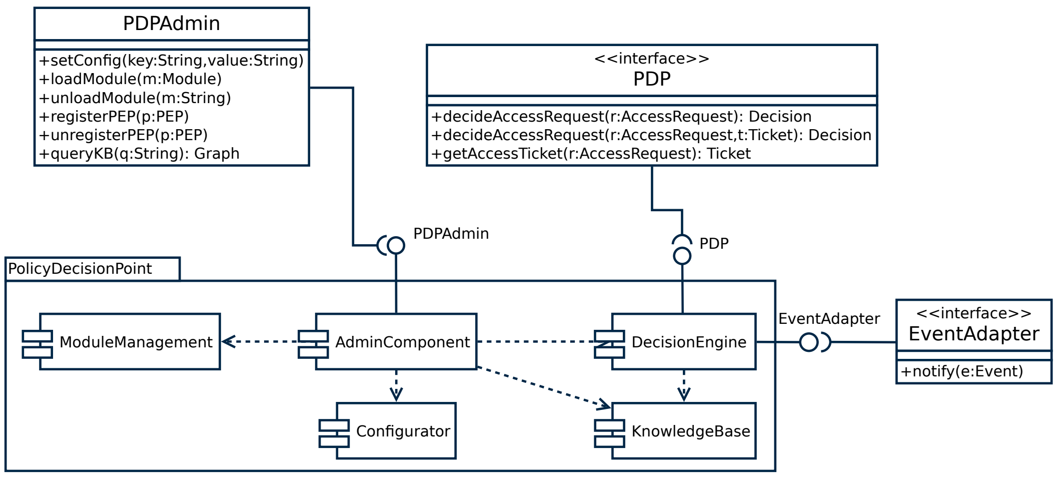


Figure 6.2.: Sub-components of Policy Decision Point

i.e. the middleware that is used to provide and interconnect services, and might require additional implementations of service proxies, event handlers, etc. Therefore, this layer targets developer users who know the extension points of the pervasive system middleware and are able to implement additional service handlers or event listeners, for example. Usually, this task will however have to be done only once when the policy framework is deployed to an existing middleware or even not at all, if the middleware is already “policy-enabled”. So, while establishing the enforcement layer requires the most system-specific knowledge, it can be used for any application and policy model, so that a developer is required only once during deployment.

6.3. Architecture

How can these layers now be put into practice? We have already briefly introduced some main components of the policy framework by Figure 6.1. In this section, we go into the details of each component and explain their design and interaction, starting with the core components of the framework, the policy decision components.

6.3.1. Policy decision components

The policy decision point is the main component and is central to a *policy domain*, i.e. the set of resources under the control of one administrative entity. It consists of multiple sub-components which provide the functionality of the aforementioned semantic and decision layer, along with a management interface for configuring the policy domain.

Figure 6.2 shows the main sub-components of the Policy Decision Points and the three interfaces they implement: `PDPAdmin` is the administration interface which provides methods to control the framework and is used by the command line interface, for example. The `PDP` interface wraps the actual methods for requesting access decisions and is invoked by the enforcement points. The `EventAdapter` interface enables the PDP to react on events received from an underlying event bus.

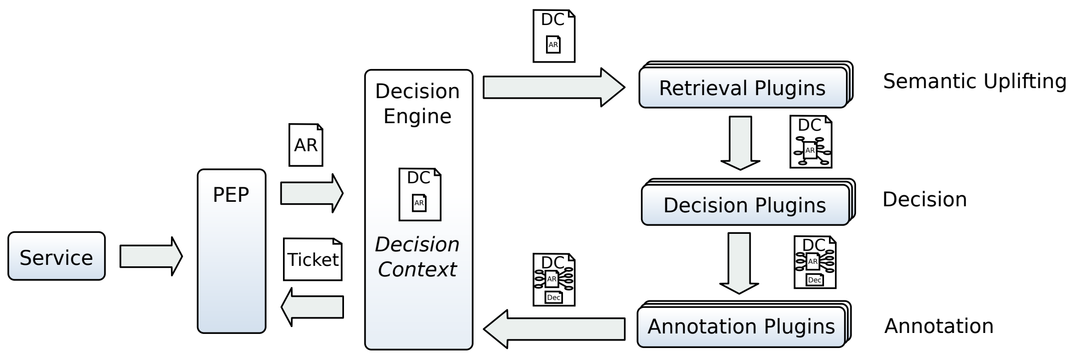


Figure 6.3.: Decision engine coordinates plugins

Separating these functions into different interfaces allows to assign different access rights to them, according to the envisioned user roles: `PDPAdmin` is supposed to be used by administrators only and this access to this interface should require individual authorisation, e.g. by means of client certificates or a username/password combination. In contrast to that, the methods of the `PDP` interface must be accessible by all PEPs within the PDP's domain, i.e. all PEPs which have been registered at the PDP before. Depending on the available resources and the required security level of the application, suitable authorisation mechanisms here range from simple checks of the PEP's source address (thereby leaving it prone to address spoofing, of course) to public-key infrastructures with client certificates. The `EventAdapter` interface, finally, will usually be publicly available so that it can be registered as a listener for an external event bus.
In general, binding access rights to interfaces is easier than controlling access to individual methods – for example the prototype implementation of the framework combines all three interfaces into one implementation but makes the `EventAdapter` interface remotely available, the `PDP` interface locally (i.e., within the same JVM) available, and allows to access the `PDPAdmin` interface only via a the dedicated command line interface and an Eclipse RCP-based rich client.

One of the PDP's sub-components is the *decision engine*. It implements the interfaces for receiving events and access requests and coordinates the policy decision process, as illustrated by Figure 6.3. Whenever a call to a policy-enabled service is made, the PEP intercepts that call and triggers the policy decisions process from chapter 5.2. Likewise, if the decision engine receives an asynchronous event, it starts the decision process and in both cases, it returns a policy decision. In case of access requests, the decision can be amended by a *decision ticket*, which is a token referring to the decision and is added by the PEP when forwarding the actual call. We will now look at the individual aspects of this decision process.

6.3.1.1. Decision Context Container

After receiving an access request or event, the decision creates at first a *DecisionContext* object (c.f. Figure 6.4) which bears the original request and serves as a container for attributes and control data which is collected during the evaluation process. At first, attributes of the event or of subject, resource, and action are extracted from the access request and stored in the context object. While in most cases, these attributes will relate to properties in the knowledge base (i.e., domain or policy model) and are set by Retrieval Plugins, as described in detail in the following subsection, it is also possible that a PEP

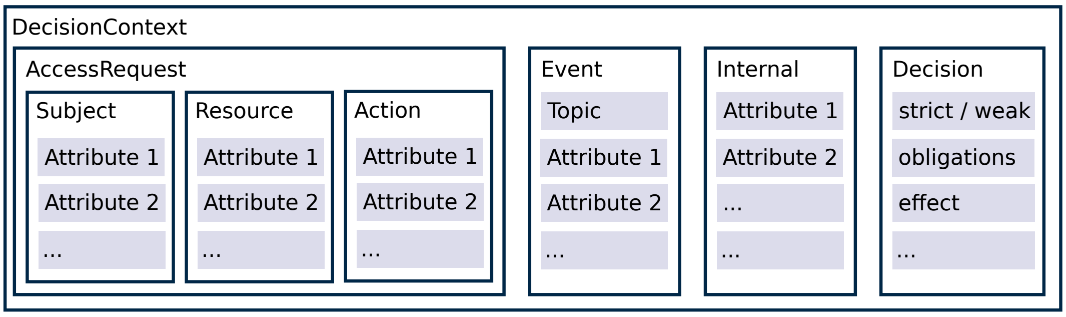


Figure 6.4.: Structure of DecisionContext object

provides attributes which do not have a semantic pendant and are directly interpreted by the decision engine. This is useful, for example, when access to the services of the framework itself needs to be controlled. Rights for accessing the `PDPAdmin` interface, for instance, are often bound to a specific user or client machine and configuring them by means of the policy model that applies to the rest of the system might not be sensible in all cases, as it would make the policy more complex and bear the risk that legitimate users to erroneously revoke their own right access the administration interface. So, by directly attaching such authorisation information, for instance a client certificate, access rights to the policy framework itself can be evaluated in a dedicated policy module without involving the models in the knowledge base. Further, the context object serves as a communication vehicle between the different plugins during the policy evaluation process, as it contains a list of internal attributes which can be set by one plugin and read by another one.

When the evaluation process has been completed, the decision engine stores the context object in a cache and assigns it a unique id. A *decision ticket* is then created, containing the unique ID, the decision effect, and the list of provisions the consumer has to fulfil. If monitors for these provisions are available, the decision engine activates them and then returns the ticket to the PEP, who in turn is responsible for executing the respective obligations before forwarding the actual call to the service.
This call will then be annotated with the ticket's ID so the PDP can relate it to the cached context object. If the decision was to permit access, the ticket is valid, and the monitors were able to confirm the successful execution of provisions, the decision engine grants access without starting an additional evaluation process. The validity strategy of tickets can be configured via the `PDPAdmin` interface and refers either to a certain timespan or to a certain number of usages of a single ticket.

6.3.1.2. Semantic uplifting

When the decision engine has created the DecisionContext object, the contained event or access request consists either of unique identifiers for the entities or a descriptive set of attributes. In order to reason over the request and evaluate it in the context of semantic policies, the attribute-based representation provided by the PEP must at first be transformed into a semantic representation, i.e. in a DL class expression. This process is sometimes called *semantic uplifting* and comprises two steps: converting the given access request into a class expression and amending it with additional semantic properties.
The first step is a mere syntactical conversion. If the PEP denotes entities (i.e., subjects, resources, or actions) by their identity, they are converted into DL individuals. Otherwise the decision engine constructs a class expression out of all attributes which correspond to

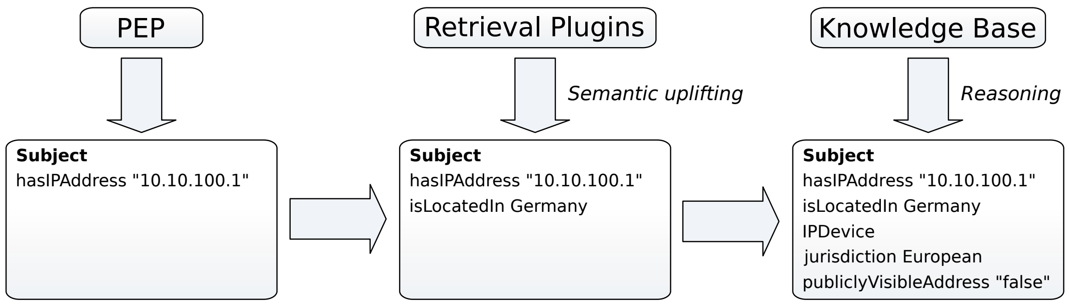


Figure 6.5.: Transforming an access request in to a semantic class expression

properties in the knowledge base. For example, assuming that the PEP describes a subject by the attribute `IP address=10.10.100.1`, the following class expression would be created: $\mathit{Subject} \sqcap \mathit{hasIPAddress}.$"10.10.100.1". The mapping between attribute names and properties can either be made explicit or can happen implicitly, based on naming conventions.
In many cases however, the attributes provided by the PEP will not be enough to evaluate the policy; assume for example that the policy refers to an attribute *isLocatedIn*, while the PEP only provides the *hasIPAddress* attribute. For this reason, the decision engine makes use of Retrieval plugins for semantically uplifting an access request. A Retrieval plugin will fetch these attributes by reasoning over the already known properties of the entity or querying any external database, and will provide the retrieved attributes in form of semantic properties. These properties are then added to the semantic description of the subject, resource, or action.

As one Retrieval plugin may depend on properties provided by other Retrieval plugins, the PDP has to schedule their execution in the correct order so that all properties can be found. For this purpose, whenever a Retrieval plugin is loaded into the framework, the decision engine creates a dependency graph and runs a topological sort (*TopSort*) [88] over it, which creates an ordered list of properties, reflecting the dependencies of the Retrieval plugins.. From this properties list, the decision engine then derives the actual execution sequence of the Retrieval plugins. The dependency graph is constructed by creating a vertex for each attribute and an edge uv if the Retrieval plugin for attribute V depends on attribute U. If for one of the dependencies no corresponding Retrieval plugin is available, or the TopSort algorithm detects a dependency cycle, then the respective Retrieval plugin cannot be loaded because its dependencies cannot be fulfilled. Otherwise, at the time of answering an access request, the computed sequence of Retrieval plugins is invoked and gradually adds properties to the class expression describing the access request.

Referring back to the example above, we assume a `LocationRetriever` plugin which provides the `isLocatedIn` property based on the IP address of the entity, i.e. it depends on *hasIPAddress* and provides *isLocatedIn*. As the *hasIPAddress* is already provided by the PEP, the dependencies are fulfilled and the plugin is invoked, looks up a database to determine the location of the IP address, and returns it in form of a semantic property $\mathit{isLocatedIn.Germany}$, for example. After the description of an entity has been extended in this way, the domain model can be used to reason over it, i.e. new facts about the entity can be inferred from the model and evaluated against a policy, as shown in Figure 6.5.

6.3.1.3. Decision Tickets

When deciding access requests, the decision engine issues a *decision ticket* to the PEP which can be used at a later time to refer to the decision. The purpose of this is twofold: on

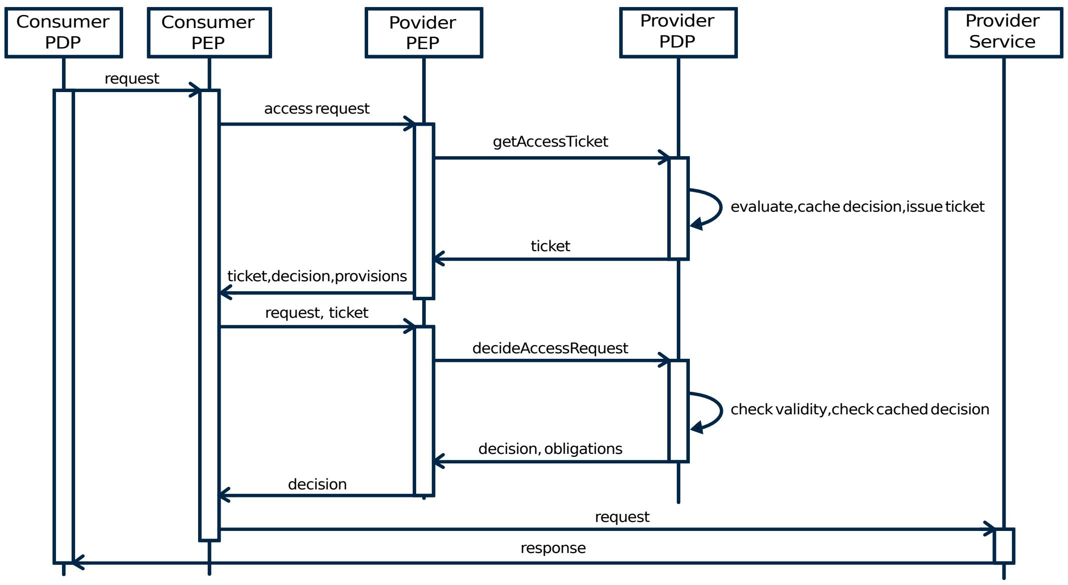


Figure 6.6.: PDP issues tickets for decisions

the one hand, tickets can be regarded as a session, i.e. it is possible to cache a once taken decision for a certain timespan or a certain amount of accesses, so that it is not required to conduct the (possibly resource-intensive) policy evaluation process for every single request. On the other hand, tickets contain a list of provisions which have to be executed by the consumer before access is granted. So, when a consumer wants to access a service, the policy framework allows the PEP to first contact the PDP and retrieve a ticket containing the list of provisions. Only when the PEP has executed the provisions, the prerequisites are fulfilled and the PEP can send the actual access request, combined with the ticket received before. The PDP will then be able to refer back to the taken decision and grant access if the provisions have been executed correctly. A typical use case for such provisions is to require encrypted transmission of a message, or to ask the user to explicitly allow an action.

To prevent service consumers from issuing tickets on their own, the authenticity of a is ensured by appending a MAC using the secret key of the PDP. The sequence diagram in Figure 6.6 illustrates this process.

At the application layer, this protocol is completely invisible, i.e. an application developer can simply made a request to a remote service and expect an answer from it, without worrying about the exchange of tickets. The protocol is part of the PEP implementation and is thus transparently integrated into the middleware's service layer, as further described in the prototype section 7.1.

6.3.1.4. Composition of policy domains

In contrast to many existing policy frameworks, the architecture proposed in this thesis allows a single resource to be under control of multiple policy domains – a typical situation in pervasive systems where devices roam across different locations and are used in multiple scenarios in parallel, as for example a smart phone that is simultaneously used in a personal and a company domain. The approach we propose for this is to annotate access requests with the result of a meta policy evaluation, indicating how individual parts of the decision may be merged with those of another domain. For this purpose, Annotation plugins can be registered in the framework and are then invoked by the decision engine, as shown in Figure 6.3.

However, in many cases it will be important for a PDP to know with which domain it is going to be combined. That is, policy authors can use meta policies to state that only trusted domains are allowed to overwrite the decision of their own domain. So, the framework must provide a mechanism to retrieve descriptions of foreign domain, which can be used by an Annotation plugin to decide if and how a decision may be combined with that of the foreign domain. This is done by setting up a *composition session* between the respective PDPs, by issuing access requests with a specific "compose" action, indicating the initiating domain's PDP as resource and the union of the composed PDPs as subject. We illustrate this by the following example.

Consider that a service from domain C combines services A and B to a "value-added" service, so this value-added service has to comply to the policies of A, B, and C likewise, where possible relaxations for the sake of composition are conceivable. By sending the following access requests, the service S opens composition sessions with each of them.

$$\begin{array}{rl}
S \rightarrow PDP_B: & req\,\langle PDP_A \sqcup PDP_C, PDP_B, compose\rangle \\
PDP_B \rightarrow S: & id_B, dec_B \\
S \rightarrow PDP_A: & req\,\langle PDP_B \sqcup PDP_C, PDP_A, compose\rangle \\
PDP_A \rightarrow S: & id_A, dec_A \\
S \rightarrow PDP_C: & req\,\langle PDP_A \sqcup PDP_B, PDP_C, compose\rangle \\
PDP_C \rightarrow S: & id_C, dec_C
\end{array}$$

Each PDP evaluates the request and sends back a ticket, containing a decision dec_i and a unique random id which refers to the decision. Just as any other access request, the composition request is semantically uplifted and thus, information about the composition partner can be taken into account by the decision plugin when evaluating the policy. So, in a practical implementation it would make sense to provide the *subject* part, i.e., the information about collaborating domains in form of a DL class expression. While this would be straightforward to implement, it assumes that the requester (in this case, service S) is trustworthy and does indeed provide correct information about the composition partners. If this is not the case, the requester could provide false information and thereby trick PDPs into joining a domain composition which does not comply with their policies. To avoid this, it would be possible to point to an external trusted ontology which contains the description of the partners and would allow the PDP the verify the authenticity of such information.

As the value-added service is accessed, decision requests are sent to all three PDPs, referring to the received ids. Each PDP takes a decision and annotates it, as described in section 5.5. PDP_c merges the annotated decision and instructs its PEP(s) to enforce it accordingly.

So, the policy domains can be composed using the existing access request protocol and in general, no additional methods or interfaces are required and users can control domain compositions solely by writing appropriate policies. In the aforementioned example, the composition is dynamically triggered by service S, but it is of course conceivable that this is done in a manual way, via some administration interface.

6.3.1.5. Knowledge base

At the semantic layer, the PDP is responsible for maintaining high-level policies and domain-specific knowledge for the sake of reasoning over policies and their possible effects. It therefore needs to manage different ontologies in a *knowledge base* and provides interfaces for *querying and reasoning*. The knowledge base is made up by a main ontology which contains concepts and individuals from the core policy model and can be extended by further domain-specific ontologies at run time. When a policy module which contains an extension to the policy model is loaded, the respective ontology of the module is merged into the main ontology and checked for consistency. Unique namespaces per module ensure that concepts are not doubled and the monotonicity property of description logic avoids that previous reasoning results are invalidated by the newly added ontology. Only if the merged ontology is consistent and satisfiable, the extension module is accepted and can be used by the decision engine. The framework includes a description logic reasoning engine which provides the basis for evaluating and analysing policies and acts as an interface for accessing the ontologies. It provides an API for ontology management and information retrieval. Developers of extension modules can use this API to evaluate queries against the ontology and provide explanations to the author, like for example answering what-if-queries ("what would be the result if an access request X would arrive in context Y?"), checking integrity constraints ("ensure that role X and Y cannot be both active for the same user"), or analysing the cause of possible decisions ("Under which conditions would an obligation X be executed?"). Because they depend on a specific policy model, such analysis components are not part of the PDP itself, but are rather implemented by *Validator* plugins which are provided by the modules.

6.3.1.6. Module management

At the decision layer, the *module management* is responsible for loading, registering and managing the extension modules by which expressiveness and functionality of the policy framework can be adapted. Modules bundle classes which implement the interfaces of different extension points of the framework, as explained below in 6.4, and can be loaded into the framework at run time. When a module is added to the framework, it is registered at the module management component, following the whiteboard pattern [127]. The whiteboard pattern can be regarded as an advancement of the traditional listener (also: observer [61]) pattern which expects a listener component (e.g., a policy decision algorithm) to directly register at a provider component (e.g., the PDP) and thereby raises some problems in dynamic architectures: when using the listener pattern, one has to make sure that the provider component is available when the listener is loaded and is not replaced by a different component at run time. This would make dynamic updates of the PDP complicated, as it would require every listener to implement functions for handling a dereferenced provider object, so that a flawless update capability of the PDP would depend on proper implementations in each and every decision module. Vice versa, replacing or removing a policy module would require a tear-down function of the module to properly unregister it from the PDP so as to avoid stale references. In case this function has not been implemented or it cannot be executed due to a crash of the policy module, the PDP would still keep the reference to the dysfunctional module, preventing the garbage collector from removing it from memory and possibly trying to invoke it during a policy evaluation resulting in an error and undefined state. Using the whiteboard pattern, the overhead and error-proneness of handling these situations explicitly in listeners and providers can be avoided. Instead of directly binding a listener to a provider, the module management component acts as a registry in between the policy modules and the actual PDP and bundles all management functionality. Whenever a policy module is loaded into the framework, it is registered by the module management and can be accessed by the PDP. There is no need for each module to implement any registration handling, as it is automatically discovered by the module management and registered by the interfaces it implements. Also, the PDP does not have to handle dynamic registrations of policy modules but rather retrieves all alive references to the appropriate modules from the module management whenever it needs to access one of the extension point interfaces.

6.3.1.7. Administration component

Furthermore, an *admin sub-component* provides methods for controlling the framework through a user interface. These include loading and unloading of policy modules, configuring settings of the policy framework and managing the knowledge base. So, when writing rules, policy authors will mainly interact with the management sub-component within the PDP. Basically, there are two way how to write rules:

First, users can directly modify the knowledge base, i.e. retrieve an ontology from it, change or modify facts and concepts in it and load it back into the framework. Especially when large changes have to be done, for instance when the domain model has to be created, this approach is most suitable. Ontology authoring tools help to keep track even of large ontologies and are freely available, such as TopBraid Composer[1] or Protégé[2] which has been mainly used during the development of the prototype.

However, not all users will be familiar with such knowledge modelling tools and also, having full control over the model bears the risk of erroneously breaking it. Therefore, the second way of writing policies is through templates which allow users to write rules in a syntax that is close to natural language and is then mapped by the template engine to the actual statements in OWL or another ontology language. Each policy module that brings its own model may additionally provide a template which is filled in by the user and translated back into OWL by the GUI and loaded into the policy decision point. This way, it is possible to write policies as a mixture of template and Manchester DL syntax which are much closer to the actual intention of the author and easier to understand in hindsight, like for example:

"`permit` **operation** `login` **to** `CorporateWifi` **for** `Smartphone THAT hasOperatingSystem Android2.3.4`"
(Template parts in **bold**, Manchester DL in `typewriter`).

The actual user interface is not part of the policy decision point but rather interacts with it remotely, so it does not necessarily have to run on the same machine as the PDP and can for example be integrated in a central management console. We will present details on the exemplary implementation of a console and a graphical policy IDE in the prototype section 7.1 below.

6.3.2. Policy enforcement

Policy Enforcement Points (PEP) are, as their name suggests, responsible for the actual enforcement of policy decisions. This comprises submitting access requests to the PDP, the enforcement of access control decisions, and the execution of obligations. In this section, we will now look into the details of their software design and discuss integration aspects.

PEPs are realised as software components which are integrated at specific points into an existing pervasive system middleware and are registered at one or more PDPs whose decisions they are obliged to enforce. The registration is done by calling the `registerPEP` method of the respective PDP and can either be done programatically or via the administration console. When a PEP is registered at the PDP, it is assumed that some trust relationship between PDP and PEP is established, so that both can rely on the authenticity

[1]`http://www.topquadrant.com/products/TB_Composer.html`
[2]`http://protege.stanford.edu/`

of messages received by the other and PDPs can assumes that PEPs reliably enforce their decisions. While the first aspect can relatively easy be covered by means of message signing and a respective key management using PKIs or Web of Trusts, the latter aspect might be more difficult to realise in some scenarios. Profound, but also heavyweight solutions that guarantee trustworthiness of a PEP towards the PDP can be realised, when a hardware root of trust is assumed to be available on the PEPs' platforms, such as a Trusted Platform Module (TPM) that attests the integrity of a PEP. However, in most cases trustworthiness of PEPs will be derived from placing them within the PDP's trust domain PDP (e.g., by running it on a platform that is under control of the domain owner). If both options are not viable, an alternative is the use of monitors in order to assess a correct enforcement.

Enforcement is split into two categories: access control enforcement and obligation enforcement.
The former is straightforward, assuming that the PEP is integrated in a place where it can intercept and drop requests to resources, if required. In a service-oriented architecture, this would typically be a service handler chain through which requests and responses have to pass. An alternative are service-wrappers which proxy each service with by a PEP, as it has been realised in the OSGi-based prototype below. While the policy framework cannot dictate a specific way to integrate PEPs into the underlying for intercepting service calls, it is feasible to assume that the middleware provides some way of achieving this – either in the form of handler chain or by hooks and proxying mechanisms.

Obligations Obligations, in contrast, require for a more sophisticated architecture to be enforced. At an abstract level, obligations refer to actions a peer has to fulfil before an access request is granted. At a technical level, this refers in the end to executing code either at the PEP's platform or even at some remote platform, whereas the result of the execution can be the adaptation of configurations, any kind of user interaction or any additional negotiation protocol that is carried out between consumer and provider. Thus, enforcement points must ideally be able to dynamically load obligation code, verify its integrity, execute it and hand its results back to the PDP. In the context of the OSGi-based prototype from section 7.1, this is realised by loading obligations in form of OSGi bundles either from a local storage or even from a trusted remote repository. Integrity and authenticity of such bundles is assured by verifying that the respective Jar files have been signed by a trusted authority, just before executing their code. If a remote platform shall be instructed to execute an obligation, the prototype architecture allows to do so by requesting the respective PEP via R-OSGi to load and execute the bundle. At the generic architectural level that we consider in this chapter, everything that is required is that this code complies to a specific `IExecutableAction` interface. Also, it makes sense if the PEP is able to dynamically load code, as this way it is not limited to hardcoded obligations but can rather execute even before unknown protocols by retrieving the respective implementation from a remote repository.

6.3.3. Communication layer

PEPs intercept outgoing messages at the consumer's side and incoming messages at the provider's side. Apart from the service communication between consumer and provider, PEPs need also to communicate directly with each other over a separate channel, for example in order negotiate and agree on provisions before the actual request is forwarded to the provider. The existence of such a *control channel*, separated from the actual *data channel*, is especially important for multi-step negotiation protocols like the one described in section 7.5. In the framework architecture, this is realised by a *communication layer* which interconnects components and is able to transport messages over an out-of-band channel,

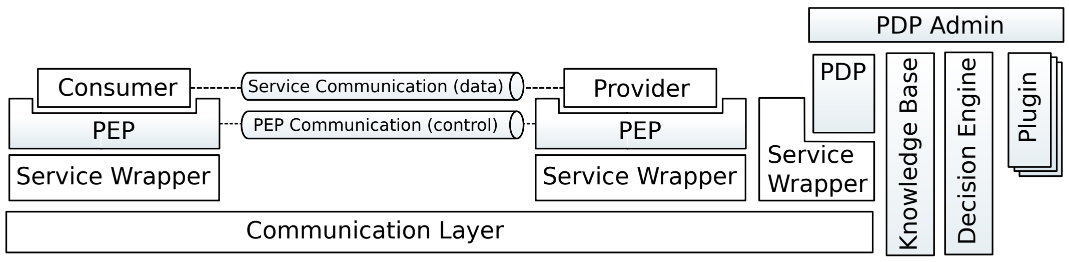


Figure 6.7.: Integration of services (green), framework (blue), and middleware (red)

without interfering with the communication payloads. The communication layer of the framework is responsible for transporting events, coming from the underlying pervasive system middleware, as well as for providing logical channels between the different framework components, such as between PDPs and the administration components, or between PEPs, as illustrated in Figure 6.7. In practice, there are different ways to realise such an out-of-band communication.

Firstly, meta-data can be transported within the actual payload protocol. If components communicate using the SOAP protocol, for instance, policy-relevant information can be integrated into messages using SOAP headers. The advantage of that design is that no new protocol is required and that the policy framework can communicate seamlessly over a protocol that is already supported by the middleware layer. In fact, policy extensions for SOAP like WS-Policy [181] are using exactly this communication channel. However, the drawback of this approach is that it requires the payload protocol to support meta-data. Many protocols do not support this at all and others, like SOAP, only in optional, feature-rich implementations, e.g. Apache Axis, or Microsoft's WSE, as opposed to the more lightweight kSOAP which does not use SOAP headers.
Secondly, the communication layer can be realised by a separate policy-specific protocol layer. Here, the advantages will probably outweigh the disadvantages in most cases: while the drawback of a specific "policy protocol" is that an additional protocol stack is required and thus, the overall footprint of an application increases, it makes the policy framework independent from the middleware's and application's communication layer so that it is for example possible to change protocols in the application without adapting the policy framework itself.

Both options are supported by the policy framework, as components communicate over a `Channel` interface which provides generic methods for sending and receiving messages, independently from the actual implementation. For the prototype realisation, we have opted for the second option — the usage of a dedicated policy communication layer, based on R-OSGi, as it provides the greatest flexibility and allowed us to test different service communication protocols without modification of the policy framework.

6.3.4. Event Mechanisms

So far, we have covered how the policy framework can be administered, how the decision engine reacts on access requests and events, how decisions are enforced, and how the components of the policy framework can communicate with each other. What is still missing is the aspect of gathering, transporting, and processing events.

The framework must be able to react to all kinds of events, for example sensors gathering information about a user's environment, network events indicating the availability of new devices, and middleware events indicating the installation of new components. For this purpose, the framework can be connected to the event transportation mechanism of the underlying middleware. The event buses provided by typical pervasive systems middlewares are usually based on a publish/subscribe pattern. While there are different variations of this pattern, such as topic-based, content-based, or type-based publish/subscribe [53], this differentiation is not relevant for the architecture of the policy framework, as we assume that the framework simply receives all events, which are policy-relevant. For example, the practical prototype uses the publish/subscribe mechanism of the OSGi event admin service, and simply subscribes the PDP to all topics at startup, so that all events are routed into the PDP at first, and it is up to the policy decision process to determine the events to react upon.

Events are then received by the PDP, which forwards them to all registered event adapters and thereby starts the decision process. In which way events shall be processed exactly depends however on the specific use case. While in some cases, it will suffice to deal only with a few, dedicated events, other use cases will require for processing of large volumes of events and for detecting complex patterns in the event streams. In order to be universally applicable in different applications, the framework should not dictate any of these approaches but rather provide basic functionality for the most used event processing mechanisms and apart from that, offer an extension point where developers can plug in their own mechanisms.

So, the event engine of the PDP itself is merely responsible for orchestrating event adapter plugins which do the actual event matching, according to the policies. Two event adapters are contained in the core framework architecture: a basic event adapter and a CEP event adapter. By using the basic plugin, users can write policies which refer directly to the event's topic and which are immediately triggered when the respective event arrives. The CEP plugin, in contrast, applies complex event processing (CEP) in order to detect patterns in the stream of arriving events, which are described by the more comprehensive event model, introduced in section Section 5.4. It therefore maps events of different topics or types (depending on the specific publish/subscribe mechanism in use) to different event streams, for which policy authors can define patterns using the event model. Once the CEP engine has detected one of the patterns specified in a rules, it calls back the decision engine which then continues with the evaluation of conditions, and possibly the execution of the actions of the triggered rule.

6.3.5. Security considerations

It is the goal of the framework to "policy-enable" an existing pervasive system middleware. Because the framework itself is realised by a set of middleware services, it will be able to use all mechanisms for communication security and component authentication which are available at the middleware layer and are also used by other existing services. Firstly, this reduces the footprint of the policy framework because it removes the need to redundantly implement all kinds of encryption, authentication, and user authorisation mechanisms which might already be available in the middleware. Secondly, this allows for a seamless integration, as the policy framework does not dictate specific mechanisms or trust models which might not be compatible with the ones used by the application, for example. Nevertheless, for the case that the middleware does not provide the required security mechanisms, the framework provides callback methods which allow the application developer to plug in the missing functions.

In the following, we will discuss the points developers have to consider when integrating the policy framework into a middleware, in order to guarantee an adequate level of security.

Authorised policy administration Policies must only be modified by authorised users. Otherwise, it would be possible for attackers to connect to the PDP and set up policies granting them access or, even worse, tamper with the system's integrity by instructing PEPs to execute any obligation. For this reason, all administration functionality has been encapsulated in the `PDPAdmin` interface which can be protected when integrating the policy framework into the middleware. It is recommended to restrict access to this interface to locally connected clients only and to authorise users by a username/password combination, or preferably, by client certificates.
In addition, it is possible to apply access rules to the `PDPAdmin` interface, so that administrators can regulate access to the policy framework by using the framework itself. On the one hand, this gives a great degree of flexibility in determining access conditions to the `PDPAdmin` interface. On the other hand however, this bears the risk of locking oneself out of the administration service if the policy is flawed and in most cases, a simple user authorisation will suffice. Service middlewares like OSGi come with user administration components which could easily be used for this.

Trust relationships between the components of the framework Not everybody should be able to issue policy decision requests to the PDP. Otherwise, the PDP could be misused as an oracle which allows attackers to extract main parts of the policies in use by systematically sending decision requests to the PDP. Also, limiting access to the PDP's decision interface will prevent malicious PEPs from starting Denial of Service (DoS) attacks against the PDP by sending large amounts of decision requests. So, a trust relationship has to be established between PEPs and their PDP. While in some cases this might be done when deploying the policy framework, we also have to consider that PEPs may dynamically be added to a domain and the trust relationship has to be set up at runtime. This trust relationship is established whenever the application registers a PEP at the PDP. At this point, the PDP receives a reference to the remote service interface of the PEP and remembers it for later checks against the incoming decision requests. It depends on the service middleware if any additional authentication must be added at the application layer. If the RPC protocol authenticates communication endpoints, for example by running over TLS with client certificate authentication, the application can simply register PEPs and does not have to provide any further mechanisms. Otherwise, the PDP provides a callback interface `RegistrationListener`, in which an authentication protocol can be implemented at application layer.

Communication security It must neither be possible for an attacker (i.e., any component which is not a legitimate PEP) to fake decision requests or decision tickets, nor to replay a previously recorded decision ticket in order to refer to a previously made policy decision. Sending fake decision requests is prevented by the aforementioned registration of trustworthy PEPs at the PDP. However, faking decision tickets or replaying recorded tickets would be possible, if the policy framework would not apply some additional protection mechanisms: when the PDP issues a decision ticket, it includes a unique id (such as a current timestamp) into the ticket which acts a proof of freshness. Only if the ticket has been issued for the requesting PEP, is properly signed and still valid, the PDP will accept it and return the previously taken decision.

Trustworthiness of dynamically loaded code The framework allows PEPs to dynamically load code from remote repository and to execute it as part of an obligation, for example. This bears the risks that the loaded code has been injected by some illegitimate outsider

or that is behaves maliciously in other ways. For this reason, it is necessary to guarantee the authenticity of the remote repository and the integrity of the remote code packages. Authorising the remote repository is a task which will usually be done at the middleware layer, by means of a traditional secure communication channel with certificate-based server authentication. Checking the integrity of loaded code is done by signing code packages before uploading them to the repository and verifying the signature at the PEP, before executing the code.
Not supported is however an automated checking of the semantics of the loaded code. Automatically identifying the purpose of a generic piece of code is an unsolved challenge and goes far beyond what would be achievable (and reasonable) within a policy framework. Thus, the repository must be set up in a way that only legitimate users can use it to provide dynamic code to the PEPs.

6.4. Policy modules

First and foremost, a framework has to be extensible in an easy way. Thus, the policy framework proposed in this thesis provides a set of extension points which can be implemented by *modules*. A policy module is a software component which contains multiple subcomponents, so-called *plugins* and can be loaded into the framework at runtime. The registration of a module is done using the whiteboard pattern, i.e. it is not necessary to take care of the registration within the module itself, but rather the `PDPAdmin` component is notified by the framework whenever a new module becomes available. It then retrieves all plugins from the module and registers them at the corresponding extension points. So, developers writing a new module for the policy framework merely have to provide implementations of the extension point interfaces and leave the management work to the framework.

In general, developers can bundle as many plugins into one modules as they wish. However, it makes sense to create separate bundles per functionality, because it will allow users to switch on only those modules which are required for a specific application. For the exemplary pervasive systems extensions presented in the subsequent chapter, we have chosen to implement each policy model in its own module. This way, users will be able to adapt the framework's functionality by simply adding or removing modules.

In the following section, we will introduce the individual extension points in detail and subsequently briefly discuss the collaboration between modules.

6.4.1. Extension points

In the following, we will give an overview of the framework's extension points at which a module can register its plugins. The registration is done by a whiteboard pattern, so the module itself does not have to care about its integration into the framework – as long as it provides implementations of the specific interfaces, they will be found and registered by the PDP. Figure 6.8 provides an overview of the extension points and how they integrate into the different framework components.

Configuration The plugins contained in a module must of course be configured. In order to make their configuration accessible via the central `PDPAdmin` interface a module can provide a list of settings which are automatically registered in the PDP's main configuration when loading the module. A setting is a simple key/value pair, combined with an

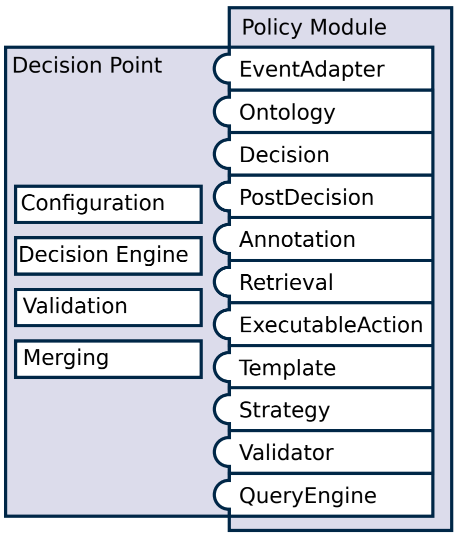


Figure 6.8.: Policy Modules (r.) extend the functionality of the decision engine (l.)

explanatory description which can be displayed to the user at the command line interface, for example.

Retrieval Retrieval plugins contribute to the Decision phase by providing a mechanism to add semantic information to an entity, i.e. a Subject, Resource, or Action. Each Retrieval plugin has lists of required and provided attributes and contains a `retrieve` method which gets the DecisionContext object and retrieves values for certain attributes – either by reasoning over the already existing description of an entity, by querying some external information source, or simply by assigning some hardcoded value. The result is a `SemanticObject`, which is able to represent the entity as a description logic class expression and is used by the decision engine to evaluate the entity against the policy. When a module containing Retrieval plugins is loaded into the framework, the PDP tries to resolve the dependencies of the contained Retrieval plugins using the algorithm explained in the section on semantic uplifting (cf. pp. 106) above. It is however possible that circular dependencies between Retrieval plugins exist – in that case the Retrieval plugin is loaded but the framework issues a warning. When a policy decision is requested but the dependency still has not been resolved, the Retrieval plugin is not invoked and as a result, some attributes of an entity might not be available, thereby resulting in an erroneous decision. Usually, this is unwanted as it could for example lead to subjects gaining illegitimate access to resource, so this situation should lead to a safe default decision (e.g., deny access and log the event). While this is not predetermined by the framework's architecture, it would be advisable to make the default behaviour in that situation controllable using the framework's configuration feature.

Decision After attributes have been collected using the Retrieval plugins, they need to be evaluated during the Decision phase. The evaluation functionality is provided by Decision plugins, which may add operators, or even implement a whole decision engine in itself. Operators are needed to evaluate conditions over attributes of entities or events, such as $age \geq 18$ for example, and are mapped to the values of properties in the model (e.g., `:booleanOp :gt`). Operators provided by one Decision plugin can be used by all other plugins as well, so it is possible to extend an existing model by further operators without modifying its decision engine. Besides operators, Decision plugins are also the place to implement the actual decision engine, i.e. the logic for evaluating any policy model which goes beyond the semantics of the simply core model and its evaluation using OWL features.

So, Decision plugins allow users to create their own policy model from scratch by implementing a dedicated evaluation algorithm which is not necessarily based on the predefined concepts. Each such plugin receives the access request and its configuration, processes the request, e.g. including one or more Retrieval plugins, and returns a decision which consists of either permit or deny, along with a set of obligations which have to be executed before the decision is enforced.

PostDecision PostDecision plugins contribute to the PostDecision phase. They cannot modify the decision anymore but can take additional actions such as creating a history over passed policy decisions or creating audit trails. That is, a PostDecision plugin retrieves the DecisionContext object, processes it but does not return any results. It has access to the knowledge base, so a plugin for creating audit trails would be able to use the reasoner in order to retrieve explanations of the taken decision and store them along with a signed timestamp.

Annotation Annotation plugins are responsible for evaluating meta policies and annotate a policy decision with their result. If a module implements this interface, the Annotation plugin is registered in the framework and invoked during the Annotation phase. It receives the Decision Context object, gets access to the knowledge base and is expected to add an annotation to the Decision Context object.

Ontology Fragment Each module may provide an ontology which will be added to the knowledge base of the PDP. The axioms added by the ontology are accessible by the plugins of all modules and can thereby extend the existing policy model. For instance, the concept of *rights delegation* can be supported by adding an ontology which declares subject B to be a subclass of subject A, thereby effectively delegating all rights from A to B.
In terms of Description Logics, the ontology provided by a module will in most cases represent a TBox, i.e. a set of concepts and relations within the namespace of the module. The TBox is included into the knowledge base of the PDP and policy authors can populate it by specifying instances providing application-specific information.

EventAdapter As described in chapter 5, not only access requests require for the evaluation of polices, but also asynchronous events can start the evaluation process and trigger actions. An event consists of a topic and a topic-specific set of key-value pairs. Policy modules which need to react on events can contribute EventAdapter plugins to the framework an register them in the PDP by their topic. Once an event arrives at the PDP, all registered event adapters are called in the order of registration. They evaluate the event against a policy fragment defining reactive policies and initiate the execution of actions, if required.

ExecutableAction Policy decisions can include obligations, i.e. actions which the enforcement point is obliged to take – either as a result of an access request, or triggered by a reactive policy. Obligations can range from very simple actions, such as logging an access request, on to complex processes and user interactions which are have to be carried out. Whenever a PEP receives a policy decision, either triggered by an access request or asynchronously by an event, it will try to execute the contained obligation by means of a dynamically registered ExecutableAction plugin and only if an appropriate plugin was available, the respective action could be executed and returned a positive result, the PEP confirms the obligation so that either the access request can pass or the asynchronous notification is approved.

Templates In general, the decision engine operates on semantically represented policies in the knowledge base. Users can thus write policies straightforward in form of an ontology. This might however not be well-suited for users who are not proficient in semantic modelling and bears the risk that a user accidentally modifies not also the ABox, but also the policy model itself. So, some users might demand for alternative ways to write a policy and prefer a simple Domain Specific Language (DSL).
While it is of course conceivable that external tools translate any model-driven policy specification into an ontology, the framework itself provides a template mechanism which allows developers to write policies in a simpler syntax. For this purpose, module developers can provide a template of a human-readable syntax which is translated into OWL by a template engine before it is loaded into the knowledge base. The template file must be named `template.vm` and placed in the root directory of the module. It must contain an identifying name, as well as two sections: an *input* section and a *rendering* section. The input section defines the template of a human-readable syntax and variables which have to be filled by the user when writing the policy, and the rendering section defines the ontology output which is created from the specified variables, i.e. all variables which are present in the rendering section must be present in the input section as well. While OWL as an ontology language will be the natural choice in most cases, the rendering section also supports other languages like the Manchester DL syntax or Turtle. As the prototype uses the Apache Velocity engine[3], both parts of the template file have to be written in the corresponding Velocity Template Language (VTL). However, this choice was only made for practical reasons and other template engines could be used as well. An example template file can be found in Appendix B.

Strategy A strategy is an algorithm for selecting the final policy decision out of a set of potential decisions. It is not uncommon that multiple policy decisions are generated for a single access request or event, for example when multiple modules are activated at the same time or a single module creates multiple decisions. Therefore, the PDP chooses one applicable decision before it enters the Annotation phase, using one of the provided strategies. While the core PDP comes with two simple strategies which either prefer permit or deny decisions, more advanced strategies are of course conceivable, such as preferring more specific obligations over generic ones, for example. Although a module can provide an arbitrary number of strategies, only one strategy can be activated at a time, of course. The activation is done via the generic configuration module and can be set by the user via the command line interface.

Validator A Validator plugin checks if the policy model complies with certain constraints. Firstly, this includes validating the schema, i.e. the structure of a policy, in order to guarantee that all rules are well formed. Secondly, Validator plugins can check side constraints to ensure the policy is consistent in terms of the model. A typical example for this is the Separation of Duty constraint which guarantees that one user cannot be assigned to two conflicting roles at the same time. Checking such constraints manually is almost impossible if policies become sufficiently complex, so it is important for policy authors that

[3]`http://velocity.apache.org/engine/`

the framework provides the means to automatically identify violations of their constraints. It is left up to the implementation of each Validation plugin how it conducts the verification. Nevertheless, two approaches are possible: querying the model to detect possible flaws or modelling constraints directly in the ontology. The latter option is the more rigorous one, but the former option is universally applicable and does not require modifications of the model. Validator plugins get access to the reasoning API and can use it to query the knowledge base. They are not part of the decision process and thus not registered for any of the phases, but are rather used when authoring a policy, i.e. from a modelling GUI or from the framework's command line. When a Validator has run its checks, it either confirms that the policy is flawless or returns an explanation of the constraint violation.

QueryEngine Query engines can be used to include queries to external databases or the knowledge bases into the evaluation of a condition. A query engine accepts a query string as input parameter and is expected to provide a set of key-value attributes as a result. Using this extension point, policy modules can for example allow policy authors to write conditions against an SQL database or a Prolog fact base. Further, RetrievalPlugins may make use of existing query engines. However, the core framework does not specify how plugins shall directly communicate with each other, so this is left to the module developer's responsibility.

6.4.2. Collaboration between modules

Often, a module will provide functionality which is also useful for other modules. For example, as just mentioned, a QueryEngine plugin for retrieving items from an external database might be intended for the evaluation of conditions based on the contents of that database, but it might also be helpful for a Retrieval plugin which could use it to retrieve information about an entity from that database and add it to the entity's semantic description. Fostering the reusability is thus an important criterion to make the framework a benefit for developers.

In general, the public APIs of each plugin can be directly accessed by all other plugins. On the one hand, this allows to build increasingly feature-rich modules but on the other hand, it introduces dependencies between modules which must be managed by the developer. There are two ways how modules can depend on each other: first by using information from another module's ontology fragment, as we described above. In that case, the dependency is automatically detected by the framework and developers do not have to care about it. If a developer wants to directly use the API of another module, however, she has to explicitly state that by adding a `Require-Capability <module>` property to their module.

6.5. Summary

In this chapter, the software architecture for putting the core policy model into practice has been introduced. First and foremost, the architecture has been designed with extensibility and easy integration into existing pervasive systems in mind. Policy-enabling an existing system requires in-depth knowledge about the system architecture itself, knowledge about the policy framework, and last but not least about the high-level security policies which should actually be applied. Apart from small and well-defined systems, these tasks will devolve to different user roles and it is unlikely that a single user will be able to carry out each of them equally proficient. The framework's architecture takes this into account by a three layer partitioning – *semantic*, *decision*, and *enforcement* layer – where each layer addresses different user competencies and roles and is decoupled from the other layers as far as possible.

The software architecture described herein does not rely on any language- or platform-specifics and can thus be regarded as a universal approach which could be applied to different platforms like web service based infrastructures, networks of resource-limited embedded devices or a distributed service middleware. One important requirement for the software architecture was the ability to cope with constantly changing environments:

- Infrastructures change as services are added or removed from a domain and different communication channels are in use
- Assessments of security mechanisms change as new mechanisms become available and existing ones may become broken or not sufficiently secure anymore
- Policy models change, either as new applications on top of the middleware layer are created or as the security model of an existing application changes

While most existing policy frameworks are focussed on a single policy model like RBAC and an infrastructure that is known in advance, the modular software architecture introduced in this chapter allows to dynamically load policy modules and thereby extend and modify the policy model at run time. In combination with the model-based approach of describing the application domain and the policies, the framework allows to reflect these changes without the need of re-deploying any software or even changing to a different policy framework.

Policy modules can contribute to various framework extension points, ranging from additional PIP functionality, over additional ontology concepts, to whole new policy models. The following chapter will validate the component-based software architecture by means of a prototype and then introduce some examples of how extension modules can be built upon the concepts of the existing core policy model.

CHAPTER 7

Application to pervasive systems

So far, we have explained the core policy model and the software architecture which will put it into practice. However, the model itself is rather simple and the software architecture is first and foremost extensible, but may not fulfil all application's requirements out of the box. So, in this chapter, we will point out how the framework can be used to extend the core functionalities by increasingly abstract high level policies which solve typical pervasive systems challenges.

All extensions presented herein have been realised in form of prototypical extensions for the policy framework and thereby served as tests validating the afore presented extensible software architecture.

At first, in section 7.1 we will describe the prototype realisation of the main framework architecture, as well as interfaces for user interaction in form of a command shell and a graphical user interface (GUI). Based on this prototype, multiple extension modules have been designed and realised, each solving a specific pervasive systems problem. The first extension for a dynamic role-based access control model in section 7.2 shows how the universal concepts of the core model can be extended and combined so as to formulate increasingly abstract high-level policies.
Section 7.3 further extends this model on to an even more abstract model for so-called *Situation-Goal policies*, a novel policy pattern based on reactive policies, as well as models of situations and protection goals, in order to realise a self-adapting system.
The demands on a policy framework increase if multi-domain settings are considered, where different administrative domains have to collaborate with each other. In section 7.4 we thus describe how the framework can handle these situations. Finally, as a further increase of complexity, we will show in section 7.5 that the policy framework is able to take into account individual preferences of each domain and that it can be used to autonomously negotiate an optimal set of obligations which satisfies policies and preferences of all involved parties.

7.1. Prototype realisation

The afore described software architecture is generic and could be applied to any service-oriented system that fulfils the prerequisites from section 6.1. In order to test the concept and its practical applicability, it has been implemented as a prototype for a typical pervasive systems platform. Here, we describe the choices and experiences made during the implementation and draw a conclusion on the applicability of the architecture, where the most interesting aspects are the dynamic loading of policy modules, a non-invasive integration of policy enforcement points that does not require any modification of the underlying middleware and ways to realise user-friendly interfaces.

7.1.1. OSGi

OSGi stands for *Open Services Gateway Initiative* – a name that indicates where the origins of this technology lie: in 1999, companies like Philips, Ericsson, IBM, Motorola, Lucent, Sun, and others teamed up in order to specify a Java-based software framework for so-called "service gateways", i.e. set-top boxes and embedded home servers. The initiative was focused on embedded devices that would run in a user's home, route traffic between the internal LAN and the Internet and could be equipped with third-party services, e.g. from media companies or financial institutions. The monolithic software platforms that were available at that time did not allow dynamic updates of components in these gateway devices and thus the OSGi initiative was founded to work towards a lightweight "micro-service" middleware that applies the service-orientation paradigm to components within a single JVM. The first proposal was made in form of the Java Community Process by JSR-8 [22] and soon, a number of implementations of the first specifications were available. OSGi began to spread from home service gateways to other embedded systems like in-vehicle software and finally found its way into other application areas beside embedded systems. A major push for the OSGi technology was its adoption by the former Eclipse consortium in the newly founded Equinox project whose aim was to break up the monolithic architecture of Eclipse that increasingly resulted in runtime problems and difficulties to more easily integrate other open source projects into the Eclipse IDE. After an evaluation of available technologies, the Equinox project opted for OSGi to create a modular software architecture that would allow to extend the Eclipse IDE in a plug-in manner and to manage dependencies between individual modules. Since then, OSGi has become an established technology that is applied in diverse software products: it is still used in embedded devices and vehicular applications (e.g., the software for the simTD field trial [8]), but also builds the basis of many large-scale application servers like GlassFish, JBoss, or WebSphere. Nowadays, the specification is driven by the *OSGi Alliance* and implementations for different application areas are available. Most of them are open-source and community-driven, like Apache Felix (former Oscar) or Equinox, some are commercial products like the implementation by proSyst. Numerous extensions have been developed, and today, OSGi cannot only be used as a component layer but rather at a complete distributed system architecture.

7.1.1.1. Main concepts

OSGi is a middleware that runs in a JVM and allows to load, modify or remove components at runtime, i.e. without stopping or restarting the JVM.

Bundles One central concept in OSGi is that of a *bundle*. A bundle is a Jar file, i.e. a specially crafted zip archive which contains a set of classes and a *Manifest* file. The classes implement functionality which the bundle can provide to other bundles in the framework. and the Manifest file contains meta data about the bundle, such as its name, version, and the Java packages which are being imported and exported by the bundle. Only those packages of a bundle that are marked as exportable can be imported and used by another bundle – all other packages are private and can only be accessed from within the bundle itself. OSGi realises this functionality by spawning a separate Classloader for each bundle so that it is (usually) not possible to access classes from a different bundle without the help of the underlying OSGi framework. This should however not be regarded as a security feature, because without an additional Java SecurityManager, it is indeed possible to access internal classes from other bundles by means of some class loading and reflection detours. Bundles have a life cycle, i.e. the OSGi framework manages loading, starting, stopping and unloading of a bundle. When a bundle is loaded, the framework first tries to resolve all dependencies of the bundle, i.e. each packages the bundle imports must either be exported by another bundle or must be directly available through the root classloader. If all dependencies can be satisfied, the bundle is set to the *Resolved* state. In that state, it acts like a library that has been loaded into the classpath of a monolithic Java program, i.e. it does not start any code execution by itself but it provides methods which can be imported and used by other bundles. If the Manifest file references a so-called *Activator* and the OSGi framework is configured to start the bundle, it is further put into the *Starting* state. At that point, the `start()` method of the indicated bundle Activator is called – analogue to the `main()` method of a normal Java program. When the `start()` method has successfully terminated, the bundle is in *Activated* state. Likewise, when a bundle is unloaded from the framework, it is at first stopped by calling the Activator's `stop()` method which may implement some clean up code, and leads the bundle into the *Stopping* state. When the stop() method has terminated the bundle is in *Uninstalled* state and cannot be accessed by other components in the framework anymore. The left part of Figure 7.1 depicts the bundle life cycle.

Declarative services While the bundle concept already provides the means for a component-based software architecture it still bears some drawbacks so that various additional ways to structure components and realise inter-component communication have been developed in the course of time. Loading a class from a different bundle is similar to loading importing a class in traditional Java: it requires developers to know the exact package and name of the class and closely couples caller and callee, i.e. if the imported class would be removed or modified at runtime, the caller's reference would get lost and potentially result in an undefined state. In order to realise a more loose coupling OSGi provides a *service registry*. Bundles can register services, assign a name and properties to them and publish them in the OSGi framework's services registry where they can be found by other bundles. This allows developers to simply get a reference to some implementation of a known interface, without necessarily knowing by whom (i.e., which bundle) the actual implementation is provided, thereby following the SOA paradigm of considering only the required functionality, rather than its location or implementation. However, the service feature requires developers to spend significant efforts on the management of service references which leads to complex and error-prone implementations, e.g. if dereferenced services are not correctly handled. To overcome this problem, *declarative services* have been developed. By declarative services users can configure the provided and referenced services in a declarative way, i.e. by means of an additional configuration file, and the management logic for resolving services references and notifying consumers is hidden in the underlying

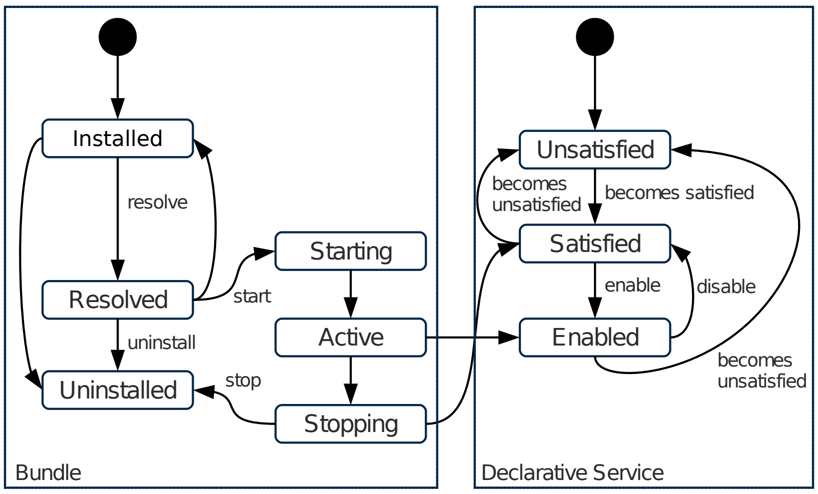


Figure 7.1.: Interleaved life cycle of bundles (left) and declarative services (right)

framework. Declarative services are provided by so-called *components* which have their own lifecycle within the containing bundle's lifecycle: when the bundle has been activated, the component (i.e. its declarative services) can be enabled, effectively calling the enable method of the component's implementation, analogue to starting a bundle by the `start()` method. Figure 7.1 illustrates how the life cycle of declarative services fits into that of a bundle. Thus, using declarative services it is nowadays possible to create component-based applications which handle dynamic updates, additions, or removals of components in a consistent way and do not require the developer to implement additional management logic.

Besides the concepts of bundles, components and declarative services, various other extensions to the original OSGi platform are available, such as Optional and Dynamic Imports, Uses Declaratives, Required Bundles, and Capabilities. Apart from Capabilities, these concepts do not play any significant role for the prototype described in this section and will therefore not be explained in further detail. Capabilities allow to explicitly express dependencies between bundles which are not caused by package imports or any other language-level mechanism. One example is that the policy modules will depend on the availability of a reasoning engine. As they however do not import any packages from the reasoner, the dependency cannot be expressed at service or package level but is rather expressed by a `Provides-Capability:Reasoning engine` statement in the reasoner's bundle and a `Requires-Capability:Reasoning engine` in the policy modules. Only if all required capabilities are satisfied through respective capability providers, the OSGi framework starts the bundle.

Communication As described so far, OSGi is merely an architecture for managing components within a single JVM. For creating a pervasive system test environment, a communication layer that connects OSGi bundles across multiple remote platforms is however essential, including discovery, binding and invocation mechanisms. One typical service communication mechanism is Web Services and the SOAP protocol and indeed, they have been used in various middleware projects like AMIGO [3], LinkSmart[1], or LooCI[2]. An

[1]`http://sourceforge.net/projects/linksmart`
[2]`http://code.google.com/p/looci/`

advantage of Web Services is the comprehensive standardisation, including additional protocols like WS-Policy, WS-Security, or WS-Addressing. However when it comes to practical use, especially in the context of OSGi, available Web Service implementations turned either out as heavyweight, such as the ones based on Apache Axis, or not standard-compliant and therefore not fully interoperable with other Web Services, such as the lightweight kSOAP protocol which does not support SOAP headers, for example. In the context of Equinox, the Eclipse Communication Framework[3] (ECF) aims at supporting different communication protocols for OSGi platforms, among them HTTP/REST, XMPP, and Remote-OSGi (R-OSGi). R-OSGi[4] is a lightweight binary protocol dedicated to the communication between OSGi bundles. It has been created by ETH Zurich and was donated to the ECF later on. The idea behind R-OSGi is to make the distribution of bundles as transparent to the developer as possible. Developers who want to provide a service over the network simply add a property `remote.registration=true` to the service definition and R-OSGi automatically provides it as a remotely accessible service. The R-OSGi framework then uses a discovery component to publish the availability of the service over the network. When a client requests a reference to the service, R-OSGi automatically creates a stub of the remote service interface and sends it over the network to the client. This "remote proxy" implements the service's interface and marshalling code for transferring method calls and parameters over the network. The actual protocol is a proprietary one and aimed at reduced data traffic. Although R-OSGi is focussed on Java interfaces, it also requires not much effort to create R-OSGi services on different platforms like .NET, as long as only primitive data types are used as parameters and return values. Vice versa, implementing R-OSGi clients is difficult on other platforms than Java, because the remote proxy object that is sent from the server to the client upon connection establishment has to be dynamically loaded and called. It is transferred in form of Java bytecode, so that if the client's platform is not able to interpret it, the remote proxy object cannot be loaded. In a prototype implementation of the multilateral policy refinement module (c.f. chapter 7.5) which has been done as part of a thesis [67] supervised by the author of this document, R-OSGi clients have been implemented for the Android platform which work by translating the received remote proxy object on-the-fly from "normal" (i.e., Sun) Java bytecode to "dex" bytecode which can be loaded into the Android Dalvik VM. So, although such translators introduce additional overhead, it is theoretically possible to realise even R-OSGi clients on other platforms than Java.

For discovering services in the network, R-OSGi originally made use of the Service Location Protocol (SLP). Although SLP was designed with ad hoc discovery of embedded devices in mind and should therefore be well suited for a pervasive system test set up, it raised some practical problems such as a non-configurable privileged port number 427 which raised problems with applications running with user rights in Linux systems, as well as some bugs in the jSLP implementation used by OSGi. For this reason, an mDNS- and DNS-SD-based implementation has been done for the discovery capabilities of the prototype. mDNS stands for multicast DNS [32] and has been specified in order to allow DNS name resolution in IP networks without the need to configure a central DNS server. Another extension to DNS, DNS-based Service Discovery (DNS-SD) [31] uses the DNS protocol to publish and query services in a network so that in combination with mDNS, a configuration-less discovery protocol for IP networks can be realised. In fact, the combination of mDNS and DNS-SD make up the Bonjour protocol which is first and foremost used by Apple but is also increasingly applied for discovery of various home network devices, such as media players, for example.

[3]`http://www.eclipse.org/ecf/`

[4]`http://r-osgi.sourceforge.net/`

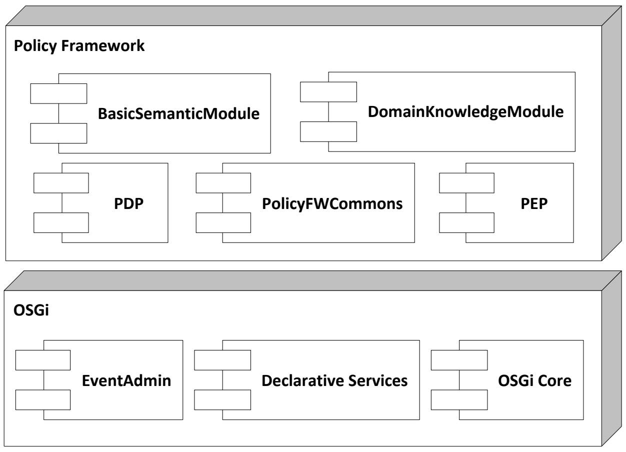


Figure 7.2.: Main components of the prototype implementation

The combination of OSGi, R-OSGi and mDNS provides the basis platform for the prototype implementation. OSGi in general has been proven to be a suitable component layer for pervasive systems and is used in many existing middlewares [37, 6, 139, 65, 199]. It provides service provisioning, discovery, binding and invocation and allows to dynamically add, remove, or modify components and services. While the Equinox framework implementation has been chosen because of its comprehensive capabilities and stability, some tests with more lightweight implementations have been done. For example, some services have been hosted on a Chumby[5] internet radio, using the Concierge OSGi framework running on an ARM9 platform and the jamvm Java machine[6].

7.1.2. Framework components

Based on the OSGi platform, a prototype of the policy framework has been developed. It mainly consists of the components *PDP*, *PEPs*, *PolicyFWCommons*, *BasicSemanticModule*, and *DomainKnowledgeModule*, which implement the functionality of the core framework. Each component is realised as an OSGi bundle and provides its methods via declarative services. As an event bus, the OSGi EventAdmin bundle is used – a simple mechanism for transporting events which are identified by a topic and contain a set of key-value arguments. In combination with R-OSGi, the EventAdmin can be used as a networked event bus in a straightforward way. Other publish/subscribe protocols could be adapted to the event listeners of the frameworks, for example during experiments with the LinkSmart middleware, a SOAP-based event transporting mechanism has been attached and others, such as the MQTT protocol [102] which is currently gaining momentum in the machine-to-machine (M2M) community, could be integrated. Further, a PolicyFWCommons bundle bears all interfaces which are implemented by the publicly accessible components. The PDP component contains the logic of the generic policy decision process and is the central component that serves as a registry for additional policy modules. It provides two interfaces: a `PDP` interface and a `PDPAdmin` interface. The latter one provides methods for managing

[5]`http://www.chumby.com`
[6]`http://jamvm.sourceforge.net/`

the policy framework itself, i.e. for configuring it, inspecting the knowledge base, and (de-) registering policy modules. This interface is remotely published but access to it is controlled by a simple token authentication so only administrators of the framework can use it. The `PDP` interface is likewise remotely accessible and provides methods for policy evaluation and merging. To get access to these methods, PEPs must first be registered via the `PDP.registerPEP()` method so that only legitimate PEPs can invoke the costly policy evaluation. By a call to `PDP.getAccessTicket()`, a PEP asks for the evaluation of an access request and receives a *decision ticket* which can be used to refer to the decision at a later time. Decision tickets identify a session between the PEP and the PDP and refer to a specific access request. Sessions allow on the one hand to cache a policy decision, so the evaluation process does not have to be started over and over again for repeated accesses to the same service. On the other hand, decisions can require the PEP to execute actions (*provisions*) before finally granting the access. Here, a decision ticket is used to identify the previously taken decision, once all required provisions have been fulfilled.
Decision tickets are handed to the PDP via the `PDP.decideAccessRequest()` method which evaluates an access request against a previously taken decision, if the ticket is still valid. In addition, the PDP offers a method `mergeDecisions()` which can be used by registered PEPs to merge policy decisions from different PDPs in a multi-domain scenario.

When evaluating an access request, the PDP merely coordinates the loaded policy modules and the plug-ins they provide. While existing policy languages could easily be integrated into the framework by wrapping their implementation in a module, the focus of this thesis is to leverage semantically represented policy models. Thus, a BasicSemanticModule provides the necessary ontology management, reasoning capabilities, as well as the basic ABAC and ECA model from section 5.2. As a reasoning engine, Pellet 2.2.2[7] has been chosen, because among the freely available reasoning engines, it is the one which sticks the closest to the semantics of OWL2, i.e. the $\mathcal{SHOIQ}(D)$ logic. Pellet supports SPARQL ABox queries in conjunction with the Jena library, interprets different ontology formats such as Manchester DL syntax, Turtle, or OWL2, allows ontology modification at run time by means of the OWL API[8] and is able to apply so-called DL-safe [64] SWRL rules. Although Pellet showed satisfying performance [163], other tools like the Sesame library from OpenRDF[9] might provide better results for query response time and ontology updates, but it operates on RDF semantics only.

The concepts of the ABAC and ECA models are represented in an ontology in Turtle syntax based ontology and provided to the main PDP as an *OntologyFragment*. In a decision plugin, the actual policy evaluation code using the reasoning engine is contained and further, an event adapter is registered at the PDP for receiving data over the event bus and checking it against the ECA policies.

7.1.3. Integration of PEPs

PEPs enforce access control decisions and obligations (respectively, provisions) received from the decision point. For that purpose, they must be able to intercept service invocations in the middleware and receive event-triggered notifications from a PDP. It is important that all method calls are in fact intercepted and the PEP cannot easily be circumvented. In most web service stacks, dedicated *handlers* can be attached to incoming and outgoing service messages and log, modify, or drop messages. By concatenating handlers to a handler chain, subsequent transformations like authorisation, encryption, and signature

[7] `http://clarkparsia.com/pellet/`
[8] `http://owlapi.sourceforge.net/`
[9] `http://www.openrdf.org/`

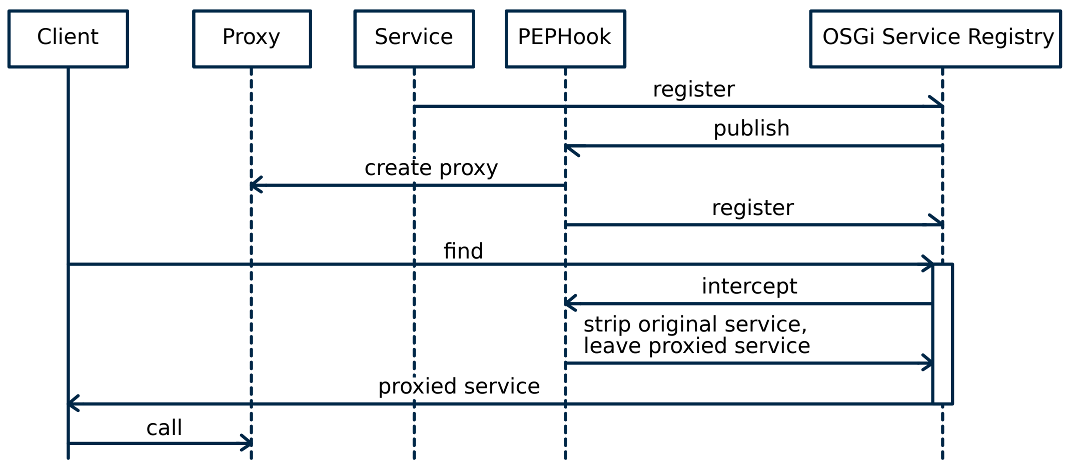


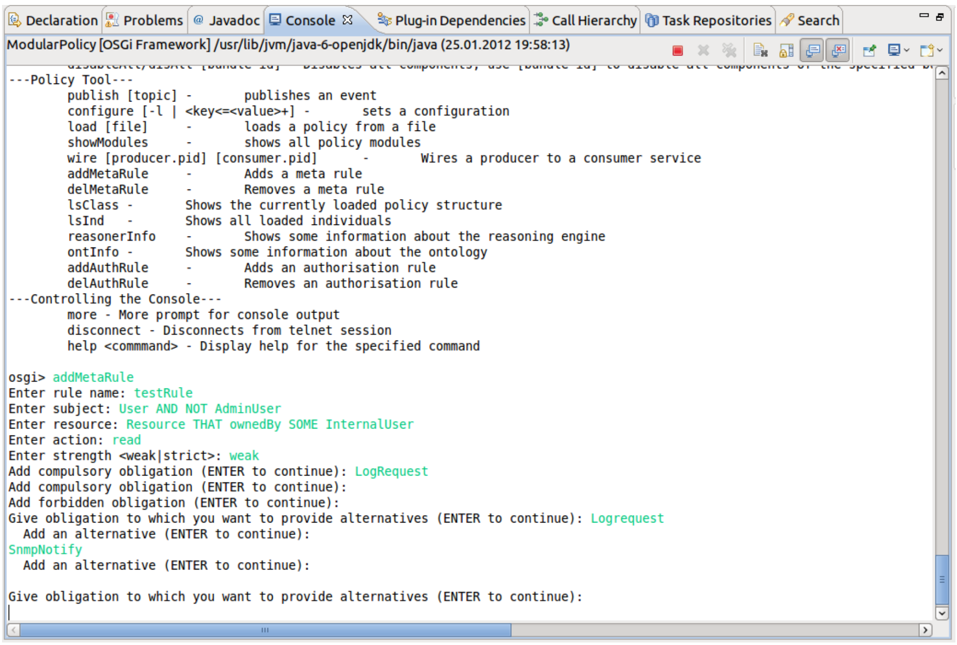
Figure 7.3.: Sequence of hooking into service registrations and replacing services by PEP proxies

can be applied to a message, for example. However, pervasive systems are often based on lightweight architectures, and no such handler mechanisms are available, let alone complex message transformation mechanisms. The R-OSGi architecture used for the prototype implementation, for example, does not provide any message interception mechanisms and the approach of directly integrating PEPs into the individual services would work indeed, but of course require modifications of the services themselves. A non-invasive method to intercept method calls of remote services can be realised using dynamically generated proxy services as follows:

The idea is to replace each remotely published service by a proxy which implements the PEP, i.e., intercepts and forwards method calls to the PDP. For this purpose, we use *service hooks* to catch internal events of the OSGi framework which are sent when a service is either published or searched by another bundle. For each service that is registered in the framework, a proxy object implementing the services interface and containing the PEP code is created and registered so that for each remote service one original non-proxied and one PEP proxy exists. By hooking into the `find` service event, it is possible to hide the original services so that only the proxied versions are available via the OSGi service registry. So, whenever a client discovers and invokes an R-OSGi service in the prototype, it will work with the proxied version containing the PEP code. This way, it is possible to “policy-enable” a R-OSGi based middleware in a non-intrusive way, i.e. without any modifications to the existing services.

7.1.4. User interfaces

A framework alone is not usable without any user interface by which users can configure the framework’s settings and modify policies. Although user interface design is not subject of this thesis, on the one hand we needed tools to control the prototype for our experiments, and on the other hand we intended to practically evaluate whether the framework’s API is



ModularPolicy [OSGi Framework] /usr/lib/jvm/java-6-openjdk/bin/java (25.01.2012 19:58:13)

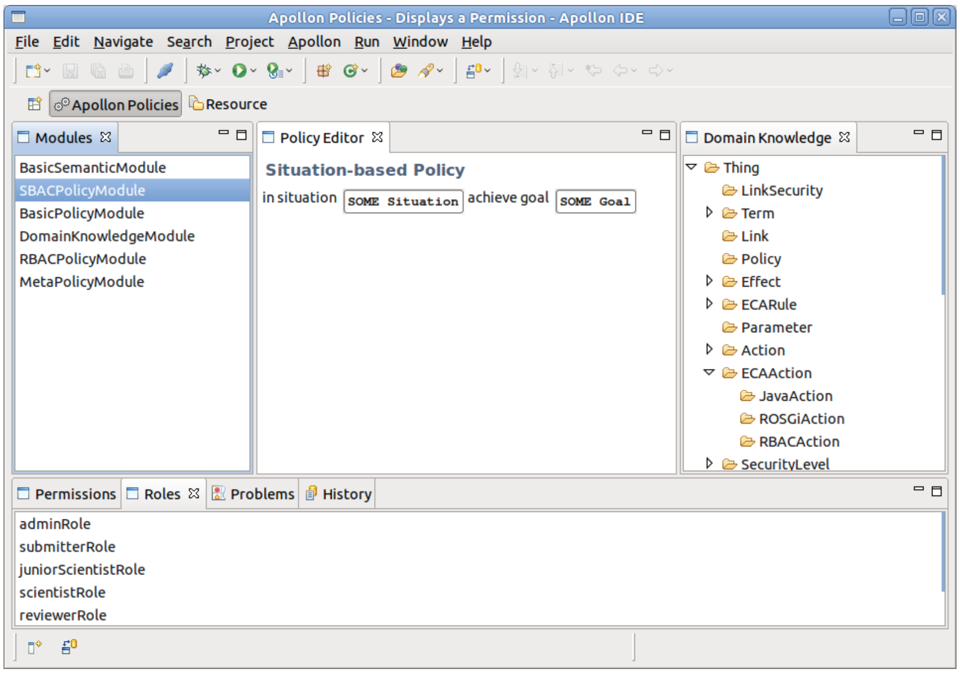
```
---Policy Tool---
        publish [topic] -       publishes an event
        configure [-l | <key<=<value>+] -       sets a configuration
        load [file]     -       loads a policy from a file
        showModules     -       shows all policy modules
        wire [producer.pid] [consumer.pid]      -       Wires a producer to a consumer service
        addMetaRule     -       Adds a meta rule
        delMetaRule     -       Removes a meta rule
        lsClass -       Shows the currently loaded policy structure
        lsInd   -       Shows all loaded individuals
        reasonerInfo    -       Shows some information about the reasoning engine
        ontInfo -       Shows some information about the ontology
        addAuthRule     -       Adds an authorisation rule
        delAuthRule     -       Removes an authorisation rule
---Controlling the Console---
        more - More prompt for console output
        disconnect - Disconnects from telnet session
        help <commmand> - Display help for the specified command

osgi> addMetaRule
Enter rule name: testRule
Enter subject: User AND NOT AdminUser
Enter resource: Resource THAT ownedBy SOME InternalUser
Enter action: read
Enter strength <weak|strict>: weak
Add compulsory obligation (ENTER to continue): LogRequest
Add compulsory obligation (ENTER to continue):
Add forbidden obligation (ENTER to continue):
Give obligation to which you want to provide alternatives (ENTER to continue): Logrequest
  Add an alternative (ENTER to continue):
SnmpNotify
  Add an alternative (ENTER to continue):

Give obligation to which you want to provide alternatives (ENTER to continue):
```

Figure 7.4.: Screenshot of the Command Shell Prototype

in fact usable and provides the necessary functionality. Therefore, two user interfaces have been implemented for the prototype: a local console application and a remote rich client.

The console interface was used to control the prototype during experiments with the use case evaluations which are described in chapter 7. It has been implemented as an extension to the OSGi console admin service and is locally installed in the PDP. Figure 7.4 shows a screenshot of the console shell. It allows to change settings, as well as to load additional policy modules from a repository at run time. Users can either load policies in the form of OWL documents, or create rules individually. Apart from the commands which are provided by the framework core, each policy module can extend the shell by providing its own command set.

The remote rich client was implemented as a proof-of-concept in order to evaluate how the framework could be used from a "Policy IDE", i.e., an integrated development tool for authoring policies, loading them into different PDPs, and managing the settings of each PDP. In contrast to the console interface, the rich client thus does not necessarily run in the same OSGi platform as the PDP but rather connects to the PDP Admin service via the R-OSGi protocol. So, because the rich client is not directly integrated into the PDP, the advantage is that it can be used to connect to and manage various PDPs in a pervasive system. Apart from changing configuration settings and loading policies into the framework, the rich client further makes use of the templates which can be provided by policy modules. Figure 7.5 shows a screenshot of the rich client with an editor for the Situation-Goal module (c.f. section 7.3) which has been automatically created from the module's policy template.


Apollon Policies - Displays a Permission - Apollon IDE
File Edit Navigate Search Project Apollon Run Window Help
Apollon Policies
Resource
Modules
Policy Editor
Domain Knowledge
Permissions
Roles
Problems
History

BasicSemanticModule
SBACPolicyModule
BasicPolicyModule
DomainKnowledgeModule
RBACPolicyModule
MetaPolicyModule

Situation-based Policy

in situation SOME Situation achieve goal SOME Goal

Thing
LinkSecurity
Term
Link
Policy
Effect
ECARule
Parameter
Action
ECAAction
JavaAction
ROSGiAction
RBACAction
SecurityLevel

adminRole
submitterRole
juniorScientistRole
scientistRole
reviewerRole

Figure 7.5.: Screenshot of the Policy Administration GUI Prototype

7.2. Dynamic role-based access control

Role-based access control (RBAC) was invented when it became apparent that directly assigning users to permissions lead to complex and hard-to-maintain policies. By introducing roles as an abstraction between users and permissions, access control policies could be formulated which were closer to the security models of most companies, where access rights are usually not bound to persons but rather to positions. Since then, RBAC has been further developed, various extensions have been proposed and a lot of research on *role mining*, i.e. the derivation of appropriate roles from a high-level security model has been undertaken. In the context of this thesis, such in-depth considerations of different RBAC models shall not be of interest. Rather, the purpose of this section is to show that the proposed framework is well suited to implemented one variant of this popular access control model whose application makes especially sense in pervasive and context-sensitive systems: Hierarchical and Dynamic RBAC. Compared to the core RBAC model where user-role-assignments are static, dynamic RBAC allows users to activate a subset of their assigned roles at any time. This takes into account the conditions of systems where users do constantly act under a certain role, but rather may switch between the role of a paper *reviewer* and *author*, or a project *proponent* and *grantor*, for example. While in static RBAC, *Separation of Duty* (SoD) is an often-used constraint which shall guarantee that a user is not assigned to two incompatible roles at the same time, in a DRBAC model, this condition can be classified into *Static SoD* (SSoD) and *Dynamic SoD* (DSoD). The former prevents a set of roles from being assigned to the same user, no matter whether the roles are activated or not, while the latter does only prevent incompatible roles to be activated at the same time, but allows to assign them to the same user in general. In addition, role hierarchies will be considered, i.e. roles can be ordered in a tree-like structure where a more specific sub role will usually be assigned to more permissions than its more generic super role.

In this section, it will be shown how hierarchical DRBAC can be realised using the proposed framework by proposing a hierarchical DRBAC model in DL, explaining how SSoD and DSoD can be tested against that model and how role activations would be realised.

7.2.1. Modelling RBAC in OWL

A number of attempts on formalising RBAC in Description Logic / OWL have been made before. The authors of [59] discuss two approaches whereas the first one ("roles as classes") models users as individuals and roles as classes. By assigning the individual of a user to the classes of its role, role membership of the user can then be expressed. The second approach ("roles as values") models roles as individuals and assigns them to users using a `hasRole` property. Both approaches have however drawbacks: as stated by the authors of [57], the first approach does not support the specification of SoD over two hierarchical roles. As SoD is modelled as class disjunction, i.e. $A \sqcap B = \emptyset$, a SoD constraint over hierarchical roles $A \sqsubseteq B$ would result in an inconsistent model, due to $B \sqcap B = \emptyset$. According to the authors of [59], the second approach has the drawbacks that rules, e.g. in SWRL, need to be applied in order to model a sub role's inheritance of permissions assigned to its super roles. Therefore, the authors favour the first (flawed) approach which does not require rules. In [57], the authors adopt the second approach in order to extend XACML by a DL-based RBAC model. They remove the need for rules by modelling SoD by disjunct concepts as follows: $(hasRole.A) \sqcap (hasRole.B) \equiv \emptyset$. However in their model, permissions are directly assigned to users instead to roles, which does not comply with the traditional core RBAC model [55] that is considered in this section and works around the question on how to inherit permissions from super roles without applying rules.

The DRBAC model proposed in this thesis is based on the approach from [57], but has been adapted to match the RBAC specification without requiring any rules. At first, the generic class subject from the basic authorisation model in section 5.3 is extended by a subclass User, in order to still allow subjects which are not part of the RBAC model, being an instance of $\neg User \sqcap Subject$. Users are assigned to roles using a $hasRole$ property and permissions are assigned to roles using a $hasPermission$ property. To distinguish between activated and deactivated role memberships, a further sub-property $hasActiveRole \sqsubseteq hasRole$ is introduced. Both, roles and permissions are modelled as instances of the $Role$ and $Permission$ class, respectively, and by adding a transitive $subRoleOf$ relation it becomes possible to express role hierarchies. To ensure that a user is automatically assigned to all super roles of her roles, so-called *property chains* are used – a feature which is supported in OWL 2 and replaces the need for DL-safe rules at this point. Property chains allow to build classes from the conjunction of properties, i.e. the case that a user is assigned to all super roles of her roles can be written by $hasRole \leftarrow hasRole \circ subRoleOf$. Likewise, property chains for $hasActiveRole$ and for applying the permissions of super roles to a sub role are added to the model. Altogether, the DRBAC model can be written as follows:

$$\begin{aligned}
\mathit{User} &\sqsubseteq \mathit{Subject} \\
\mathit{User} &\equiv \forall \mathit{hasRole}.\mathit{Role} \sqcap \\
\mathit{Role} &\equiv \forall \mathit{hasPermission}.\mathit{Permission} \sqcap \\
& \quad \forall \mathit{subRoleOf}.\mathit{Role} \\
& \quad \forall \mathit{hasActiveRole}.\mathit{ActiveRole} \\
\mathit{Permission} &\equiv \forall \mathit{hasResource}.\mathit{Resource} \sqcap \\
& \quad \forall \mathit{hasAction}.\mathit{Action} \\
\mathit{ActiveRole} &\sqsubseteq \mathit{Role} \\
\mathit{superRoleOf} &= (\mathit{subRoleOf})^{-} \\
\mathit{hasRole} &\leftarrow \mathit{hasRole} \circ \mathit{subRoleOf} \\
\mathit{hasActiveRole} &\leftarrow \mathit{hasActiveRole} \circ \mathit{subRoleOf} \\
\mathit{subRoleOf} &\sqsupseteq \mathit{subRoleOf} \circ \mathit{subRoleOf} \\
\mathit{hasPermission} &\leftarrow \mathit{subRoleOf} \circ \mathit{hasPermissions}
\end{aligned}$$

A developer who wants to specify a set of DRBAC rules can do so now by creating individuals for users, resources, actions and roles at first. Then, users will be assigned to roles using the *hasRole* property and roles will be assigned to permissions using the *hasPermission* property.

Evaluating an access request against this DRBAC model can be done in pure DL, without requiring any proprietary decision mechanism. Whenever an access request arrives, containing subject, resource and action, the DRBAC Decision plugin queries the knowledge base to find the set of users for which an active role exists that is assigned to a matching permission. This query can be written in Manchester DL syntax as in Listing 7.1, and returns a non-empty set of users if the access request shall be granted or an empty set if no such permissions exist and access shall be refused.

Listing 7.1: Query for evaluating access requests in the DRBAC model

```
User AND <user> AND hasActiveRole
  SOME (Role THAT hasPermission
    SOME (Permission THAT hasAction VALUE <action>
                AND hasResource VALUE <resource>))
```

7.2.2. Modelling Separation of Duty

A common constraint in role-based access control is separation of duty, as explained above. The DRBAC model proposed herein supports formulating both static and dynamic separation of duty constraints, relying only on Description Logics. That is, just as for the actual policy decision, also testing SoD constraints can be done using a standard semantic web reasoner.

7.2.2.1. Static Separation of Duty

Static Separation of Duty (SSoD) shall prevent a user from participating in incompatible roles at all times, i.e. no user must be member of incompatible roles, no matter whether they are activated or not. This constraint can be modelled by creating a set of disjoint classes, each representing members of one of the incompatible roles. One example would be to express that no member of a research funding agency (funder) must apply for funding proposals (*applicant*), two classes $Ssod_1$ and $Ssod_2$ can be specified and marked as disjoint, as shown in the formula below. Whenever there is a user that has both roles, applicant and funder, the knowledge base will contain an instance of the unsatisfiable class $Ssod_1 \sqcap Ssod_2$ and will thus become inconsistent. Therefore, a consistent knowledge base inherently guarantees all SSoD constraints to be fulfilled.

$$\begin{aligned} Ssod_1 &\equiv hasRole.\{\texttt{applicant}\} \\ Ssod_2 &\equiv hasRole.\{\texttt{funder}\} \\ \emptyset &\equiv SSod_1 \sqcap Ssod_2 \end{aligned}$$

7.2.2.2. Dynamic Separation of Duty

Corresponding with the modelling of SSoD, also Dynamic Separation of Duty (DSoD) can be expressed by class disjointness axioms. Consider the example where a scientist may act as a reviewer and as an author, but not both at the same time. In that case, two classes $Dsod_1$ and $DSod_2$ would be created, corresponding to *active* membership is each role and declared as disjoint.

$$\begin{aligned} Dsod_1 &\equiv hasActiveRole.\{\texttt{applicant}\} \\ Dsod_2 &\equiv hasActiveRole.\{\texttt{funder}\} \\ \emptyset &\equiv DSod_1 \sqcap Dsod_2 \end{aligned}$$

Just as for SSoD, the knowledge base would become inconsistent if multiple incompatible roles would be activated. Thus, the role activation plugin described in the following section will only have to check consistency of the knowledge base and refuse to activate a role if it would render the knowledge base inconsistent.

7.2.3. Dynamic Role Activation using ECA Policies

So far, the DRBAC model has been statically defined in Description Logic. This changes now as roles need to be activated and deactivated at run time. Role activations are triggered by external events and result in a modification of the knowledge base, concerning the *hasActiveRole* property. In order to trigger role activation upon external events, the DRBAC module extends the core ECA policy model, defined in section 5.4. It defines the following two subclasses of *ECAAction*:

$$\begin{aligned} ActivateRoleAction &\sqsubseteq ECAAction \sqcap hasRole.Role \sqcap \\ &\quad hasSubject.\texttt{string} \sqcap \\ &\quad hasExecutable.\{org.example.rbac.ActivateRole\} \\ DeactivateRoleAction &\sqsubseteq ECAAction \sqcap hasRole.Role \sqcap \\ &\quad hasSubject.\texttt{string} \sqcap \\ &\quad hasExecutable.\{org.example.rbac.DeactivateRole\} \end{aligned}$$

The *hasRole* property declares the role to activate (deactivate) and the *hasSubject* data property takes a variable which is to be filled with the name of the subject whose role membership is to be activated (deactivated). The actions for activating (deactivating) the role are identified by the class name of the respective ActionPlugins, here for the sake of illustration named `org.example.rbac.(De-)ActivateRole`. Using these actions, an ECA policy for activating the role *scientistRole* of the subject identified by the event's *subject* attribute could be written as follows:

$$\begin{aligned} \mathit{ExampleRule} \equiv\ & \mathit{hasAction}.(\mathit{ActivateRoleAction} \sqcap \\ & \quad \mathit{hasSubject}.\{"?\mathit{subject}"\} \sqcap \\ & \quad \mathit{hasRole}\{\mathit{scientistRole}\}) \sqcap \\ & \mathit{hasEvent}.(\mathit{ExampleEvent} \sqcap \mathit{hasVariable}.\{"?\mathit{subject}"\}) \end{aligned}$$

Upon receipt of the event, the PDP fills the ?*subject* variable with the value of the *subject* attribute of the event and hands it as a parameter to the *ActivateRoleAction*. The ActivateRoleAction then checks at first if the subject is a member of the stated role and if is not already activated. Then, a *hasActiveRole* property is added to the subject's instance and the knowledge base is re-classified. Only if the knowledge base is still consistent after re-classification, all SSoD constraints are fulfilled and the role may actually be activated (deactivated). Otherwise, the change of the knowledge base is reverted and hence role activation (deactivation) is refused. The ActivateRoleAction is thought to be a generic component which can be integrated into any specific application and therefore does not make any assumptions about where the events come from or who created them. It is thus up to the application developer to write ECA policies triggering the ActivateRoleAction or create a user interface for manually triggering role changes, for example.

7.2.4. Analysing Properties of the DRBAC Model

Now that it has been shown that traditional access control models like DRBAC can easily be realised using the proposed framework and that Description Logic is expressive enough to model most aspects of such a policy, the benefits of a DL-based model shall be pointed out by giving some examples on how a DL-based policy can be analysed with respect to questions frequently occurring during the policy specification process. Consider the following exemplary questions:

Will user *U* ever get access to resource *R*? This will probably be the most interesting question, as it reveals both if a user will have access to all required resources and if a resource is not accessible for illegitimate users. It can be tested using the following query.

```
User AND {U} AND hasRole SOME (Role THAT hasPermission SOME (Permission THAT
  hasResource R))
```

Will resource *R* be accessible from role *O*? The same question as above, only with respect to roles instead of users can be answered by the even shorter query

```
Role AND {O} THAT hasPermission SOME (Permission THAT hasResource R)
```

Which operations are in general allowed on resource *R* and who can carry them out? To assess the exposure of an individual resource, it might be of interest to find out which operations users are carry out on a resource. This can be found out using by first querying for the allowed actions:

```
Action isActionOf SOME (Permission THAT hasResource R)
```

Then, for each result *A* of the previous query, find out the set of users which are assigned to a respective permission:

```
User AND hasRole SOME (Role THAT hasPermission SOME (Permission THAT hasResource
  R AND hasAction A))
```

So, due to the policy model being in Description Logics, it is possible to reason over policies and to realise numerous analysis tasks which will help users to understand the effects of a policy and reveal possible flaws. In contrast to languages like XACML where such analysis is either not possible at all or requires an error-prone and tedious translation into a specific formalism, the DL-based policy is inherently formalised and can be analysed using a standard reasoning engine. What cannot be analysed however, are occurrence and possible chronology or role activations, as this information is not available in the knowledge base. In the current model, any role can be activated at any time and there is no semantics of role activation events. As this information is application-specific, it has not been integrated into the generic DRBAC model on purpose. Nevertheless, the modular design of the proposed policy framework would allow to add a further, high-level policy model on top of DRBAC, which would describe events leading to role activations, as well as their interdependency. For example, if roles would be activated according to the location of a user, a respective high-level model would allow to analyse whether a single user could be synchronously active in two roles assigned to different locations.

7.2.5. Prototype Evaluation

As reasoning over description logic is up to NexpTime [75], it is interesting to evaluate whether a DL-based policy decision process shows acceptable performance to be used in practical applications. In order to test the decision speed of the DRBAC model, a prototype has been implemented and two populations of the DRBAC model have been tested for performance: one gracious population size which relates to a typical small-scale system with only few users and roles (*Small Ontology*) and one more demanding population size which relates to a large-scale pervasive system (*Large Ontology*).

The prototype has been implemented as a module for the basic policy framework, introduced in section 7.1, which provides the DRBAC model, the decision process and the role activation / deactivation functionality. The module is realised as an OSGi bundle providing the `PolicyModule` service which can be discovered and registered by the framework at runtime. It depends on a `BasicSemanticModule` which bears all semantic-related functionality, making use of the Pellet v2.2.2 reasoner[10] and the OWL-API v2[11] for managing access to ontologies. We tested the time for deciding an access request in the RBAC model, i.e. the runtime of the `RBACDecisionPlugin.decideAccessRequest` method for both the *Small* and *Large* test set on an Intel Core 2 Duo 2GHz, Ubuntu 10.04, Sun Java 1.6.0.22 platform. Pellet's optimisation settings have been left untouched.

Both test sets populate the RBAC model, which is of $\mathcal{ALCOI}(\mathrm{D})$ expressivity. The *Small* test set consists of four users, six roles and three permissions assigned to them and results in an ontology whose metrics are shown in Table 7.1. In contrast to that, the *Large* test set features a significant greater number of individuals: here, 1003 user, 205 roles and 200 permissions have been specified. Again, the detailed metrics are shown in Table 7.1.

In Table 7.2, the runtime results for both test sets are shown. As expected, the *Small* test set performs significantly better than the *Large* set, requiring about 5 ms per policy decision. Yet, the *Large* test set still delivers a decision in about 20 ms by average. While these results

[10]http://clarkparsia.com/pellet/
[11]http://owlapi.sourceforge.net/

Axioms	Small	Large
Individual axioms	88	4279
Class axioms	13	19
Object properties	28	46
Data properties	7	7
RBAC Entities		
Users	4	1003
Roles	6	205
Permissions	3	200

Table 7.1.: Metrics of the *Small* and *Large* test sets: number of axioms and RBAC entities

Test set / Runtime	Min	Max	Avg
Small	1.02	6.96	5.04
Large	10.87	25.02	20.47

Table 7.2.: Time for deciding access requests for the *Small* and *Large* test sets of the DRBAC model (in ms)

cannot be regarded as universally applicable, they still show that, considering that only few systems will feature a greater user- and role base than modelled in the *Large* test set, the pure DL-based policy decision process of the DRBAC model would be usable in real-world applications for input sets of reasonable size.

7.3. Situation-based security

The foundations of the extensible policy model have been described in chapter 5 and in the previous section, it was shown how a traditional DRBAC access control model can be realised by extending this model. In this section, a policy model on top of the basic ECA model for reactive policies is introduced, which is more tailored to the requirements of pervasive systems. Parts of the results presented in this section have been published in [154] by the author of this thesis.

Pervasive systems are characterised by context-awareness, i.e. responsiveness to changes in the environment, as well as a heterogeneous and ad hoc connected infrastructure. As this design might require a constant re-configuration of the system in order to keep up with the security requirements, it is obvious that a manual approach is not feasible. Rather, policy authors need to define goals which the system will automatically attempt to achieve, depending on the current situation. Simple Event-Condition-Action (ECA) policies, as introduced in section 5.4, have been in use for years in order to specify actions with which a system should react to certain events. However, while ECA policies might be useful in many cases, they lack the additional semantics that would be required in order to make a pervasive system "self-adaptive", i.e. adapting its configuration in such a way that it matches some predefined criteria, even under changing conditions. On the one hand, reacting to a single event is too short-sighted, because a reaction might need to be triggered by a complex constellation of different events within a certain time span, a so-called *complex event*. On the other hand, declaring a dedicated "action" does not necessarily result in the intended effect: it remains unclear for the user whether the action is supported at all, if it is properly executed, and if it actually leads to the desired result or if it would be counteracted by any other action.

In this section we therefore extend the classic ECA model by the notion of *situations* and *goals* which allows users to formulate high-level policies of the form "`in situation X, achieve goal Y`", rather than the traditional "`on event if condition do action`". Given an appropriate model of the system, including its supported actions, this type of policy will allow to realise a self-adaptive system, which automatically reconfigures itself depending on the current situation and thus goes beyond the semantics of traditional ECA rules.
Section 7.3.1 starts by giving an overview of the involved components and their integration into the overall policy framework. Then, section 7.3.2 described an extension of the ECA model from section 5.4. Finally, section 7.3.3 gives an overview of the implementation of a corresponding policy module and a discussion of its practicability.

7.3.1. Overview

The developed policy module adds an abstraction layer on top of ECA policies, so that users do not have to think in terms of specific events and actions, but rather in terms of goals which should be achieved in a certain situation. As an example, a policy could express that when working in an "insecure environment" (situation), all communication channels should apply at least "some encryption technique" (goal). This cannot be easily expressed in ECA rules, as it would require to define actions for each constellation of supported encryption protocols on the one hand, and to define individual rules for any event that could lead to the situation of an "insecure environment". The abstraction layer must therefore provide two aspects:

Firstly, situations must be modelled on top of events. In contrast to an event, a situation is state-based and therefore temporarily constrained. It is triggered by some initialising event and ended by another event or after a certain time span, whereas an event does not necessarily have to refer to one single observable incident, but can rather be a pattern of events, i.e. a complex event. Detecting such patterns in a stream of events is the task of Complex Event Processing (CEP), a technique which has originally been used in applications like algorithm-based stock trading, and credit card fraud detection, but has soon spread to other application areas like security information and event management (SIEM), sensor networks [48, 166], or spam detection [201].

Secondly, goals must be modelled on top of actions. While actions simply refer to a piece of code that is to be executed by the enforcement mechanism, the assumptions and consequences of an action remain unclear, so that it is not possible to make statements about their expected effect or interdependencies. In contrast to that, goals refer to the system state which should be achieved and do not pre-determine the actions which have to be taken in order to reach that state. The task of refining the goal definition to a suitable set of actions is called *Planning* and is an area in the field of Artificial Intelligence.

Both technologies, CEP and Planning will be integrated into the decision process of the situation-based policy module and the ECA policy model will be extended so as to express situations and goals, instead of events and actions. In addition, both CEP and Planning require specific information, which has to be provided by the model as well:

The definition of complex events requires CEP queries, defining operations on so-called event streams. An event stream comprises all events of a specific type (e.g., an SNMP event) and the query defines how events from different streams should be correlated with each other in order to match a predefined pattern. So, the event definition will either have

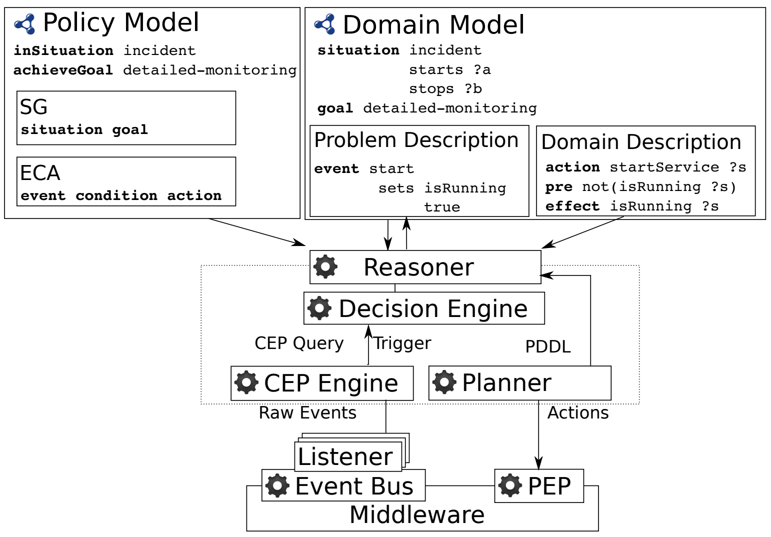


Figure 7.6.: Interplay of components in the situation-based security module.

to directly provide a query that can be interpreted by the CEP engine or model an event algebra, from which such a query can be constructed.
A planning engine requires input of two different types: a problem description and a domain description. The problem description contains a model of the current state of the system and the goal, while the domain description models all possible actions with their pre- and postconditions. Based on problem- and domain description, a planning engine is able to either compute possible *plans*, i.e. sequences of actions which lead from the current state of the system to the desired goal or to tell that no such plan exists.

Figure 7.6 illustrates how CEP engine, Planning engine and the required models are composed in the situation-based security module: complex event queries are constructed from individuals in the situation model and registered at the CEP engine. The system emits events, such as sensor data, SNMP status messages or OSGi-internal events, and the EventAdapter of the Situation Based Policy Module receives them. The CEP engine then continuously tests the streams of incoming events against the predefined queries and notifies the Decision plugin, if a matching pattern has been found. Depending on the situation model, the detected event pattern will either start or stop different situations. To each situation a number of goals is attached and it is the Planning engine's responsibility to identify how a goal can be achieved. Therefore, the Planning engine is triggered whenever a situation has been started or stopped, receives the goals attached to the currently active situations and computes a plan, i.e. a sequence of actions, which leads from the current state of the system to a state in which all required goals are achieved. For this task, it requires input in form of all actions supported by the system, including their pre- and postconditions (*domain description*), the current system state and the goals which must be achieved (*problem description*).
The goal description is part of the policy, so there is no need to extend any of the existing concepts for that. The set of supported actions and their pre- and postconditions needs to be provided as an extension of the basic *Action* concept in order to read it into the planning engine, and the current system state needs to be measured by the policy module and provided to the Planning engine. As it changes continuously, it can apparently not be predefined in a model, but has either to be constantly measured by the Situation Based

Policy Module or needs to be assessed whenever a re-planning becomes necessary. The "system state" is determined by the predicates of the planning domains, i.e. the attributes to measure can be read from the planner's domain description. In section 7.3.3 below, we will illustrate how these attributes can be measured using the ECA model.

7.3.2. Model extensions

As shown in Figure 7.6, the Situation Based Policy Module extends the basic ECA policy model by concepts for describing situations, goals, and actions.

Although there is no common and clear-cut definition of *situations* in the area of pervasive systems, significant work in the field of Situation-Awareness has been done, where the term "situation" mostly refers to knowledge about a system which is valid within temporary limits. A very early and one of the most-cited definitions of Situation-Awareness has been given by Endsley and suggests that situations are constrained in "time and space" and build the basis for decision-making [51]:

> *Situation-Awareness is the perception of elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future.*

In the field of ubiquitous computing, Moschgath attempts a differentiation between the terms "context", "situation" and "environments" [111] and concludes that they mainly differ in the perspective taken when considering a system: while "context" is subject-specific, "situation" takes a global perspective and "environment" explicitly excludes the subject from the consideration. As in the area of ubiquitous computing, these different perspectives would hardly make any difference, Moschgath further suggests that "situation", "context" and "environment" could be used interchangeably.

For the Situation Based Policy Module, we will use the term "situation" to stress statefulness and temporary limitation. A situation defines a state which the system enters upon occurrence of a certain event and leaves it again upon another event or after a certain timespan. "Context", in contrast to that, is mainly referred to as "any information that can be used to characterize the situation of an entity" [43] and does not express a temporary constrained state. So, all that is required to define a situation in the Situation Based Policy Module is the (complex) event that triggers the situation and the event that terminates it. A situation is therefore modelled as:

$\mathit{Situation} \equiv \forall \mathit{startedBy}.\mathit{Event} \sqcap \forall \mathit{stoppedBy}.\mathit{Event}$

A *Goal* describes a system state which is to be achieved in a certain situation. By which predicates the system state is defined depends on the specific system and its measurable parameters, so we define a concept *Goal* but do not put any further constraints on it. Rather, when applying the Situation Based Policy Module in a specific application, the *Goal* concept will need to be extended so as to match the requirements of the underlying planning engine and its supported goal definitions.

Further, the Situation Based Policy Module models a *PlannableAction* concept. Just as for the *Goal* concept, *PlannableAction* is a generic description of actions, along with their pre- and post-conditions and needs to be refined by sub-concepts and and instantiated by individuals in a practical application, as we will show in section 7.3.3 below.

$\mathit{PlannableAction} \equiv \forall \mathit{pre}.\mathit{String} \sqcap \forall \mathit{post}.\mathit{String}$

Now that *Situation* and *Goal* have been modelled, users can write policies to assign the goals to achieve in a situation:

$\mathit{SCPol} \equiv \forall \mathit{in}.\mathit{Situation} \sqcap \forall \mathit{achieve}.\mathit{Goal}$

Finally, further concepts are required so as to instruct the planning engine how to retrieve information about the current system state. A system state is described by a set of key-value-pairs, called *predicates*. By modelling a subconcept of *Action*, we can declare a *SetPredicateAction* which sets the value of a predicate, identified by the *key*. This action can be triggered by ECA rules as any other actions.

$SetPredicate \equiv\sqsubseteq Action \sqcap \forall setsPredicate.Predicate$

A predicate is then modelled as follows:

$Predicate \equiv \forall hasKey.String \sqcap \forall hasValue.String$

These predefined concepts are the common least denominator for formulating goal-based policies for self-adapting systems. In order to make use of them in a practical application, some concepts have to be refined in order to match the capabilities of the underlying middleware, event processing and planning engine.
In the following section, an instantiation of these concepts for a practical application of the Situation Based Policy Module in a pervasive system will be introduced. Concepts like *Goal* and *Action* will be refined and amended by a simple security model that allows users to formulate situation-specific protection goals, which the system autonomously tries to achieve.

7.3.3. Prototype

As a proof of concept, an implementation of the Situation Based Policy Module has been created and applied in a pervasive systems use case where policies control the security of an adaptive middleware in changing situations. Such self-adaptation is a common requirement in pervasive system, as it is necessary to achieve a consistent level of security while the infrastructure of the system is continuously modified – services appear and disappear, are bound to different consumers and roam across different platforms and connection types. It is obvious that a manual adaptation of security mechanisms is not feasible under these conditions but rather, the system must be able to configure itself autonomously.

We first give an overview of the assumed system architecture and point out that it is in accord with the design of typical pervasive system middlewares. Then, we describe how the above-described models are extended so as to allow users to formulate generic security goals and illustrate how high-level security policies can be formulated on top of the extended models using the Situation Based Policy Module. Finally, we discuss some aspects of the implementation, including the middleware layer, a CEP and planning engine.

Use case: a self-adaptive middleware architecture The concept of adaptivity is featured by most pervasive systems middlewares to a greater or lesser extend. For this use case, we consider a component-based and loosely coupled middleware layer that allows to load, unload discover and bind components at runtime. While this system architecture provides great flexibility and therefore has been adopted by many pervasive systems middleware, it comes at the cost of complexity and difficult maintainability. We will therefore apply the Situation Based Policy Module in order to allow users to assign abstract protection goals to situations and let the Situation Based Policy Module autonomously identify and apply appropriate security mechanisms.

Components in the middleware layer are classified into *consumers* (consuming and processing data) and *providers* (collecting and providing data). Both are made available over the network via a remote binding mechanism and can connected to each other by a *connector*. For each wiring between a consumer and a provider, a connector is created which retrieves data from the provider and forwards it to the consumer – either at the

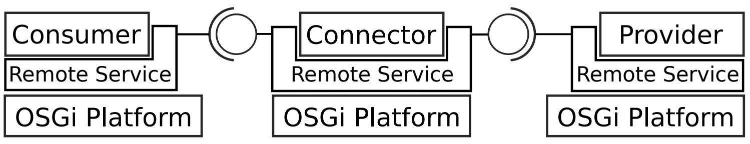


Figure 7.7.: Architecture of a component-based middleware

consumer's (pull) or at the provider's request (push). This way, both consumers and providers are anonymous to each so the wiring between them can be set up at runtime in an ad hoc fashion. Various practical implementations of this generic pattern (c.f. Figure 7.7) are available: with version 3 of the OSGi specification the Wire Admin service has been introduced and in [101], a proposal on using it for connecting remote services over the network has been made. By now, remote wiring according to the connector pattern is also used in various middlewares like RUNES [36], LooCI [79] or Hydra [197]. Another architectural pattern considered in this use case is the Invocation Interceptor pattern [186] which registers a chain of *guards* into the request- and and response process. For each request and response, all registered guards are invoked, process the message and either pass it on to the next guard or interrupt the processing chain by dropping the message. Guards are components as well and therefore can be loaded at runtime by the middleware and attached to consumers and providers. Again, this pattern is prevalent in pervasive systems: LooCI supports interception of event delivery [79], LinkSmart uses Axis handlers for message encryption and signing [72] and RUNES comes with a full-blown "Interception Meta Model"[12] [35].

Whichsoever implementation is used is not relevant for the model extensions introduced in the following, but rather only the concepts of the connector and interceptor pattern are important. The specifics of the proof of concept implementation will be discussed further below.

Domain and security model At first, the generic concepts of the Situation Based Policy Module need to be extended by application-specific information which describes the system architecture and its security implications. The component-based architecture introduced above consists of components, which are either *consumers*, *providers*, or *guards*. Consumers and providers can be connected to each other over by a *link* and guards can be assigned to both, consumers and providers. This results in the following model which needs to be populated when the Situation Based Policy Module is applied to a specific application – either manually by the user or by some automatic mechanism which investigates the system architecture and generates the appropriate individuals for the model.

$$
\begin{aligned}
&\mathit{Consumer} \sqsubseteq \mathit{Component} \sqcap \mathit{guarded}.\mathit{Guard}\\
&\mathit{Provider} \sqsubseteq \mathit{Component} \sqcap \mathit{guarded}.\mathit{Guard}\\
&\mathit{Guard} \sqsubseteq \mathit{Component}\\
&\mathit{Link} \equiv \mathit{from}.\mathit{Consumer} \sqcap \mathit{to}.\mathit{Provider}
\end{aligned}
$$

As users should be able to state protection goals in their policies, which then map to concrete system configurations, it is further necessary to describe the security properties of the components and their interconnections. Various security models have been proposed in the past, covering various perspectives such as threat- or countermeasure-centric views, compliance aspects and with focus on requirements analysis, implementation or test phase

[12] http://runesmw.sourceforge.net/intercept.html(August2011)

Security Level	Allowed actions	Predicate notation
high	$\emptyset$	`not(read),not(delete),not(insert)`
secrecy	$\{delete\}$	`not(read),delete,not(insert)`
integrity	$\{read, delete\}$	`read,delete,not(insert)`
none	$\{read, delete, insert\}$	`read,delete,insert`

Table 7.3.: Allowed actions per *Link* protection level

Guard	Allowed actions	Predicate notation
access control	$\{repudiate\}$	`not(call)`
logging	$\{call\}$	`not(repudiate)`
none	$\{repudiate, call\}$	`repudiate,call`

Table 7.4.: Supported guards and the implied predicates

in the lifecycle. As the aim of this section is mainly to point out how security models can easily be integrated into the policy definitions, we will not derive a new sophisticated security model but rather apply a rather simple one, that reflects security requirements and capabilities in a system architecture, similar to the UMLsec model [82], which has been introduced by Jan Jürjens. UMLsec is basically an UML profile for expressing security assumptions in UML diagrams and has been applied in a number of theses and research projects[13]. Yet, UMLsec is a very complex model with which cryptographic protocols, information flows and implementation details of components can be modelled. While we adopt some generic ideas of UMLsec, the security model used for this proof-of-concept implementation of the Situation Based Policy Module is therefore by far not as extensive and complex as UMLsec. Analogous to UMLsec, we describe the security properties of links and services by the actions an outsider (e.g. an attacker) is able to perform on them. Possible actions for a link are *reading* data, e.g. by eavesdropping communication on a network, *deleting* data by intercepting and dropping messages and *inserting* data by replaying or constructing messages. A service can be *called* or a call to it *repudiated* by the caller. From combinations of these actions, different protection levels can now be defined, as shown in Tables 7.3 and 7.4.

With the exemplary model of domain- and security information, users would now be able to describe components in the system and and assign security requirements in terms of protection levels to them. However, for the system to autonomously adapt itself so as to match these requirements, it has to be aware of its current configuration and possible actions it can take to reach the desired state. For monitoring the current system state, we write policies based on the ECA model and use a *PredicateSetter* action to remember the system state in form of a set of *predicates* and for describing the actions provided by the middleware layer, we define a *PlannableAction* concept which includes pre- and postconditions of the respective action in form of a boolean expression over the predicates, identified by a leading "?" (a usage example will be given below under *Implementation*).

$predicateSetter : (Action \sqcap \neg PlannableAction)$

$PlannableAction \sqsubseteq Action \sqcap pre.String \sqcap post.String$

With these application-specific extensions of the model, it is now possible to formulate situation-goal-policies which require certain protection levels in different situations. For

[13] A list of theses and applications related to UMLsec can be found at `http://inky.cs.tu-dortmund.de/main2/jj/umlsectool/applications.html` (last access August 2011)

Listing 7.2: Example of a goal-based policy

```
:in MyHomeAppRunning;
:achieve nonRepudiatableDoorAccess;
:achieve authenticSensorComm;
:achieve secretStorage.

:MyHomeAppRunning rdf:type :Situation;
                  :startedBy [a :OSGiServiceEvent;
                                :type "registered";
                                :service "myHomeApp".];
                  :stoppedBy [a :OSGiServiceEvent;
                                :type "unregistered";
                                :service "myHomeApp"].
```

example, a pervasive system middleware running the policy framework presented herein, might provide various services which can be composed by an application. The services themselves just provide generic data and controls, such as temperature and light measurements and controls for electric window blinds or door locks in a home automation setting. It is therefore not reasonable to bind security requirements directly to the services, but it should rather be the specific application running on top of the middleware layer that declares which protection goals have to be achieved by the services and their communication channels. An exemplary policy that requires all accesses to a door lock service to be logged for auditing purposes, all sensor measurements to be authentic and communication to a storage service to be secret could be modelled, as shown in Listing 7.2.

Used technologies In order to put the aforementioned policies into practice and to autonomously take actions in order to achieve the required protection goals, the Situation Based Policy Module needs on the one hand be aware of the current system configuration on the other needs to know which actions the underlying platform provides and what their effects would be. For the prototype implementation we use again the OSGi framework, as it has already been used to implement the core framework itself and supports bundle- and service management at runtime. However, other component layers are available, such as the Java Plugin Framework[14] (JPF), Zope[15] for Python, or Boost Extension for C++[16] just to name a few. To connect consumers and providers, OSGi provides the Wire Admin service, which is however only targeted at connections within the same platform. For the sake of this prototype, it has therefore been extended to a remote Wire Admin using the R-OSGi protocol for communication and service binding and mDNS for discovery of services and wires. Further, there is a control bundle providing commands for controlling the reconfiguration actions. These actions have been modelled as instances of the *PlannableAction* concept in order to describe their pre- and postconditions. Table 7.5 gives an overview of the actions and their representation in the model.

Besides the choice of the underlying component layer, another requirement for the Decision Plugin is the ability to receive events and process event queries over them. This requires an event bus mechanism, i.e. a broker component that accepts events and distributes them to the appropriate receivers. Here, various mechanisms are available,

[14] `http://jpf.sourceforge.net`
[15] `http://www.zope.org`
[16] `http://www.boost.org/`

Action name	Param.	Description
applyGuard	?comp ?guard	$\mathit{applyGuard} \sqsubseteq \mathit{PlannableAction}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{comp}\text{"}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{guard}\text{"}$ $\sqcap \mathit{pre}.\text{"}\mathit{not}(\mathit{guard}(?\mathit{comp}\ ?\mathit{guard}))\text{"}$ $\sqcap \mathit{post}.\text{"}\mathit{guard}(?\mathit{comp}\ ?\mathit{guard}),$ $\mathit{when}(=(?\mathit{comp}\ \mathit{log}))$ $\mathit{log}(?\mathit{comp})$ $\mathit{when}(=(?\mathit{comp}\ \mathit{ac}))$ $\mathit{ac}(?\mathit{comp})\text{"}$
wire	?link	$\mathit{wire} \sqsubseteq \mathit{PlannableAction}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{link}\text{"}$ $\sqcap \mathit{pre}.\text{"}\mathit{not}(\mathit{active}(?\mathit{link}))\text{"}$ $\sqcap \mathit{post}.\text{"}\mathit{active}(?\mathit{link})\text{"}$
unwire	?link	$\mathit{unwire} \sqsubseteq \mathit{PlannableAction}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{link}\text{"}$ $\sqcap \mathit{pre}.\text{"}\mathit{active}(?\mathit{link})\text{"}$ $\sqcap \mathit{post}.\text{"}\mathit{not}(\mathit{active}(?\mathit{link}))\text{"}$
applyEnc	?link	$\mathit{applyEnc} \sqsubseteq \mathit{PlannableAction}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{link}\text{"}$ $\sqcap \mathit{pre}.\text{"}\mathit{read}(?\mathit{link})\text{"}$ $\sqcap \mathit{post}.\text{"}\mathit{not}(\mathit{read}(?\mathit{link}))\text{"}$
applySig	?link	$\mathit{applySig} \sqsubseteq \mathit{PlannableAction}$ $\sqcap \mathit{hasParameter}.\text{"}?\mathit{link}\text{"}$ $\sqcap \mathit{pre}.\text{"}\mathit{write}(?\mathit{link})\text{"}$ $\sqcap \mathit{post}.\text{"}\mathit{not}(\mathit{write}(?\mathit{link}))\text{"}$

Table 7.5.: *PlannableActions* supported by the prototype

including messaging systems like JMX or full-fledged Enterprise Service Buses like Apache ServiceMix[17] or OpenESB[18]. As the main framework is already based on OSGi, which comes with a lightweight eventing mechanism that has also been used in large-scale scenarios with embedded devices, such as the simTD[19] field trial for Car-to-Car communication [162], the choice fell on the OSGi Event Admin bundle. The Event Admin is specified by the OSGi Alliance and is part of the OSGi compendium specification [128]. It provides functions for publishing and subscribing to events, whereas each event is identified by a topic and can contain attributes of any type (that is, any Java objects). The developed policy module therefore contains an EventAdapter which subscribes to all events published by the Event Admin and forwards them to the CEP engine for further processing.

Further, an appropriate CEP engine had to be chosen. With the advent of CEP as a business-relevant technology, several production-grade CEP solutions, as well as a number of research-driven implementations have become available. Examples are the Drools[20] rule engine, which has been extended by CEP features, ERMA[21], or Esper[22]. For the prototype implementation, the only criteria were a sufficiently high throughput of the engine that is comparable with state-of-the-art engines, as well as a reasonable size and an easy integration into the overall policy framework. The choice therefore fell on the Esper engine, a mature CEP engine which is used in several products and published under a dual-license model. Esper is more mature and feature-rich than ERMA and has a smaller footprint, as compared to Drools (~4 MB, vs. ~30 MB). It accepts event queries in EPL (Esper Processing Language), an SQL-like syntax for queries over event streams that is easily extensible by custom functions. As Esper is provided as Java library, converting it to an OSGi bundle which can be integrated into the main framework is almost straightforward.

There is a variety of planning engines to choose from. Planning is a mature research area and the main advances have been made in first years of the AI Planning Competition, which has started in 1998. During that series, a number of planning engines such as *IPP*, *PropPlan*, *FF* and *GraphPlan* have been developed. All of them interpret the PDDL language [62] for specification of the planning problem (i.e., the domain- and problem description). For the sake of easy integratability, the GraphPlan engine has been used, simply because it is written in plain Java and could therefore easily be converted to an OSGi bundle.

Implementation These technologies – event bus, CEP engine, and planner on top of a component-based platform – are now combined in the Situation Based Policy Module in order to realise two main functions: monitoring the current state of the system and reacting to detected situations by enforcing actions according to a plan which matches the goal of the policy.

Monitoring the system state requires to know which parameters should be observed and how they can be monitored. Using the extended ECA rules, activations of components, and wires can be observed and corresponding predicates be tracked by the Decision Plugin of the Situation Based Policy Module. For example, the following ECA rule sets the appropriate predicate when a component in form of an OSGi service is activated (Listing 7.3).

[17]`http://servicemix.apache.org/`
[18]`http://wiki.open-esb.java.net/`
[19]`http://www.simtd.org`
[20]`http://www.jboss.org/drools`
[21]`http://erma.wikidot.com/`
[22]`http://esper.codehaus.org/`

Listing 7.3: Example of an ECA policy to monitor system parameters

```
:on [ :a :OSGiServiceEvent;
      :type "registered";
      :providesParameter "?service".];
:do :setPredicate :hasParameter [
                       :key "service";
                       :value "?service".];
                    :predicate "isRunning ?service".].
```

In combination with the static facts from the aforementioned domain and security model, the Situation Based Policy Module can now generate input for the planning engine in form of a PDDL *domain-* and *problem description*.

First, type definitions are generated from all classes of the domain model, as in Listing 7.4.

Listing 7.4: Type definitions in PDDL

```
(:types component service link guard - object)
```

Then, the `predicates` block is composed of all properties from the domain model, together with the predicates monitored by ECA rules (Listing 7.5).

Listing 7.5: Predicate definitions in PDDL

```
(:predicates
 (providesService ?Component - component ?Service - service)
  (consumesService ?Component - component ?Service - service)
  (from ?Link - link ?Component - component)
  (to ?Link - link ?Component - component)
  (isrunning ?Component - component)
)
```

Finally, all instances of the *PlannableAction* concept (c.f. Table 7.5 above) are translated into a set of `action` definitions (Listing 7.6).

Listing 7.6: Action definitions in PDDL

```
(:action applyGuard :parameters (?C - component ?G - guard)
  :precondition "(not(guard(?C ?G))"
  :effect "guard(?C ?G)
           when (=(?C log)) log(?C)
           when (=(?C ac)) ac(?C)"

(:action applySigModule :parameters (?Link - link)
  :precondition (write ?Link)
  :effect (not(write ?Link))
)

...
```

Now that available actions have been modelled and the current system state can be assessed, high-level policies over situations and protection goals can be formulated. The

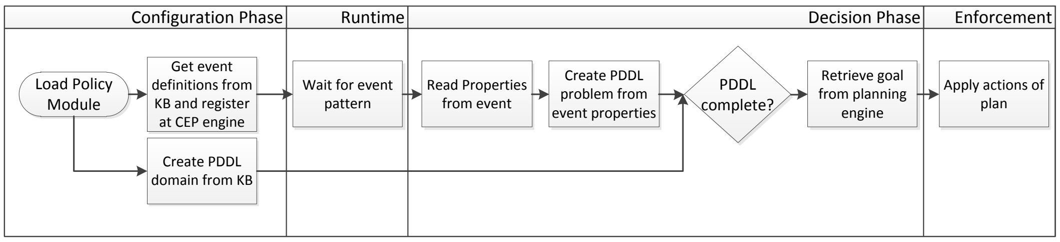

Configuration Phase
Runtime
Decision Phase
Enforcement
Load Policy Module
Get event definitions from KB and register at CEP engine
Create PDDL domain from KB
Wait for event pattern
Read Properties from event
Create PDDL problem from event properties
PDDL complete?
Retrieve goal from planning engine
Apply actions of plan

Figure 7.8.: Operations of the Situation Based Policy Module in the different runtime phases.

following Listing adds the definition of the goals from the policy in Listing 7.2.

Listing 7.7: Goal definitions of a goal-driven policy

```
# Situation—Goal Rule
:exampleRule a SGRule;
             :in       :WorkingEnvironment ;
             :achieve  :SecretCommunication ].

# Goal definitions
:nonRepudiatableDoorAccess a :SecurityGoal;
         :hasComponentSecurity [
                    :component  :DoorLockServices; //All door locks
                    :level      "auditableAccess"].

:authenticSensorComm a :SecurityGoal;
          :hasLinkSecurity [
                            :link  [a :Link;
                            :to :Sensor] //All links to sensors
                    :level "integrity")].

:secretStorage a :SecurityGoal;
          :hasLinkSecurity [
                    :link  [a :Link;
                          :to :StorageProvider] //All links to storages
                    :level "secrecy")].
```

With the goal description, all information required by CEP and planning engine is now available. Once the Situation Based Policy Module is loaded and its configuration phase is executed, complex event queries are read from the event model and registered in the CEP engine. Once one of the event patterns which start a situation according to the Situation-Goal policy is detected by the CEP engine, the goals assigned to the activated situation are read from the ontology, along with the description of actions, and predicates. The Decision plugin writes this information into a PDDL file and hands it over to the planning engine. The planner then generates the plan, i.e. a sequence of actions, and returns it to the Decision plugin, which sends each of these actions to the PEP for execution. Figure 7.8 summarises this process.

7.4. Handling cross-domain policy conflicts

While many traditional policy frameworks either allow for hierarchical structuring of administrative policy domains or simply assume that all resources are under control of a single domain, this does not fit the requirements of pervasive systems where policy domains can be combined in an ad hoc fashion and their decisions need to be aligned. The policy framework developed in this thesis supports metapolicies to allow users to express how decisions from their own domain shall be combined with those of another one.

In this section, the results of a prototype realisation of the metapolicy module will be discussed by means of an illustrative example. First, it will be shown how users can apply metapolicies in order to create "mergeable" policy decisions, while allowing each administrative domain to run their own policy model. Second, we will discuss the benefits of reasoning over the policy model and the characteristics of the prototype.

7.4.1. Use Case

To illustrate the need for metapolicies, let us consider the following fictive use case:

We consider an aggregation service which collects medical data from various sources, processes it and provides it to the user. Such a service could for example combine patient records from a hospital with live data from a body area network (BAN) of the patient and display it to a general practitioner (GP). Using this aggregation service, the GP will be able to remotely observe the health status of a patient, so the patient can stay at home and still be sure that critical conditions will promptly be discovered. The patient further has a base station to connect to his BAN and display its measurements – it might even be possible to connect to the BAN base station via Bluetooth so the user can manage the BAN with a smartphone app.

Scenarios like this are already partly reality and are appealing for users and service providers likewise. However, especially health data is critical and besides legal regulations, privacy issues play an important role and must be taken seriously in order to ensure user acceptance. Looking at the conceivable access rights and usage requirement in this example, it becomes clear that it can be tricky to policy-enable multi-domain systems.

The clinic runs a comprehensive access control model in which users have roles depending on their function and department, and medical staff can further be assigned to cases which gives them the right to inspect the medical history of the patient belonging to that case. In addition, the clinic's access control model reflects legal requirements by stating that patient data has to be deleted five days after usage and that storing it anywhere else is not allowed.

The patient, being the owner of the BAN, has configured the BAN's management service so that it grants access to health data to the patient himself and his family members. In addition, he requires every health data request to be logged so he can keep track of the BAN usage.

When combining these two domains, the PEP of the aggregation service is registered at both policy decision points and requests to the aggregation service require a decision from both, the hospital's and the patient's domain. Merging the policies of both domains is not a viable option: neither does the patient want to reveal the identity of his family members to the aggregation service, nor is it possible to directly match rules of the different policy models. However, if both the patient and the clinic want to support the aggregation service, there must be a way to align the access- and usage control requirements of the clinic with those of the patient.

7.4.2. Policy Model

At first, we look at how the requirements of both domains would be modelled with the policy framework introduced in this thesis.

Policies The clinic runs a RBAC model, extended by some usage control obligations. We can therefore take the DRBAC model from section 7.2 as a basis and add obligations and monitors for the specification of usage constraints and populate the model with use-case specific roles.

We assume that for each treatment of a patient, a dynamic role is created and assigned to the medical staff involved in that medical case. Users who are active in the role assigned to the case get *read* and *write* access to the patient's record.

Further, the DRBAC model needs to be extended in order to express usage control constraints. RBAC is merely an access control model, and does not include the specification of obligations. This means that with a pure (D)RBAC model it is not possible to declare obligations in order to require a subject to comply with some usage constraints. We therefore add an obligation assignment (OA) to the traditional RBAC user assignment (UA), and permission assignment (PA), as depicted in Figure 7.9.

In our example, we consider usage control obligations which allow to delete a datum after a certain timespan or to track accesses to the datum once it has been retrieved. In order to ensure that clients do in fact comply with the usage control requirements, we further add a monitor to observe deletion events.

$$
\begin{array}{rcl}
\{\mathit{read}, \mathit{write}\} & : & \mathit{AuthAction} \\
\mathit{treatmentPermission} & : & \mathit{Permission} \\
\mathit{recordJohnDoe} & : & \mathit{Resource} \\
(\mathit{caseJohnDoe}, \mathit{treatmentPermission}) & : & \mathit{hasPermission} \\
(\mathit{treatmentPermission}, \mathit{read}) & : & \mathit{hasAction} \\
(\mathit{treatmentPermission}, \mathit{deleteRecord}) & : & \mathit{hasObligation} \\
(\mathit{treatmentPermission}, \mathit{recordJohnDoe}) & : & \mathit{hasResource} \\
\mathit{MedicalCase} & \sqsubseteq & \mathit{Role} \\
\mathit{caseJohnDoe} & : & \mathit{MedicalCase} \\
\mathit{GP} & \sqsubseteq & \mathit{User} \\
\mathit{Administration} & \sqsubseteq & \mathit{User} \\
\mathit{drSmith} & : & \mathit{GP} \\
(\mathit{drSmith}, \mathit{caseJohnDoe}) & : & \mathit{hasActiveRole} \\
\mathit{deleteRecord} & : & \mathit{Obligation} \\
(\mathit{deleteRecord}, \text{"}5\mathit{days}\text{"}) & : & \mathit{hasParameter} \\
(\mathit{deleteRecord}, \mathit{deletionMonitor}) & : & \mathit{monitoredBy} \\
\mathit{deletionMonitor} & : & \mathit{Monitor}
\end{array}
$$

At the patient domain, simple access control rules based on the core model introduced in chapter 5 are used.

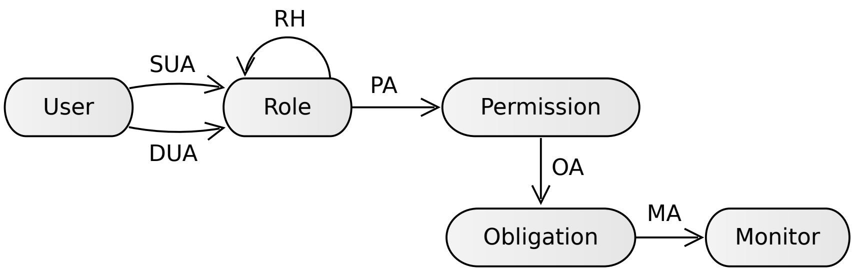


Figure 7.9.: RBAC model extended by obligations

$$
\begin{aligned}
\mathit{FamilyMember} &\sqsubseteq \mathit{Subject}\\
\{\mathit{lauren}, \mathit{john}, \mathit{marry}\} &: \mathit{FamilyMember}\\
\mathit{banDataJohnDoe} &: \mathit{Resource}\\
\mathit{read} &: \mathit{AuthAction}\\
\mathit{dec} &: \mathit{Decision}\\
(\mathit{dec}, \mathit{permit}) &: \mathit{hasEffect}\\
(\mathit{dec}, \mathit{dbLog}) &: \mathit{hasObligation}\\
\mathit{banRule} &: \mathit{Rule}\\
\left(\mathit{banRule}, \mathit{FamilyMember}^{\mathcal{I}}\right) &: \mathit{hasSubject}\\
(\mathit{banRule}, \mathit{banDataJohnDoe}) &: \mathit{hasResource}\\
(\mathit{banRule}, \mathit{read}) &: \mathit{hasAction}\\
(\mathit{banRule}, \mathit{dec}) &: \mathit{hasDecision}
\end{aligned}
$$

Metapolicies Both domains use metapolicies to express how decisions can be merged with that of another domain. In general, the clinic does not allow anybody access to the patient record who is not assigned to the medical case role (*caseJohnDoe*). However, for the sake of combining the record with health data from the patient's BAN, the clinic is willing to make an exception and to allow the patient himself to access the record.
As the clinic is obliged to adhere to legal regulations, it states that the requirement to delete patient data after five days is mandatory. It also refuses all obligations which would lead to an uncontrolled storage of this data.

```
:clinicMR rdf:a MetaRule ;
          :hasSubject :aggregationService ;
          :hasResource :recordJohnDoe ;
          :hasAction :read ;
          :hasStrength "weak" ;
          :hasAnnotation [
               :hasEffect "permit" ;
               :compulsory :deleteRecord ;
               :forbidden [ rdf:type :PersistentStorage ] ;
          ] .
```

The patient has configured his BAN management service so it grants only family members access to health data and logs every access. As the patient is interested in using the aggregation service, he sets up a meta rule which allows also the aggregation service to access the data and relaxes the constraints from logging to a database to any logging facility:

```
:patientMR rdf:a MetaRule ;
          :hasSubject :aggregationService ;
          :hasResource :banDataJohnDoe ;
          :hasAction :read ;
          :hasStrength "weak" ;
          :hasAnnotation [
               :hasEffect "permit" ;
               :alternative [
                    :replaces :dbLog ;
                    :by [ rdf:type :LoggingAction ]
               ]
          ] .
```

Whenever the aggregation service requests the clinic and the patient's BAN to retrieve data from them, the clinic's and the BAN's PDP will return annotated decisions which allow access for the aggregation service in general, but contain different sets of mandatory and forbidden obligations. The PDP of the clinic will answer with an annotation $dec_{meta_{clinic}} = \langle permit^{weak}, deleteRecord, \emptyset, PersistentStorage\rangle$ and the PDP of the BAN will return $dec_{meta_{BAN}} = \langle permit^{weak}, \emptyset, \{(dbLog, \{LoggingAction\})\}, \emptyset\rangle$.

When these results are merged into one applicable decision, the aggregation service will receive the following decision: $dec = \langle permit, \{deleteRecord, LoggingAction \sqcap \neg PersistentAction\}\rangle$.

This means that the aggregation service is allowed to access the patient record and the BAN data. In addition, however, there are obligations which must be executed by the PEP of the aggregation service: the first obligation (*delete*) requires the aggregation service to delete the data after five days. The second obligation does not refer to a specific action but rather to the set of actions which perform a logging operation on the one hand, but are not classified as *PersistentAction* on the other hand. A feasible obligation for this set might be a notification of the user via SNMP, for example.

Discussion This use case illustrates how policy decisions from multiple domains can be merged with each other, provided that policy authors of both domains are willing to relax their constraints for the sake of combination. But it becomes also obvious that such combinations will not work in all cases. If the patient would not have accepted alternative logging methods to *dbLog*, an unresolvable conflict with the requirement of the clinic for non-persistent storage would have been occurred. In this case, it would not be possible to combine the decisions of both domains and the aggregation service would not provide any data to the user.

Further, it becomes clear that all involved parties must share the same taxonomy of obligations (i.e., they must use the same ontology fragment). When the BAN refers to *dbLog* and the clinic to *PersistentStorage*, the aggregation service must know that *dbLog* is actually an instance of the concept *PersistantStorage*.

This means that the proposed approach does not enable any random policy domains to seamlessly work with each other, but that rather a common vocabulary for obligations is required, at least.

Nevertheless, the use case illustrates that none of the domains needs to reveal their internal policy and only annotated decisions are given to the aggregation service. This fact is actually what enables the two domains to work with each other in this use case: the patient can access the clinic's data for a dedicated purpose (i.e., for using it in the aggregation service) but the clinic does not have to reveal their policy to him, including critical information about employees and roles. Vice versa, the patient can keep his policy including information about his family members private and still be sure that the aggregation service sticks to it. This is a strength of the approach and distinguishes it from *policy combination* approaches aiming at the aggregation of whole policy sets.

7.4.3. Prototype

The mechanisms for annotating and merging policy decisions on the basis of metapolicies have been implemented in a prototype. Here, we briefly discuss the details of the implementation and discuss the practicability of the approach.

The prototype set up consists of the core policy framework with the BasicSemanticModule. This module provides the plugin for deciding the semantic ABAC policies of the core policy model (*ABACDecisionPlugin*) and provides methods for managing ontologies and reasoning over them to the other modules.
In addition, a MetaPolicyModule comprises the evaluation logic for meta policies, which in our simple use case, are based on a triple of subject, resource, and action and declare the compulsory, forbidden, and alternative obligations from the previous subsection. The MetaPolicyModule provides its evaluation method in form of a plugin for the post decision phase of the PDP, the *MetaPolicyPlugin*.

A small test service represents the aggregation service from the use case above and is protected by a PEP which is registered at two PDP instances, one representing the BAN, the other representing the clinic. Whenever a request to the aggregation service is made via R-OSGi, the PEP contacts both PDPs and receives back an annotated decision from each of them. The actual logic for merging decisions has been implemented in the PDP component, because on the one hand it has direct access to the reasoning engine anyway and on the other hand, the PEP should be kept as lightweight as possible, so it can be integrated at multiple places in the architecture without excessively increasing the overall footprint. So, when the PEP has received the annotated decisions, it contacts again one of the PDPs in order to merge them into one applicable decision.

The focus of the prototype implementation was to validate the feasibility of the concept and to test its practical applicability, rather than producing a lean and performance-optimised library. Therefore, we integrated the defeasible logic engine SPINdle, in order to keep the implementation as close as possible to the theoretical foundations from chapter 5. When a PEP requests the merging of two annotated decisions, the PDP first transforms the decisions and their annotations into two defeasible logic theories and then combines them into one theory, following the procedure described in 5.5.3 on page 90. It then retrieves all defeasibly and definite provable individuals and marks them as required obligations. Further, the PDP constructs a class expression out of the provable concepts by intersecting all provable positive and negated concepts. SPINdle is a pure defeasible logic engine and does not have any understanding of Description Logics. Thus, the PDP explicitly adds strict rules $C \rightarrow i$ for every individual i of class C that is contained in the defeasible logic theory. Further, class hierarchies are reflected in the defeasible logic theory by adding two strict rules $A \rightarrow B$ and $\neg B \rightarrow \neg A$ for every subsumption $A \sqsubseteq B$ (following the approach from [63]). In order to distinguish between individuals and classes in the defeasible logic theory, the prototype further assumes all small letter variables as individuals and all capital letter variables as concepts.

Applying this strategy to the use case above, and using the SPINdle reasoner on the merged dl theory, returns the provable conclusions from Table 7.7 (note that *definite provable* implies *defeasibly provable*, thus the duplicates). As can be seen, the individual $dbLog$ is explicitly forbidden (negated) and $deleteRecord$ is explicitly required (proved). In addition, the PDP constructs the class expression $\neg PersistentStorage \sqcap LogAction$ from the conclusion

Strength	Conclusion
$+\Delta$ (definite provable)	*LogAction* $\neg$*PersistentStorage* $\neg$*dbLog* *deleteRecord*
$+\delta$ (defeasibly provable)	*LogAction* $\neg$*PersistentStorage* $\neg$*dbLog* *deleteRecord* *effect* : *allow*

Table 7.7.: Provable conclusions of the merged decisions

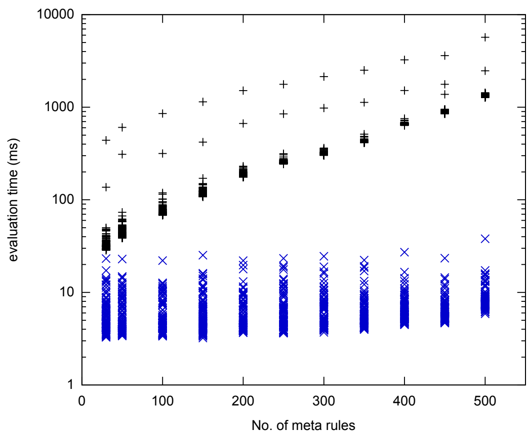
set and selects an individual from the ontology that matches the class expression.

7.4.3.1. Performance

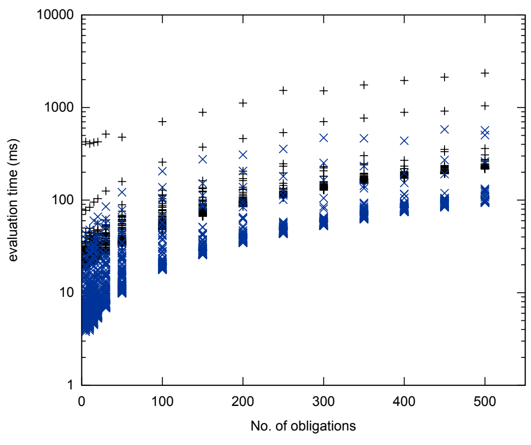
The prototype has not been designed with a focus on performance, so evaluating the absolute run time does not provide great insights, as it could be significantly improved by optimising the implementation. For example, it is certainly not the most efficient design to integrate a complete defeasible logic engine like SPINdle and to transform annotated policy decisions into its proprietary syntax before reasoning over it. However, it is interesting to investigate how the prototype performs with respect to scalability.

We therefore measured the average run time of the prototype implementation, including the evaluation of the access requests, transformation into the dl theory, as well as the actual composition process. PEP and PDPs were running in the same platform, so we do not have to consider network latency. As a platform we used a Pentium i7 2.7 GHz with 4 GB of RAM. Two factors are of interest when evaluating the run time of the approach: the number of meta rules and the number of obligations in an annotation. The former is relevant for the PDP when annotating the policy decision, because the more meta rules have to be evaluated by the PDP, the slower the annotation process will be. In contrast to that, the number of obligations influences the time for merging two annotated decisions, as with the number of obligations, the size of the defeasible logic theory, derived from the annotations increases.

Thus, in a first test the run time behaviour has been tested with sets from 30 to 500 meta rules and 100 test runs per set. Whether these are realistic numbers depends of course on the use case, but as meta rules are intended to be more generic than the actual access control rules, we deem the lower end of the range as more likely in reality. When looking at Figure 7.10a two main observations become obvious: first, the time for deciding an access request and annotating a policy decision increases about exponentially (note the logarithmic scale), while the time for merging the annotated decisions remains more or less constant. Second, the first runs in a test series are outliers which took significantly more time than the rest of the series. These observations are caused by the policy evaluation in Description Logic which maps to the concept satisfiability problem. Due to the internal caching of the Pellet reasoner, only the first runs in a series require significant more time, while the subsequent runs are significantly faster. As expected, the time for merging policy decisions remains constant with an increasing number of metapolicies.
Figure 7.10b shows the results of a second test, in which we kept the number of metapolicies



(a) Run time depending on the number of metarules



(b) Run time depending on the number of obligations

Figure 7.10.: Time for deciding access requests (+) and merging them (x) at 100 runs per test. Scale is logarithmic

constant, but continuously increased the number of obligations within the annotated policy decisions. In this test, the run time increases about linearly for both, the decision taking and the merging phase (note again the logarithmic scale). Again, the internal caching causes subsequent runs in a test series to be clearly faster than the first runs.

So, these tests confirm the theoretical claim that reasoning over defeasible logic requires linear time, while satisfiability checking in Description Logic is a significantly harder problem and can require up to NexpTime. In the context of the service composition approach, this means that the merging of multiple annotated decisions can be done quite efficiently and scales well with the number of services (i.e., policy decisions). In contrast to that, deciding and annotating access requests by reasoning over Description Logic can become heavyweight – especially if instances of complex classes need to be found. There are however different options to optimise the decision process. Apart from pre-computation and caching, the complexity of the policy evaluation can be reduced in different ways:

1. leaving out complex class expressions in metapolicies and rather relying on atomic types will on the one hand prevent users from describing subjects and resources by expressions like `SOME User THAT isFamilyMemberOf VALUE {Bob}`. On the other hand, it would however also remove the need for satisfiability checking for complex classes and result in significant performance improvement.

2. leaving out inference at all and only querying for direct individuals of classes would reduce the reasoning approach to a mere database query over a triple store and could thus be done in linear time (or even better, with appropriate indexing).

7.4.3.2. Practical Considerations

Apart from performance, another interesting question is which information must be shared between the PDPs. As stated above, the PEP invokes a PDPs to carry out the merging operation, but it is irrelevant which of the PDPs is actually in charge of this. Of course, this requires the PEP to trust both PDPs to correctly perform this operation. This is however not an issue, as this trust relationship is given by definition – the PEP has become part of both domains at the moment it has been registered at the PDPs. However, it also requires both domains to share some information. While the involved domains can keep their policies private, they need to share some information in order to merge their decisions. If the PDP in charge with the merging process is not aware of class memberships or hierarchies, it will not be able to correctly resolve forbidden and compulsory obligations and thus return a wrong policy decision. So, while policies and domain knowledge can be kept private to each PDP's domain, it necessary that the PDPs share the model of supported obligations. While it was out of the scope of the prototype to implement any autonomous exchange of ontologies between the PDPs, it is conceivable that the PEP acts as the intermediate component to send the "obligation ontology" from one PDP to the other one, respectively. As ontologies, being part of the Semantic Web, have been designed to be merged, combining two "obligation ontologies" will not create any conflicts: namespaces are used to distinguish axioms and the open world assumption guarantees that conclusions are not invalidated by adding further axioms.

7.5. Multilateral policy refinement and negotiation

The modules discussed so far show how increasingly complex policy models, addressing typical pervasive systems challenges, can be realised by extending the concepts of the core policy framework. We already discussed multi-domain scenarios and a possible conflict resolution strategy in the previous section. However, the strategy aims at finding *one* applicable policy decision which fits the requirements of all domains but does not take into account individual preferences. In pervasive systems, services have individual requirements and capabilities. Requirements have already been considered in the form of obligations. However, up to now we did not show how the capabilities of a service (or its underlying platform, respectively) can be taken into account when selection obligations. Also, we did not consider the typical trade-off between criteria like security, battery lifetime, and performance. While in most static service-oriented architectures, requirements are formulated as Service Level Agreements (SLA) and preferences are only considered in manual negotiations between service provider and consumer, this is obviously not possible in pervasive systems where services are orchestrated dynamically and without human interaction. Each domain might have its own preferences and in order for the policy framework to support truly "self-adapting" systems, it must be able to autonomously find an optimal set of obligations.

In this section, a *Multilateral Policy Negotiation* (MPN) module will be described, an extension module for the policy framework which allows to formulate *requirements* and *capabilities* to each service. When multiple services are orchestrated, i.e., combined to an application, the policy framework evaluates in a first step a set of common mechanisms which fulfil the requirements and capabilities of all involved services and in a second step selects a mechanism that best fits the individual preferences on categories like confidentiality, integrity, or performance. With this module, users can set up systems which autonomously adapt communication protocols or configurations in order to achieve an optimal configuration according to predefined high-level quality-of-service policies (e.g., in the form of SLAs).

The module consists of two main contributions: a negotiation using semantic descriptions of requirements and an optimisation component which selects a pareto-optimal solution out of the set of all possible configurations. An early version of the negotiation protocol has been integrated into the LinkSmart[23] middleware, developed in the Hydra project [7]. The

[23]`http://sourceforge.net/projects/linksmart/`

protocol design and the refinement process have been described in a paper by the author of this thesis [151], published in the proceedings of the SEC2010 conference. Regarding the optimisation approach, first experiences with a centralized genetic algorithms engine were again made in the Hydra project and have been published by author of this thesis in conjunction with others [198] in the proceedings of the ICSoC 2009 conference, however without considering the actual negotiation protocol. An alternative to genetic algorithms are micro-economics-based approaches like auction protocols. As they provide greater flexibility and can be carried out in a distributed way, an exemplary auction-based optimisation component has been developed for the MPN module. Its applicability has been shown by a prototype implementation conducted in the diploma thesis of Stephan Heuser [67], supervised by the author of this thesis. Further, the author of this thesis published results on the prototype implementation in [150].

This section is structured as follows: at first, an overview of the addressed problem will be given by means of a use case scenario and the solution is outlined at a high level. Then, related work regarding the semantic description of service requirements and capabilities, as well as corresponding negotiation approaches are discussed in subsection 7.5.2. After that, it is described how the core policy model from chapter 5 is extended by concepts for representing requirements, capabilities, and preferences for services. These concepts are then used by the exemplary auction-based optimisation component, discussed in subsection 7.5.4. Finally, subsection 7.5.5 describes the realisation of the multilateral policy negotiation in form of the MPN software module for the framework presented in this thesis.

7.5.1. Overview

In service oriented architectures, both, service provider and consumer may have their own requirements on how a service shall be consumed. For instance, the consumer might require confidential communication, while the service provider would like to limit the bandwidth and require consumers to authenticate themselves before requesting data from the service. Nowadays, such requirements are still either agreed manually before integrating a service into an application and technicians are responsible for setting up appropriate mechanisms to ensure compliance with these SLAs, or protocols such as WS-Policy [188] are used to negotiate mechanisms. However, as pointed out in the following subsection on related work, state-of-the-art protocols are rather limited and not well suited for an autonomic preference-based negotiation. In order to illustrate the problem, we consider the following scenario.

We assume a pervasive system middleware, connecting a number of services which run on different platforms and communicate over different protocols like Wi-Fi or 3G. One application on top of this infrastructure is a mobile collaboration tool that interconnects different mobile devices and allows them to interact with each other, for example by means of joining a common video chat or accessing a common document editing tool. In combination with the different platforms and connection types, a number of possible communication protocols could be used, each with its own characteristics. Considering that each service comes with its own preferences, the challenge is now to find a system configuration which matches each platform's capabilities and is an optimal compromise between the individual preferences, in contrast to a traditional SOA where SLAs are negotiated only between a single consumer and provider.

Solving this problem consists of mainly two steps: finding a common set of applicable mechanisms which match all requirements and capabilities, and selecting an optimal solution out of this set.

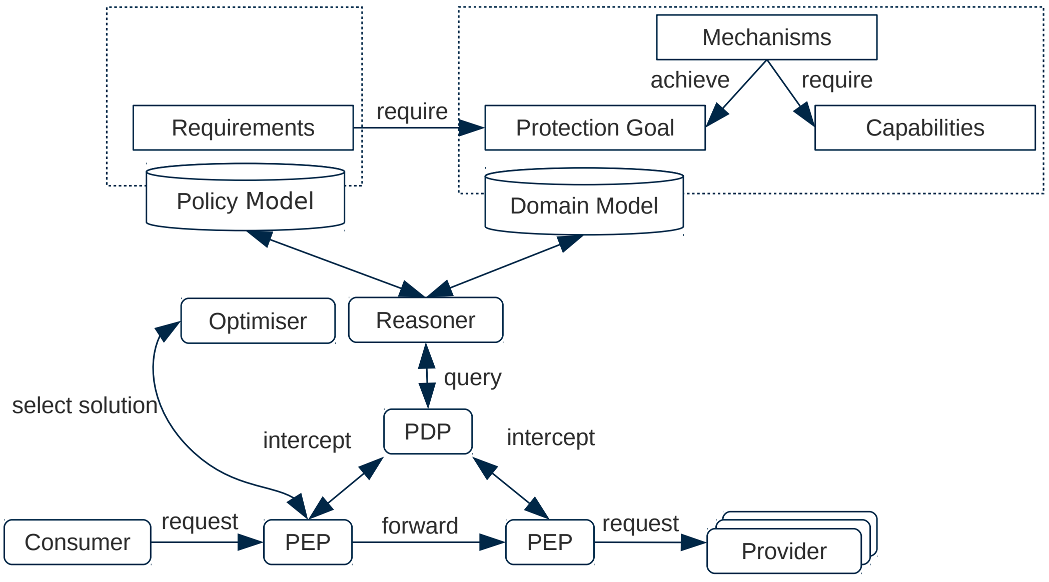


Figure 7.11.: Components and their interaction for multilateral security negotiation

For the first part, the core policy model needs to be extended by a semantic model of requirements and capabilities, as well as semantic descriptions of security mechanisms and their properties. Based on these descriptions, a negotiation protocol can then be applied in order to find a set of applicable mechanisms.
The second part requires for a multi-criteria optimisation component which takes the individual preferences as input and computes an optimal solution. As an example, and because it is one of the most generic approaches, we used an auction-based optimisation for the implementation of the MPN module. Conceptually, the specific algorithm is however not of importance and the respective component could simply be exchanged by a different one. For instance, in earlier experiments on self-adapting platforms, we have used an optimisation based on genetic algorithms [198].

Figure 7.11 depicts the information sources and shows how components interact during the negotiation process (for the sake of readability, all services share the same PDP). We will now describe this setup in more detail, beginning with the information in the *domain model*.

The domain model is an application-specific model which has to be shared by all domains. It is linked to the policy model and provides information about run time components such as devices, services, users, or the links between them. How this domain model has to be constructed is not predetermined by the policy framework. One option is to simply set up the domain model in form of a static ontology and load it into the PDP. However, in settings where services dynamically join the infrastructure, it might be more suitable to create the domain model from information which is provided by the services at run time, for example in the form of semantic annotations for web services using SAWSDL [54], OWL-S [187], or discovery protocols like SLP or UPnP. Independently from how information about devices and services is gathered, the policy framework itself is agnostic towards the data source and simply expects this information to be available via the reasoner interface.

One part of the domain model is the description of *capabilities* of services and their underlying platforms. This information will usually be provided by the services themselves

and be added to the domain model as soon as the service has been discovered. One way to achieve this in practice is to use service wrappers, providing meta data about a service, as it is done by the LinkSmart middleware. Another part of the domain model is the description of available *security mechanisms* and their properties. As this information is not expected to change rapidly, it is reasonable to provide it in form of a predefined ontology. Users would only have to change this ontology if either new mechanisms become available or if the classification of mechanisms with respect to the criteria changes – for example, when a flaw in a security protocol is discovered and the protocol mechanism has to be reclassified as "insecure" or "low security".
The *requirements* demanded by a service may either be static (such as in a traditional SLA) or may depend on dynamic factors like the identity of the consumer or any other conditions. They are therefore part of the policy model and are evaluated as part of the access control decision whenever a service is invoked or a consumer initiates a service call, respectively.

Service consumers can orchestrate multiple service providers by initiating a call and thereby opening a session. The outgoing call is intercepted by the consumer's enforcement point PEP_C which then queries the PDP for an access control decision on that outgoing call. This decision will contain an effect, possibly along with a set of obligations. If this set contains a dedicated *NegotiableObligation*, this indicates that consumer and provider have to agree on a common security mechanism, for example a communication protocol. In that case the PEP_C initiates the negotiation protocol:

1. PEP_C sends a session initiation request to the provider's PEP_P, containing the consumer's requirements req_C, the capabilities cap_C from the domain model, and the intended method call.

2. The session initiation request is received at the provider's PEP_P which queries its PDP for an access decision and retrieves P's capabilities cap_P from the domain model. Again, the decision contains an effect and a set of requirements req_P.

3. If the decision's effect is *allow*, PEP_P queries the PDP for a matchmaking, i.e. a set of mechanisms M that matches requirements and capabilities of both parties. This is done by querying the reasoner for all individuals of the complex class $fulfils.\{req_C\} \sqcap fulfils.\{req_P\} \sqcap requires.\{cap_C\} \sqcap requires.\{cap_P\}$

4. PEP_P sends M back to PEP_C and waits for PEP_C to select an appropriate mechanism.

At this point, the consumer can either just randomly select and apply one of the mechanisms or add more providers to the same session by executing steps 1-4 with them. In the latter case, the set of applicable mechanisms M' is the intersection of all individually supported sets, s.t. $M' = M_1 \cap .. \cap M_n$. If set M' is large enough, it will be reasonable to apply the second function of the MPN module – the selection of an optimal mechanism $m \in M'$. For this purpose, the *optimisation* component of the MPN module is invoked and receives references to the pending sessions. The optimisation component then retrieves the preferences of each service on the possible solutions and chooses a mechanism that matches the preferences best. What "best" means depends on the algorithm implemented by the optimisation component – we will concentrate here on an auction-based protocol to find a pareto-optimal solution, as explained in more detail in 7.5.4. Finally, if PEP_C has identified the mechanism $m \in M'$, it communicates it back to all providers and applies the mechanism by loading and executing it.

7.5.2. Related work

Before we describe the approach of this thesis on multilateral policy negotiation, we first review existing work in that area. Readers who are only interested in the solution may thus directly go to the policy model description in section 7.5.3. The state-of-the-art policy language for negotiating policies before starting an actual communication is WS-Policy [188], which, as the name implies, is mainly focused at web services. WS-Policy itself is just a framework which provides a basis for extensions with a specific purpose, such as WS-SecurityPolicy. With WS-SecurityPolicy a web service provider can require its clients to apply different security mechanisms before allowing them to use the service provider. For this purpose, policy statements are added to the provider's WSDL service description, including the set of required security mechanisms and information on the strictness of the policies. Clients then have to apply appropriate security mechanisms, such as XMLEncryption or XMLSignature configurations. Besides WS-SecurityPolicy, further extensions are available, such as WS-Coordination [124] or WS-ReliableMessaging [125] which support general agreements on SOAP transaction contexts, or on SOAP message reliability, respectively. Common to all of them is that it is merely the service provider who can choose a set of possible policies and leave it to the client to choose one out of multiple alternatives, if available. Also, the area of web services considers just a simple client-server communication, so WS-Policy does not support negotiation of policies between more than two parties. Also, the set of possible mechanism is predefined by the specifications, which means on the one hand that the policy language itself is bound to the functionality of the software and it is not possible to add mechanisms which are not considered by the policy language at a later time. On the other hand, the meaning of these mechanisms could remain unclear as they do not bear any additional semantics. For example, WS-SecurityPolicy allows to define required "ProtectionTokens" (i.e., X509v3 certificates with certain attributes), specific sets of algorithms like "Basic256", and configurations on how to combine encrypted and signed parts. Such technical information is of course required in order to apply the security mechanisms, but in many cases it will be too detailed for users who simply want to ensure that nobody can read or modify data during transmission. Various authors have made attempts to overcome this problem: in [183], the authors aim at matching WS-SecurityPolicy specifications from multiple services at a semantic, instead of a syntactic level and propose to extend WS-Policy by semantic operators and descriptions to allow the matchmaking. More recently, in [164], a similar approach to the same problem has been proposed which represents policy assertions by OWL concepts and maps the matchmaking to a satisfiability test of the merged concepts. Besides the integration of semantics into the existing language, Parsia and Kolovski et al. [132, 93] have proposed to represent WS-Policy in OWL to support reasoning over the policies themselves.

P3P (Platform for Privacy Preferences Project) [189] is another language for expressing non-functional requirements for communication. As the name suggests, it is however explicitly focused on expressing privacy profiles of the provider. Because P3P does not consider any negotiation protocol, the consumer can only either accept the privacy profile or decide to not use the service. Just as for WS-Policy, P3P is limited to a two-party client-server model, does not use semantic descriptions, and does not allow for preference-based trade-offs between multiple parties. Nowadays, P3P must be be regarded as outdated. Its final specification has been proposed in 2006 and due to lack of interest of browser manufacturers to integrate P3P, work on P3P has been suspended in 2007.

Lamparter proposed auction-based trade-offs in [98] and [99] with the aim to create a marketplace for web services where clients can choose one out of a set of possible web services which best suits their requirements in terms of non-functional criteria like price, performance, availability, etc. Although the addressed problem differs from the one considered in this section and Lamparter's approach is focused on web services only, the general idea of carrying out auctions over preferences has also been applied to our MPN module.

While auction protocols have traditionally been used before to solve scheduling problems (e.g. like in [41]), the application for policy negotiations in service-oriented environments seems to be promising as well. The Carisma middleware [29] makes use of auctions to find an optimal trade-off between multiple services and addresses in this respect a similar problem as we describe in this section. However, Carisma does not use a semantic model and therefore does not allow to formulate policies at increasingly abstract levels. Further, it relies on a single auction algorithm – a sealed-bid, one-shot model (see following section for explanations) – and particularly considers the underlying middleware to be fully trusted, i.e. it assumes that participants cannot create "virtual money" that would allow them to make bids on their own. In contrast to Carisma, the module presented herein has a framework character, i.e. it supports various auction models and optimisation mechanisms and provides security mechanisms to ensure that peers cannot create their own money to manipulate the result of an auction to their benefit.

7.5.3. Policy model

At first, we describe the extensions to the core policy model that are provided by the multilateral policy negotiation (MPN) module.

Because not all obligations will require a multilateral negotiation, and we do not want to constrain generic obligations, we leave the *Obligation* concept untouched and rather create a subconcept to describe obligations which need to be negotiated with another party.

$\mathit{NegotiableObligation} \sqsubseteq \mathit{Obligation} \sqcap \mathit{hasProperty}.\mathit{ObligationProperty}$

Individuals of the *NegotiableObligation* concept may refer to mechanisms, i.e. software modules which can be hooked into the middleware. For instance, a *TLS* component could hook into the communication layer and apply the TLS protocol. These obligations may have certain properties which can be referenced when writing a policy. For the prototype of the MPN module, we consider especially the performance aspects and protection goals which are achieved by the mechanisms. For example, TLS would achieve protection goals like *confidentiality*, *message integrity*, but not *non-repudiation* (because it does not include trusted timestamps). If *ObligationProperties* would be directly assigned to mechanisms, we were not able to express to which degree the protection goal is achieved, and it is obviously not realistic to state that a simple CRC checksum achieves message integrity just as good as a signature or HMAC. Therefore, *NegotiableObligations* are assigned to protection goals in combination with a strength. As OWL (and any other semantic web language) does not support *n*-ary relations, we use the *n-ary relation pattern* from the respective W3C working group note [117] and define an intermediate concept which refers to an *ObligationProperty* and a *strength* value, e.g.:

$\mathit{HighConfidentiality} \equiv \mathit{hasProperty}.\{\mathit{confidentiality}\} \sqcap \mathit{strength}.\{3\}$

Users can now define preferences and policies over these properties. Preferences indicate which properties are favoured by the user and policies indicate which obligations must be achieved. The MPN module does not contain a policy model, but can be combined with any model which supports issuing obligations upon a decision request, as the core ABAC model does, for example. For the prototypical implementation of the MPN module, we simply used the core model to specify obligations which have to be carried out upon calling a service provider — both, at the consumer as well as at the provider side. Obligations can be specified at different levels of abstraction, depending on the user's needs. For example, it is possible to require a $\mathit{TLSv}1.1$ obligation, which refers to a concrete implementation and does not provide room for any negotiations. On the other hand, users could also simply model an obligation by the complex concept $\mathit{hasProperty}.\mathit{Confidentiality}$ and thereby allow any mechanism that somehow achieves confidentiality – irrespective of its strength.

With *Preferences* user can assign priorities to *ObligationProperties* to express which of the properties are more important in the case that multiple *NegotiableObligations* with different properties are applicable. As opposed to the declaration of obligations and their properties, preferences are not application- but user-specific and are therefore part of the policy model which does not need to be shared among all peers. A preference simply assigns a value within a predefined range to a property, so the *Preference* concept is defined as follows:

$\mathit{Preference} \equiv \mathit{hasProperty}.\mathit{ObligationProperty} \sqcap \mathit{value}.\mathrm{integer}$

As an example, a preference for confidentiality in favour of data throughput could thus be expressed as:

$$\begin{aligned} \mathit{pref}_{\mathit{conf}} &\in \mathit{hasProperty}.\{\mathit{confidentiality}\} \sqcap \mathit{value}.8 \\ \mathit{pref}_{\mathit{tp}} &\in \mathit{hasProperty}.\{\mathit{throughput}\} \sqcap \mathit{value}.3 \end{aligned}$$

Further, users can assign requirements and capabilities from the domain model to an obligation. Requirements are assigned to obligations and indicate if a specific obligation puts any demands on the features of the underlying platform, such as the presence of specific hardware (a face recognition module might require a camera) or a specific software platform (an OSGi module might require the *CDC/PersonalJava* profile). This is simply modelled by adding a *requires* property to the *NegotiableObligation*:

$\mathit{NegotiableObligation} \sqsubset \mathit{Obligation} \sqcap \mathit{hasProperty}.\mathit{ObligationProperty} \sqcap \mathit{requires}.\mathit{Feature}$

In order to match requirements with the actual supported features, we need to describe services and their capabilities in the domain model. As services might join the system at run time, such information will usually not be provided in a static way at design time, but rather services will provide descriptions of their own which will be extracted and merged into the domain model whenever a service joins the policy domain. Here, each service will be modelled as an individual of the *Resource* concept and gets assigned its capabilities by the *provides* property:

$\mathit{Service} \sqsubset \mathit{Resource} \sqcap \mathit{provides}.\mathit{Feature}$

Both, requirements and capabilities, are not absolutely necessary to set up a multilateral negotiation scenario and can be skipped if the respective information is not available or the functionality of adapting obligations to specific platform capabilities is not required. Manually modelling the properties of each service and researching the requirements of each obligation mechanism would for sure be a tedious task and the necessary effort would probably exceed the benefits. However, if available, the MPN module takes into account this information and indeed, pervasive systems middlewares like LinkSmart or AMIGO do provide semantic descriptions of service capabilities out of the box so that the presented model could easily be applied to them.

With all these extensions of the core policy model in place, it is now possible for the policy framework to identify a set of mechanisms which fulfil the obligations demanded by all n peers $\mathit{service}_i$ and match their requirements req_i and capabilities cap_i. This is done by querying for all individuals which satisfy the following complex class:

$O \equiv \mathit{NegotiableObligation} \bigsqcap_{i \leq n} \mathit{requires}.(\mathit{cap}_i.\mathit{provides}^{-}.\mathit{service}_i)$

The result is a set of suited obligations $o \in O$. If $O = \emptyset$, no common mechanism could be found and the request will be treated as if it would have been immediately denied by the policy. Depending on the size of the domain model, especially the supported obligation mechanisms, and the abstraction level of the required obligations, $|O|$ can get quite large so that it makes sense to not just randomly chose one obligation out of O but to select one that fits the preferences $pref_i$ best, as described in the next sub-sections.

7.5.4. Micro-economic approach for solving the optimisation problem

The problem of finding an optimal obligation mechanism according to individual preferences can be regarded as a multi-objective optimisation problem (MOOP), i.e. a set of multiple objectives for which a Pareto-optimal solution has to be found. A solution is called Pareto-optimal if it is not *dominated* by any other solution, i.e. if no solution exists that is better with respect to at least one criterion of the problem and not worse in any other criterion. This implies that a single problem can (and in most cases will) have not just one but rather a set of optimal solutions, and finding at least one of them is the goal of the optimisation component of the MPN module.

Different approaches to MOOP exist. The simplest one is to aggregate all individual functions of the MOOP to a single function, possibly by weighting them according to their importance. This has however some drawbacks: the solution will depend on the arbitrary weighting, the combined function is often complex and non-linear so that finding an optimal solution is only possible if the original MOOP consists of only few functions (i.e. criteria).

An alternative are evolutionary approaches such as genetic algorithms. They start with a random set of tentative solutions which is continuously improved in multiple *generations* by *mutating* and *crossing* solutions and keeping those which match a fitness criteria (e.g., non-dominated solutions) for the next generation. Genetic algorithms are usually slower that a simple weighted aggregated functions but they are universally applicable and tend to converge towards the *true Pareto front*, i.e. the set of all Pareto-optimal solutions.
Using genetic algorithms to select an optimal preference-matching set of obligations would be a feasible approach, and in fact, has been applied in an early version of the MPN module, which has been published in [198]. Genetic algorithms however require that the optimisation component knows the preferences of all peers, as they build the different objective functions that make up the MOOP. As a result, each peer would be obliged to reveal its preferences, which might be a problem in some scenarios, for example if they are considered confidential as part of an internal policy or for technical reasons, because transferring large preferences sets from multiple peers over the network is not desired.

As an even more flexible alternative which allows peers to keep their preferences private, the MOOP can be solved using approaches from the intersection of game theory and microeconomics. Here, the approach is to model the MOOP as a game and let each peer act as a player. Players are supposed to act rationally and to try to optimise their personal outcome of the game, i.e. they will select a *dominant* strategy — a sequence of steps according to the rules of the game that will result in a solution which gives the maximum benefit to the player, independently of the other player's strategies. Different criteria are available in order to tell if a strategy set is "good" or not. We will only briefly mention here the Nash Equilibrium and Pareto-optimality:
In a so called non-cooperative game, i.e. a game where players do not collaborate in order to gain a higher benefit, a set of dominant strategies ("action profiles") chosen by all players is called the *Nash Equilibrium*. It determines the situation where no individual player would benefit from changing his strategy, while all other players would stick to their chosen strategy.

> A *Nash equilibrium* is an action profile a^* with the property that no player i can do better by choosing an action different from a^*, given that every other player j adheres to a^*. [126]

So, the Nash Equilibrium defines a stable situation, i.e. a situation where no rational player would change his strategy because it would not improve her outcome. This does however not automatically imply that the chosen strategies lead to an "optimal" solution. Here, the definition of *Pareto-optimality* is better suited: a strategy set is called Pareto-optimal if it leads to a Pareto-optimal solution, i.e. a solution where no criterion could be further improved without decreasing the result for a different criterion. More formally, a strategy combination s^* is called pareto-optimal, if no strategy combination s^*_{better} exists that results in a better result with respect to the utility function u_i for any player i, i.e. there is no s^*_{better}, s.t. $u_i(s^*_{better_i}) \geq u_i(s_i) \forall i, s_i \in S$.
For our purpose – finding obligation mechanisms which suit individual preferences – Pareto-optimality would be suitable criterion, as it provides solutions which are "as good as it gets" for every individual peer.

Modelling game strategies so as to achieve results with certain properties (reaching Nash Equilibria, maximised overall welfare, Pareto-optimality, etc.) is a research area on its own and is called *mechanism design*. For the sake of solving the MOOP in the MPN module, different game-theory based approaches, as well as traditional genetic algorithms could be used and it depends on the requirements of the application which one would best suited. For the realisation of the MPN module prototype, we have chosen to rely on an auction-based protocol whose parameters are configurable to a great degree, because auctions are an interesting and flexible approach to solve optimisation problems in multi-party scenarios.

A plethora of auction protocols is available and has been extensively discussed in literature. The general set-up is usually the same: one auctioneer opens the auction and is responsible for receiving bids, closing the auction, and calculating the winner and the price. Other aspects, however, such as the question if prices increase or decrease during the auction, if bidders are allowed to give more than one bid, the question if bids are open (i.e., visible to other bidders), or the way how prices are calculated depends on the type of the auction. An in-depth discussion of different auction mechanisms is out of the scope of this section, thus we just highlight the most common auction types. Most factors can be configured in the optimisation component, so the prototype of the MPN module can be used to apply multiple auction types and even votings, i.e. auctions where every bidder is limited to one bid and no virtual currency is involved.

English auctions use increasing bids from all bidders. If the last bid is not increased within a time span, the bidder with the highest bid wins the auction and has to pay the bidden price.

In a *Dutch auction*, the price decreases in fixed intervals while the auctioneer waits for bids. The first bidder wins the auction and pays the bidden price. In a network setting, one clear advantage of a Dutch over an English auction is obviously that only very few messages are required.

In an *n-th price sealed bid auction*, each bidder just submits a single bid, without knowing the bids of the other participants. When the auctioneer has received all bids, he selects the bidder with the highest bid as winner of the auction and sets the price to that of the n-th bid – thus, the winner has to pay a lower price than his actual bid.

Among the n-th price sealed bid auctions, the *Vickrey auction* is the best known variant. It is a sealed bid second-price, one-shot auction, i.e. every participant makes a single bid which is invisible for the other participants. The one with the highest bid wins the auction and has to pay the price of the second highest bid. Online auction platforms like eBay often use this type of auction (or slight modifications of it), because they maximise the actually paid price [184, 140, p. 397].

Finally, another interesting variant are *all-pay* auctions. Here, each bidder is obliged to pay the price of his bid, no matter whether he has won the auction or not. From an economical perspective, this variant is attractive because it avoids deadbeat bidders who just bid in order to increase the price but do not align their bids with the value they attribute to the good. From the technical perspective, which is more relevant in our context, all-pay auctions are promising as they lower the differences in the participant's "money" accounts, as shown in [67]. This is a helpful property in scenarios where an auction is not carried out just a single time but multiple auctions take place repeatedly, for example in a setting where new devices join a network and need to be integrated into an application, requiring a new policy negotiation. As each peer has to pay the bidden amount, even peers who never win an auction cannot rapidly accumulate money which they could use at a later time to score off the others.

The optimisation component implemented in the prototype of the MPN module provides a generic software infrastructure to set up most auction types and even some voting algorithms by configuring its parameters. Due to the limited amount of messages that need to be sent over the network, one-shot auctions are the most interesting ones for our purposes and especially an all-pay variant has been shown in [67] to provide more stable results than a Vickrey auction when multiple rounds are carried out repeatedly. Therefore, we will use an all-pay auction in this section in order to illustrate the protocol flow:

Let A be an auctioneer and I the set of peers (bidders) $i \in I$.

1. A starts the auction and calls for bids
2. Each peer i makes a bid b_i on an obligation, according to his internal utility function u_i.
3. When every peer has made his bid, the auction is closed.
4. A orders all received bids by their price $p(b_i)$, with $p : B \mapsto \mathbb{N}$, s.t.
   $B_v = \{b \in B : p(b) = v\}$ determines the set of all bids with price v and $B_i > B_j \iff i > j$.
   The result is an ordered sequence of bid sets: $B_{max} > ... > B_j > ... > B_{min}$
5. A determines the winner w by randomly selecting a $b_w \in B_{max}$ and collects the bidden price $p(b_i)$ from all peers.
   (This step might vary for a Vickrey auction)
6. A adds a fixed reimbursement amount r to all peer's accounts to ensure that peers do not run out of "money" during repeated executions of the protocol.

In the following, we will now see how such an auction protocol can be put into practice and integrated as the second steps in the aforementioned multilateral negotiation protocol.

7.5.5. Realisation of the multilateral negotiation module

In order to use the multilateral policy negotiation (MPN) model, we developed a module for the main framework which interprets the above-described model and carries out the negotiation protocol. In this section we will discuss the necessary components and have a detailed look at the implemented negotiation protocol. An overview of the protocol if given by the sequence diagram in Figure 7.15.

The MPN module consists of a set of components, shown in Figure 7.12. At first, it provides the policy model from section 7.5.3 in form of an ontology fragment to the decision point, so that at the time when the multilateral negotiation module is loaded, the PDP reads the policy model from it and merges it with the other models.

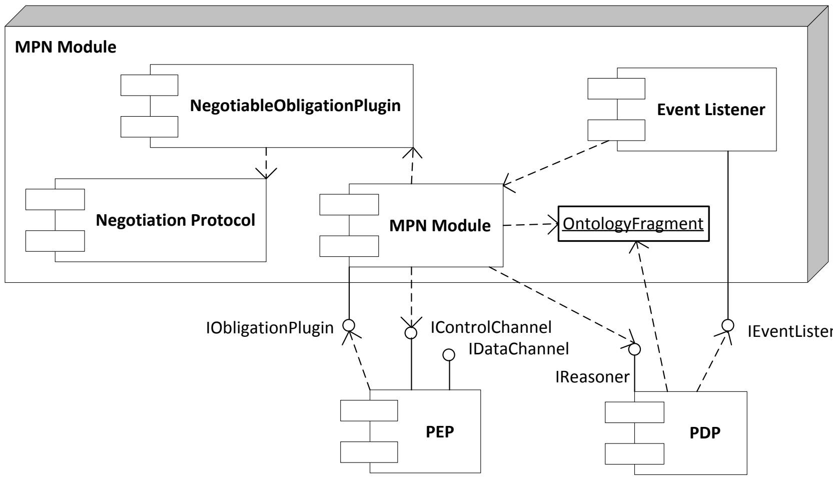


Figure 7.12.: Components of the multilateral policy negotiation module

The negotiation functionality is contained in an `NegotiableAction` plugin which is subdivided into further components and implements the `ExecutableAction` plugin interface. When the MPN module is loaded into the framework, it registers the `NegotiableAction` plugin as an implementation for the *NegotiableAction* concept. So, whenever a PEP receives a policy decision containing a *NegotiableAction*, it will search the component registry for a matching implementation and invoke the `NegotiableAction` plugin's `execute` method.

Within that method, the actual negotiation protocol is carried out in two parts: first, the set of common obligations which are supported by all peers is identified, and second, an optimisation sub-component selects one optimal solution out of this set. The first part of the protocol requires to know requirements and capabilities of all other peers and access to the PDP's reasoner in order to find obligation instances. For this purpose, the PEP first opens a session to all peers contained in the access request's resource list (requiring that they run the `NegotiableAction` plugin as well) and asks for the peer's requirements and capabilities, thereby acting as the *initiator*. While capabilities are usually static and depend on the platform a peer runs, requirements are part of a policy decision and may thus vary depending on the actual access request. As parameters of the remote method call might be private and should not be revealed before the security mechanisms have been established, the PEP strips them off and sends only the plain access request to the peers, using the control channel. So the negotiation protocol is kept separate from the actual data channel and can be implemented without putting any demands on the transport protocol of the underlying middleware.
Peers respond with their requirements and capabilities. The `NegotiableAction` plugin of the initiator then determines a common set O of obligations which suits all peers' requirements and capabilities. Depending on its configuration, the `NegotiableAction` plugin either just chooses one $o \in O$, applies the respective obligation, and communicates o back to the peers which will then apply o likewise. Otherwise, if the `NegotiableAction` plugin is configured to consider individual preferences in the decision process, the second phase of the protocols starts.

During the second phase, a multilateral negotiation protocol is carried out between peers

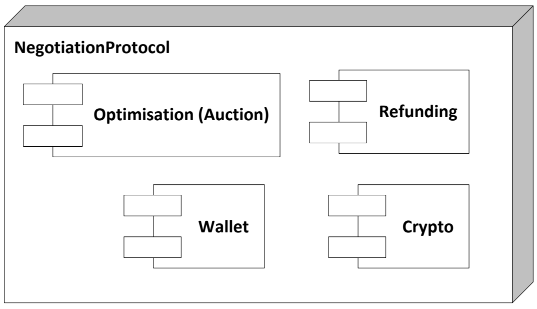


Figure 7.13.: Sub-components for the negotiation protocol

and finds the obligation which best matches all preferences, without requiring the peers to actually reveal their preferences. For testing purposes and to prove that also complex interactions between peers can be integrated as an obligation, the negotiation has been realised by an auction protocol which has been prototypically implemented in [67]. Setting up an auction protocol requires mainly four components:

One component (*Optimisation*) needs to coordinate the actual protocol. On the initiator side this components plays the role of the auctioneer, while on the peers it corresponds to the role of a bidder. In the prototype implementation, the Optimisation component realises the generic auction protocol, shown in Figure 7.14, which can be used for different variants so that changing the used auction variant from an English auction to a Vickrey auction, for example, does not require a new implementation but can be achieved by a mere change of configuration.

Because auctions rely on virtual "money", which is initially distributed to the peers and can be used to make bids, a component to store and manage balances is required on each peer – in Figure 7.13 it is denoted by the *Wallet* component.

Further, after an auction has been carried out, prices need to be paid and balances must be updated to avoid that peers run out of "money". It is the responsibility of the *Refunding* component to take care of this clearing process. The Refunding component can be configured to either continuously distribute money at a fixed interval to peers, or to be explicitly triggered by the auction component. The reimbursement amount depends on the selected auction model (first- vs. *n*-th price) and is set by *RefundModifiers* which are registered at the Refunding component and alter the paid amount according to an auction's result.

The amount of money available to one peer determines its maximum influence in a negotiation process. By bidding significantly more "money" on its preferences set, a peer can overrule others and thereby distort the original goal of a "fair" solution. It must therefore be avoided that peers can create money on their own, a problem that is not considered by the Carisma middleware [29], which also uses auctions for policy negotiations, for example. Carisma simply assumes all middleware instances to be trustworthy – an assumption which might hold in certain scenarios but is not valid in the general case. For this purpose, the exemplary auction implementation in the MPN module provides mechanisms contained in the *Crypto* component, to ensure that amounts are *authentic*, *integer* and *fresh*, that is, peers shall not be able to create or modify amounts on their own, as well as it should not be possible to re-use a possibly higher balance from a previous auction. This is done by calculating an HMAC over the amount messages, including a sequence number to

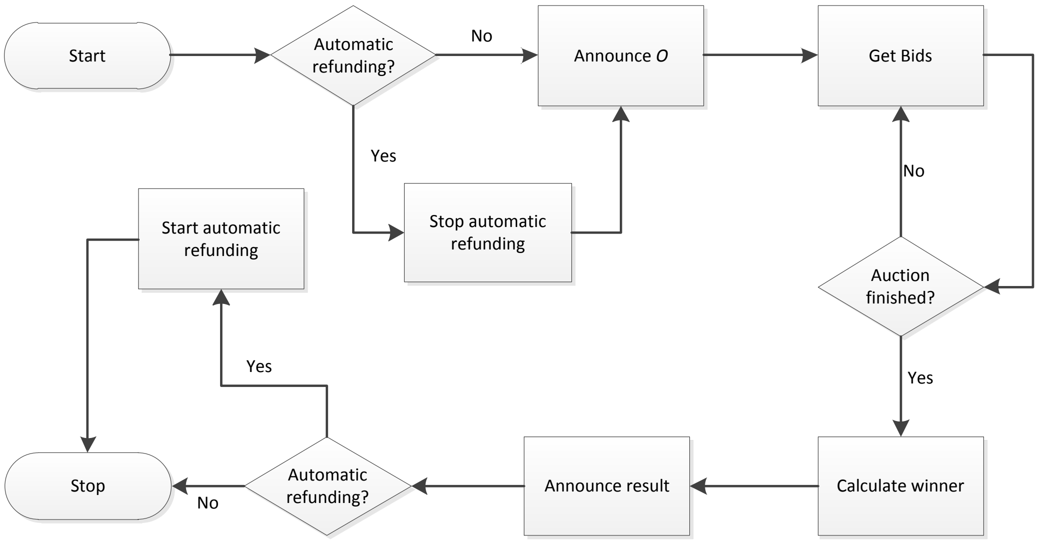


Figure 7.14.: Process of a generic auction

guarantee freshness.
In some scenarios, where auctions are repeatedly used to continuously negotiate parameters between multiple peers, one additional conceivable attack is to forcibly eliminate the initiator. The effect of this denial-of-service-attack would be that the amounts which have been built by the peers during the previous auction rounds would not be usable any more, because the symmetric key used for verifying their integrity would have been lost. As a result, peers which have used up most of their money in previous rounds would not be less well off compared to peers which have “saved” their “money” by acting more modest, as it would be fair. As an approach to avoid this threat, the prototype implementation from [67], uses a Shamir Secret Sharing [157] scheme for splitting the symmetric key into n parts of which a configurable number k would be sufficient to restore the original key. The parts are distributed to the peers when starting the negotiation sessions and when the initiator is not reachable anymore, k peers would collude to restore the key and determine a new initiator who then sets up a new session, taking each peer’s current amount as a starting point. While this functionality is provided by the Crypto component, we did not implement a protocol to recover from DoS attacks on the auctioneer, as it was not necessary to show the practicability of the MPN module in our framework in general.

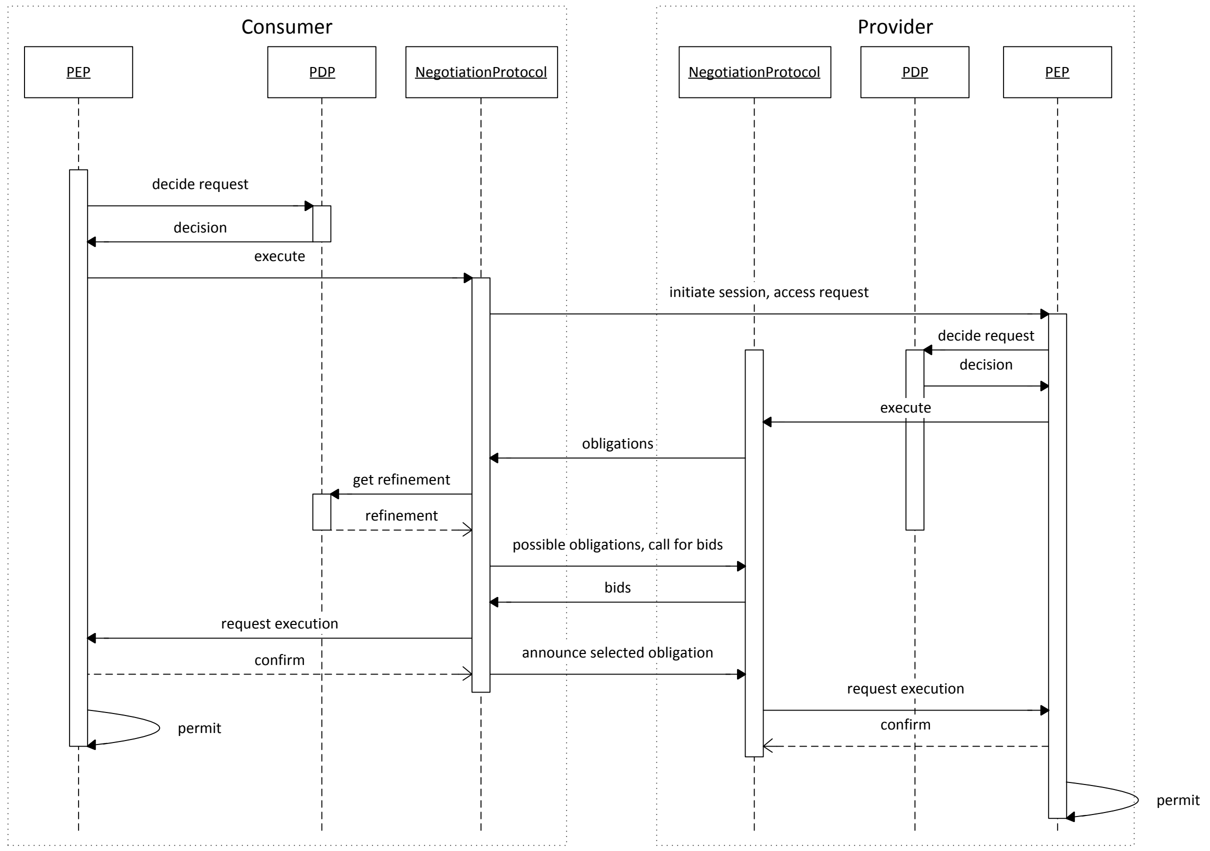


Figure 7.15.: Preference-based negotiation of obligations between a consumer (initiator) and a provider

7.5.6. Prototype

In this section we have described an extension module for the policy framework which implements a multilateral refinement of high-level obligations and carries out a negotiation protocol to select one obligation that matches the individual preferences of the involved peers best. For the negotiation, we rely on an architecture for different auction-based protocols – a very flexible, yet communication-wise more demanding multi-criteria optimisation method than genetic algorithms, for example. Nevertheless, the architecture of the module allows any multi-criteria optimisation method to be integrated.

The reason why we developed this module is to show that policy decisions do not necessarily have to refer to a single entity or to a client-server scenario, but that a negotiation between any number of participants is possible. This way, the policy framework allows a multilateral refinement of high-level obligations, like "ensure confidentiality of communication", under consideration of contradicting preferences like "ensure a good throughput", for example. Standard policy protocols for such quality-of-service requirements (e.g., WS-Policy) neither consider multi-party scenarios nor negotiations and research-based approaches like the Carisma middleware do not take into account the refinement of semantically defined obligations, nor the security in auction protocols.

A prototype of the auction-based negotiation protocol has been implemented as part of the diploma thesis of Stephan Heuser [67], supervised by the author of this thesis. The implementation was done on top of the Felix OSGi component framework, which has been ported to Android before. A test set-up of the auction protocol has been made, running the initiator on an Intel Core 2 Duo 2.26 GHz laptop, two peers on an HTC Dream (528 MHz, 192 MB RAM) smartphone running Android 1.6 and one peer on a Motorola Milestone smartphone (550 MHz, 256 MB RAM) running Android 2.0. The peers were connected to each other over IEEE802.11g, services were realised with a modified version of R-OSGi which has been "Dalvik enabled" to run on the Android platform and service discovery was done by the SLP protocol, running an OpenSLP server on the initiator. For generating HMACs and exchanging keys, AES-128 and RSA-1024 was used, based on the bouncycastle library[24]. In this set-up, the execution time for a negotiation round averaged to 3.7 seconds, where the performance hindrances are the execution within OSGi on top of the Dalvik VM, as well as the cryptographic operations.

7.6. Summary

In this chapter, we presented the prototype implementation of the core policy framework and gave several examples on how the core policy model can be extended in order to meet typical requirements of pervasive systems.

The prototype implementation of the software architecture has been done to investigate the feasibility of the design and to gain experience with its practical application. To leverage the component based design, the OSGi platform has been used as a basis and after an investigation of different remote service protocols, R-OSGi for service invocations over the network and mDNS for discovery has been applied. For accessing and reasoning over ontologies, the Pellet reasoner has been chosen, in combination with the OWL-API for ontology modification. Pellet turned out to be not the fastest reasoner available, but has

[24]`http://www.bouncycastle.org/`

been chosen as it closely sticks to the semantics of description logic, as opposed to others, like the RDF-based Sesame framework for example.

Researching suitable ways of user interaction with the framework is not in the focus of this thesis, but it is obviously important that there are easy to use interfaces for controlling the framework and writing policies. The developed prototype offers two ways users can interact with the framework: first through a console interface that consists of a set of commands on top of the basic Equinox console and second through a graphical user interface (GUI). The GUI has been developed using the Eclipse RCP platform and connects to the policy framework through an R-OSGi access to the PDP service, so it is possible to control multiple remote framework instances from one local rich client.
While the console only provides commands for controlling the framework, the GUI can also be used for writing policies using a template mechanism that allows to write policies in a nearly natural-language style.

After the practical evaluation of the core policy model and the software architecture, we presented several extension modules which solve increasingly complex problems, arising in the context of pervasive systems.

At first, it was shown how the policy framework allows to construct more abstract policy models based on the predefined core model. For this purpose, concepts from the core model like *Subject*, *Resource*, and *Action* have been taken as a basis for a dynamic role based access control (DRBAC) model, which is able to switch role memberships in an event-driven way. Such dynamic role membership assignments are important in context-aware systems, where role memberships can change rapidly, depending on the current context.
To illustrate the ability of the model to verify and enforce consistency constraints, two typical RBAC constraints have been formulated: static and dynamic separation of duty (SoD). Both can be formulated as class expressions and will lead to an inconsistent model if the policy violates either of them. This is a clear advantage of formal (in our case, semantic) models over proprietary policy languages which cannot be checked for constraints automatically. Another advantage is the ability to reason about the model and run queries against it, as shown by the exemplary analysis queries against the DRBAC model, given in section 7.2.4. Since using reasoning for policy evaluation comes at the cost of performance, a performance evaluation of the DRBAC prototype has been conducted. The results indicate that while there is a clear performance degradation with growing size of the policy, the prototype still shows acceptable performance of about 20 ms for evaluating a policy of comparatively realistic size (1003 users), so that with a carefully designed implementation, the approach would be efficient enough for practical applications.

Extensions of the basic event-condition-action pattern have been discussed in section 7.3. Here, it was shown how the traditional ECA pattern can be abstracted to so-called Situation-Goal policies by which users do not have to anticipate what the system has to *do* upon occurrence of a certain event, but can rather express what the system has to *achieve* in a certain situation. This is a more abstract way of writing reactive policies and has the advantage that it is on the one hand closer to the actual intention of a user and on the other hand can be combined with a formal security model so as to make the system achieve different well-defined protection goals.
A simple security model has been used to demonstrate the ability of the extension module to automatically enforce protection goals in an adaptive middleware architecture. This model follows the general principle of UMLSec's underpinning by describing the actions an outsider can perform on components and links in the architecture and modelling protection goals by the absence of some of these actions. By means of a prototype implementation, we showed how this extension can be put into practice using the OSGi middleware with an ad hoc wiring of components.

Then, the framework's ability to cope with overlapping policy domains is discussed in section 7.4. In the presented example use case, one domain features a role based access control model amended with usage control restrictions of patient data, while the other one applies a simple Discretionary Access Control model, combined with the requirement to log accesses. By means of a prototype implementation, we describe how policy decisions from these domains can be aligned with each other and how users can apply the meta policy capabilities of the framework in order to control the degree of policy relaxation for the sake of domain collaboration. The prototype implementation is at first discussed with respect to scalability, confirming that the defeasible logic based merging process is significantly more efficient than the actual description logic based policy evaluation. Subsequently, prerequisites for its practical applicability are discussed.

Finally, the extensions for domain collaboration are taken one step further by taking individual preferences of domains into account and building a negotiation mechanism which allows domains to agree on obligations which comply to their policies and result in an optimal trade-off between the domain's preferences. This approach is especially interesting for building "self-protecting" environments which require reoccurring reconfigurations in order to stick to certain protection goals and at the same time have to optimise themselves towards factors like battery lifetime or performance. While the presented extension for the policy framework is extensible in itself so as to support different negotiation mechanisms, we relied on auction protocols for the prototype experiments. In contrast to traditional multi-criteria optimisation algorithms, auctions have the advantage that they do not require the individual parties to fully reveal their preferences to each other. This is in line with our argumentation that policies should be considered private and parties should not be obliged to reveal their policies to outsiders. Nevertheless, auction protocols require a higher communication overhead compared to a centralised computation and it depends on the specific application if the increased privacy is worth the costs.

So, with the extensions described in this chapter, we were able to show that the extension points of the framework architecture work as expected and that the framework provides a good basis for solving the specific challenges of pervasive systems with more sophisticated policy models.

CHAPTER 8

Conclusion and prospects

In this thesis we worked out the challenges of policy-enabling pervasive systems and proposed the design of a policy framework to tackle these challenges. A decade ago, policies were mainly used for access control. At that time, standards like PCIM and COPS emerged, leading to the belief that research on policies would soon deliver a unified and universally applicable policy language. Nowadays, though, the field is more diverse than ever. With the advent of pervasive systems, social networks, and mobile devices, data has become more and more distributed among various domains, and controlling it is not merely a question of setting up a single access control system. Instead, a variety of policy languages addresses different models for access control, policy-based management, self-protection, usage control, trust negotiation and management, or privacy concerns. For developers, this leads to the undesirable situation that they must integrate multiple policy frameworks into their pervasive system. The framework proposed in this thesis attempts to solve that problem by providing a common, extensible vocabulary in form of the core policy model, in combination with an extensible decision process. Taking the core framework as a basis, we then designed several policy modules for solving typical pervasive systems challenges.

We realised the framework and its extensions as prototypes and tested them in an OSGi-based middleware. By a dynamic RBAC model, we showed how concepts of existing policy models can be reused in order to construct more high-level models. By an extension for Situation-Goal policies, an advancement from the traditional ECA pattern we propose, we demonstrated how semantic-rich reactive policies can be realised with the framework. Further, we described the ability of the framework to handle overlapping policy domains and merge policy decisions across domain boundaries, controlled by meta policies. Finally, we proposed an extension for extending the collaboration of domains so as to support a multi-lateral negotiation of preferences for achieving an optimal trade-off between individual preferences on obligations.

With these prototypes, it could be shown that the framework can indeed be used for controlling security mechanisms in pervasive systems in way that is closer to the idea of "self-protecting" systems. The design of the framework was initially led by a set of requirements stated in chapter 4 whose achievements will now be critically discussed.

8.1. Discussion of requirements

The requirements from chapter 4 shall now be reviewed and discussed in the light of the developed policy framework.

Integration into underlying middleware The integration of the framework into a middleware layer depends apparently on the middleware architecture. In general, the proposed framework requires two generic integration points: subscribing to events and intercepting service calls and responses. Further, it is helpful if the middleware allows dynamic loading of components, so they can be used to load obligation code at run time. If the middleware does not provide these prerequisites, an integration of the policy framework would be hardly possible. However, as event distribution and service hosting is the main purpose of a pervasive system middleware, it can be assumed that the vast majority of such systems would be compatible with the policy framework. Although almost all experiments have been done with an R-OSGi middleware, typical middleware systems like LinkSmart [7], OSAmi (semantic web service based) [6], LooCI [79], RUNES [36], and Gaia [147] also provide the event- and service mechanisms required for an integration.

Controlling the security of pervasive systems The proposed policy framework does not feature a single policy model but rather relies on an evaluation of generic asynchronous or synchronous decision requests. As we have shown by different extension modules, this basis can be used to apply different access control schemes, as well as reactive policies. Thus, all typical models like attribute-, role-, or history based access control can be supported by the framework. Policies for reconfiguring the system can be supported as well, as shown by the extension of event-condition-action pattern in form of situation-goal policies. Another interesting area are policies for usage control. Although usage control has only been partly considered in the practical extensions, as part of the cross-domain use case in section 7.4, the framework fulfils the general requirements of typical usage control policies:
Continuity of policy control is achieved by reactive policies, as shown in the Situation-Goal policy example in section 7.3. *Mutability*, i.e. the ability to dynamically change attributes of subjects and objects is likewise supported, as illustrated by the Dynamic RBAC model in section 7.2, which changes role memberships in the model. A negotiation mechanism for usage control policies is not contained in the framework, but it could be realised in form of Retrieval plugins which are called during the Decision phase upon outgoing requests at the consumer's side and incoming requests at the provider's side. Finally, realising provider-side reference monitors is supported by the reactive policy model.
It should however be noted that one aspect of usage control is not supported by the framework described in this thesis: consumer-side control mechanisms which can not only monitor, but rather enforce a correct usage of data. Such mechanisms usually rely on trusted containers, sometimes in combination with Trusted Computing, and highly depend on the specific data types and capabilities of the underlying platform. Designing such digital rights management (DRM) mechanisms was not in the scope of this thesis and is therefore not considered in the framework.

Handling multiple policy domains The ability to deal with overlapping administrative domains is essential for a policy framework in pervasive systems. That is, the framework must handle situations in which one resource is controlled by two or more PDPs. While significant work on merging policies from different domains into a unified policy has been done by other authors, this approach turned out not to be promising in the context of this

thesis. At first, merging context-aware policies would presume a specific context model and thereby limit the extensibility of the framework. Secondly, we consider policies as private information and therefore strived for an approach which allows to merge policy domains which leaving the policy itself private to each domain.
Thus, the framework allows users to express how their policy decisions shall be combined with those of other domains. With the help of metapolicies, users can put constraints on this combination, which are added as annotations to the policy decision and considered when merging the decisions. We proposed a defeasible logic based merging strategy which either guarantees that a merged decision is in line with the constraints of all involved domains, or detects and handles unresolvable conflicts.

Extensibility The requirement of extensibility has extremely important and has been thoroughly considered during the design of the policy framework. Going beyond the degrees of freedom provided by most existing policy frameworks, the design of our framework does not only provide several extension points for retrieving information which is required for policy evaluation, but rather allows to extend the decision algorithm and the policy model itself. For this purpose, we built a generic as possible core model based on monotonic Description Logic – a formalism whose strength is to allow future extensions without invaliding an existing model. As a result, our framework allows reuse existing concepts and create increasingly abstract policy models from them, as well as implementing completely different model which suit the requirements of a specific application.

Policy comprehension and consistency It was not in the focus of this thesis to conclusively determine the usability of the policy framework, as this would anyway highly depend on the specific application and user target group. For this reason, no usability tests were made in this thesis and it is thus not feasible to claim that semantic-based policies are easier to write and to understand than policies in a proprietary language. However, we proposed several ways to formulate constraints on policies, as well to query the model and derive information from it using a reasoning engine. So, although we cannot claim an easier specification of policies, we can still maintain that the semantic modelling approach provides greater possibilities to inspect the model and to analyse it.

Non-functional requirements Security and performance have been stated as non-functional requirements. The security requirements on the framework itself depend on the specific use case and may vary from simple authorisations for administrative tasks to accountable policy decisions and secured messages. For this reason, it was decided not to overload the framework architecture with all possible security- and key management methods which might be required, but rather to rely on the security mechanisms which are provided by the underlying pervasive systems middleware. This way, the framework components can be hosted as services in the middleware and protected just like any other service.

Regarding performance, we did several tests with the extension modules from chapter 7. The results clearly show on the one hand that using reasoning as a policy decision mechanism comes at a cost in terms of decision speed and will most probably slow down an application notably. On the other hand, although no efforts on increasing the efficiency of the framework have been made, our tests revealed that the slowdown of the DRBAC model, for example, might still be acceptable for most applications and could become negligible in combination with appropriate caching mechanisms.
Also, our focus was to keep the implementation of the framework as close as possible to the theoretical foundation, which is why the Pellet reasoner, fully supporting DL reasoning

has been chosen. In a productive implementation however, it might not be relevant if the reasoning engine does in fact stick to a specific formalism, as long as all constructors of a specific policy model are supported, and therefore a more relaxed semantics could be used – up to simple queries over triple stores which have been proved to be very efficient in practice [23].

8.2. Contributions to research questions

The goal of this thesis was to address several research questions, as stated in section 1.2. We will now review the questions and summarize the contributions made in this thesis.

Which policy models are required in typical pervasive systems and in which way must a framework be constructed in order to support all these policy models? We reviewed the characteristics of pervasive systems and derived typical policies which must be supported. Since the thesis is not limited to a specific application domain, we did not propose a specific policy model, but rather a generic framework which aims at supporting different typical models. Specific to access control in pervasive systems is context-awareness and the necessity to deal with dynamic attributes (mutability). For usage control, *continuity* is important, i.e. the ability of the policy framework to constantly monitor the usage of data. The reactive design of pervasive systems requires for policies to adapt software architecture and configurations. Thus, reactive policies need to be combined with dynamically loadable actions.
Dynamic architectures and reactive applications can quickly lead to policies which are difficult to maintain. The proposed framework therefore allows to abstract from "technical" patterns like ECA and to create higher level policy models which are closer to the actual intention of an author.

How can policy conflicts, which occur in typical pervasive systems, be handled? We reviewed different approaches on merging policies from multiple domains. Most of them took the approach of combining multiple policies into a single, conflict-free one. Since this approach is however not promising for context-aware and mutable policies, we proposed a method to merge policy decisions on the fly, controlled by meta policies. This allows domains to keep their policies private, to use their individual models, and also to integrate specific context evaluation mechanisms without forcing them to use a common formal model of their policies or context descriptions.

How can extensible, analysable and non-ambiguous high-level policies be modelled? This question aims at overcoming the drawbacks of limited policy models and representations in proprietary languages. We proposed to represent policies in ontologies, thereby putting them on the basis of a logic formalism so that firstly, ambiguities in the specific can be avoided and secondly, the model can be checked for constraints. It is further possible to use standard reasoning engines for analysing and evaluating policies. However, because it was recognised that semantic reasoning alone would not be expressive enough for some policy models and also slows down the evaluation process notably, our framework uses a hybrid decision engine which relies on a semantic core model but allows to plug in additional customised decision modules. Another benefit of using ontology-based policies turned out to be the possibility for policy authors to reuse existing concepts and meta data from the middleware and to create more abstract policy models from it.

What are best practice design patterns for policy frameworks in pervasive systems? In order to cope with the dynamic of pervasive systems, our framework makes use of some design patterns which are not commonly found in existing policy frameworks.
At first, the framework neither comes with a specific policy model, nor with a predetermined evaluation algorithm, Rather, the evaluation engine is based on a workflow of individual phases, where each of the phases can be tailored by plugins, thereby allowing the implementation of custom policy evaluation algorithms.
Further, during the prototype implementation of the core framework, we used service hooks to be notified of new services and then to dynamically replace them with a proxied variant, whereas the proxy classes are generated on-the-fly based on the respective service interface. Such a dynamic service proxying is extremely helpful in pervasive systems, as the "traditional" way of manually integrating policy enforcement points into beforehand known services is not feasible.
Finally, our framework makes use of obligations for various purposes, ranging from simple logging functionality on to updating the policy model at runtime. Due to the variety of obligations and their dependency on a specific policy model, it was claimed as a best practice to make obligations dynamically loadable from some repository and to describe their properties in a semantic model. This allows the framework to choose appropriate obligations and to take into account individual preferences on quality of service criteria.

8.3. Outlook on future research

In this section, we give an overview of the foundations for future research built by this thesis. Future work based on the results of this thesis is twofold: on the one hand, the framework itself can be extended by further advanced policy models and interesting applications which leverage its ontology-based representation of policies are conceivable. On the other hand, it would be interesting to transfer the general concepts of this thesis to other areas, apart from pervasive systems.

8.3.1. Framework extensions

A strength of the policy framework introduced in this thesis is the support of arbitrary policy models. While we mainly concentrated on models for access control and policy-based configuration, it would be interesting to apply other policy models and investigate how they could profit from the framework's functionality and the description logic based policy models.

Trust negotiation Trust negotiation is a process of gradually revealing identity attributes and can thus be regarded as a privacy enhancement to attribute-based access control. Due to the nature of pervasive systems, where no or only few pre-established trust relationships between entities exist, trust negotiation could help to autonomously build such relationships between devices or users. Throughout this thesis we assumed that semantic descriptions of services and users are available and can directly be used when evaluating a policy. However, by means of respective Retrieval plugins, it would also be possible to support trust negotiation protocols for the exchange of semantic attributes. Based on such an extension, it would be possible to build pervasive systems where entities autonomously establish trust relationships, and only gradually reveal those attributes which are required for evaluating a specific access request, for example. Although this would increase communication and computational overhead, such mechanisms will especially be interesting in settings where privacy-critical information is involved in policy decisions, for example when social network profiles of users are used to determine access rights to resources.

Usage Control Usage control is an extension of access control which does not only determine under which conditions a resource may be accessed, but also how it has to be used after the access has been granted. In fact, it is one of the most comprehensive policy models and thus, we designed the framework in this thesis in a way that it will support the general usage control evaluation process. So, integrating a usage control module into the framework should be possible.

However, the actual challenges lie in the details of usage control models and their application to real systems. One of them is to map usage control policies to the events which can be observed in the system and which indicate the usage of a resource, such as accessing a network service or copying a file, for example. In order to track data across different locations (i.e., data that is copied to multiple files or memory locations), Pretschner et al. [136] proposed to map locations to states and to use state-based usage control policies. In [95], this approach is extended by a usage control model that refines high-level policies to implementation-level policies. This refinement is described in an implementation specific model and currently assumes that the system architecture remains static, certainly not a realistic assumption in many pervasive or service-based systems – a fact that the authors also recognize. Here, our situation-goal-based policies from section 7.3 could help. As we described above, the framework is able to detect changes in the system architecture and computes a sequence of actions to return to a secure state. In combination with a usage control module, this approach could be used to adapt the implementation-specific parts of the usage control module so that they match the reconfigured system.

Apart from that, describing resources, usage requirements, and monitors in a semantic model could help users to understand the policies and to detect flaws like a usage requirement for which no monitor exist and which hence cannot be detected.

Sticky policies Sticky policies [133] denote policies which "stick" to the protected data, even when it travels through the network and is shared across domain boundaries. This concept is especially important in usage control scenarios, where accesses to a certain datum shall be controlled even when it is outside the area of control of the policy domain – for example, when a confidential document is (legitimately) sent to a user who locally stores and views it on her computer. In the context of our policy framework, sticky policies are closely related to obligations – they instruct others how to handle data by declaring actions which must be taken or prevented. Consequently, we could handle sticky policies with our framework by providing two following two extensions in form of a module.

Firstly, the semantics of obligations would have to be extended in order to meet those of the sticky policy. Sticky policies in usage control scenarios often rely on some kind of temporal logic to denote which conditions apply before, during, and after usage of a specific datum. Examples are simple Linear Temporal Logic (LTL) [137], MFOTL [14], or CTL* [81]. A sticky policy module would have to provide an ontology fragment describing the structural semantics of obligations following such a logic.

Secondly, there must be a way to actually make policies "stick" to a resource, i.e. to attach them to a message when it is sent out by a service. We could achieve this with the PEP component, when hooking it into the handler chain of outgoing and incoming service calls. Then, the PEP can annotate outgoing requests with a set of obligations which represent the sticky policy. At the receiver's site, the PEP then would enforce the obligations, as necessary, and either remove the annotated obligations from the request or ensure that they remain attached to the transported data.

Automated classifications of entities and policies Authoring a security policy can be a cumbersome process and we thus intended to support users by the ontology-based representation of policies. Nevertheless, in an ideal world, a system would write its policies by itself, only guided by some very high-level guidelines stated by the user. The fact that we use Description Logic as a foundation for the policy model opens the possibility of applying methods from the field of machine learning (ML) in order to automatically classify concepts over which policies are stated.
As an example, tools like DL-Learner[1] by University of Leipzig are able to suggest class definitions for unknown individuals or find similar individuals. Because in pervasive systems, entities are not always known at the time of writing a policy, such classification features can be extremely helpful and could be used for automatic role mining in an RBAC model, for example, where based on some examples of role memberships, further individuals can be automatically assigned to the roles they fit in best.

Combination with Model-driven architecture design The integration of policy authoring into system modelling has not been in the focus of this thesis. It would however be interesting to research ways how developers could combine our Description Logic policy model with model-driven architecture (MDA) specifications. The combination seems natural, as both approaches have a similar goal: specifying a system and its security requirement at an abstract, platform-independent level. At this level, the model remains more or less constant and can easily be analysed and checked for conceptional flaws. It is then automatically translated into platform-specific models from which code and configuration files can be generated. While we already demonstrated a limited example of combining Situation-Goal policies with an UMLsec-like model, it seems to be promising to extend this towards a general modelling layer, at which users can describe the system functionality (e.g., components and workflows) along with their security requirements. One approach in this direction is TwoUse [130] which integrates ontologies into the Eclipse EMF modelling platform. Further investigations of these ideas could lead to the development of an integrated modelling IDE, which could help developers to ensure that all components in their system work together in a coherent way and obey a consistent overall security model.

8.3.2. Applications to other areas

Although we focussed in this thesis at managing the security mechanisms in a pervasive system, many of the concepts of our policy framework could be applied to other areas. Two major trends are currently the evolution of the web on to the "Semantic" Web 3.0, as well as the overall integration of embedded devices to so-called Cyber-Physical systems (CPS). In fact, even if the term "pervasive system" is not used in these contexts, the actual system architecture we considered here has a lot of similarities to the technologies of these trends and it is thus interesting to discuss how we could transfer the results of this thesis to them.

8.3.2.1. The Semantic Web

Web 2.0 – the web of social communities, mashups, and sharing of personal information — is currently turning into what is called Web 3.0, or the *Semantic Web*. The Semantic Web is

[1] `http://dl-learner.org/`

characterised by massive amounts of machine-readable semantic data and infrastructures of distributed service providers. It was already in 2007 when Tim Berners Lee discussed the vision of a *Giant Global Graph* [18] of social relationships. Nowadays, this vision has become true to a great extend, and more and more data sources become available: Facebook's *OpenGraph* API[2] opens information about social connections, activities, and products to developers of other web platforms via the OpenGraph protocol[3]. As part of the *Open Data* initiative, scientists and governments are releasing statistical and machine-readable data. Social movements like the *Quantified Self* use ubiquitous sensors and smartphones to monitor personal behaviour, and applications ideas like *frictionless sharing* create machine-readable data from daily activities of a user.

Furthermore, with the increased use of HTML5, the architecture of the web is getting closer to that of a pervasive system. Asynchronous JavaScript requests (AJAX) have been the first step to turn the traditional client/server based model into a network of distributed services which are orchestrated ("mashed up") for different applications. Nowadays, technology has significantly evolved and client-side storage [68], web sockets [70], inter-domain communication with cross-origin resource sharing (CORS) [180], and web messaging [69] have been added. As a result, "Web" does not only denote the delivery of a web page from a single server, but rather a network of distributed computational resources which collaborate to provide a value-added service to the user. Although the specific protocols might be different than the ones considered in this thesis, the architecture of the Web 3.0 is not much different from that of a pervasive system and consequently, the results of this thesis will also be relevant for controlling security mechanisms in a future Web 3.0.

In our framework, we represent domain knowledge and policies as ontologies. In a Semantic Web, where every user and object is semantically represented, the need for explicitly modelling the domain knowledge base will be removed and instead, policies can refer to the existing data sources of the Web. Consequently, policies can rely on much richer information and our approach of declaring access rights and security configurations on the basis of semantic information will be leveraged. For example, as relationships between users are represented as a Friend-of-a-Friend (FOAF) graph, new policy models become possible, which are currently not practical due to the vast amount of information which would have to be modelled. For instance, a policy model connecting context-awareness with social relationships between users would be straightforward in a Web 3.0 environment. Here, our framework would allow to quickly create new policy models and integrate semantic web knowledge about resources, users, their activities, and interactions. For example, a policy model which distinguishes between "close friends" of a user and other persons with which an automatic trust negotiation has to be conducted first could be realised using the framework.

Further, in a Web where services from different domains are combined with each other, mechanisms to negotiate SLAs, security requirements, and provisions are required. However, currently users have only few options to express their requirements – the HTTP "do not track" header [58] is one example. User centric policy languages like P3P have been proposed, but have experienced only limited adoption so far. As a result, work on P3P has been abandoned in 2006 due to the lack of interest on the part of browser manufacturers. However, in the Web 3.0, a session will not just comprise a single user and web server, but rather a mashup of multiple services which exchange information with each other. While authorisation protocols like OAuth [141] for distributed resource sharing are already in use, there are no policy languages available yet which let users and web servers specify arbitrary conditions for access to resources. Here, the results of this thesis could serve as a

[2]`https://developers.facebook.com/docs/opengraph/`

[3]`http://ogp.me/`

basis for a policy-controlled web, where resource owners and users can specify conditions and provisions for the usage of Web resources. Apart from the ontology-based (and thus, interoperable) policy modelling, our approach of merging conflicting requirements of different domains could be translated to a future Web scenario.

8.3.2.2. Cyber-Physical Systems

The ever increasing ubiquity of networked embedded devices is currently leading to a convergence of different application areas which have been isolated before. Areas like *eHealth*, *Smart Mobility*, or *Smart Grid*, are growing and become more and more interwoven, thereby creating the vision of a “system of systems”, i.e. the combination of different complex systems to a so-called *Cyber-Physical System* (CPS).

Cyber-Physical Systems denote complex infrastructures which connect objects from the physical world with computational resources and human interactions. They are characterised by a strong interconnection between the physical and the virtual world, the combination and interaction of systems from different application areas, and by changing domain and system boundaries. Moreover, CPS are context-aware, adaptive, and able to act at least partly autonomously. The dynamically changing collaboration between domains further requires for adaptations of controls and autonomous negotiations between systems.

CPS are not focussed at a single technology. They rather span all layers, ranging from hardware design, up to networking and middleware infrastructures, applications, and human interaction. Thus, contributions from a plethora of research areas is required in order to turn the vision of interoperable systems of systems into reality:

- work on machine learning and sensor fusion is required in order to make systems situation-aware.
- research on future multimodal user interaction and new concepts for HMIs are required.
- building upon existing results from the areas of policy-based management, multiagent systems, and adaptive systems it is necessary to conduct further research on self-protecting systems and mechanisms for autonomous decision making.
- such autonomous decision making must be based on comprehensive domain knowledge and security models. Research on modelling, representing, transforming, mapping, and application of such knowledge models for self* features must thus be intensified.

This thesis is closely related to some of the most important research questions in CPS. Figure 8.1 illustrates aspects of CPS and highlights the areas to which we made contributions in this thesis. In fact, pervasive systems make up the middleware part of a Cyber-Physical System. Thus, the results of this thesis pave the way for further research on CPS, especially on the following aspects.

- the ontology-based approach of modelling policies and domain knowledge in the form of ontologies is well suited for CPS, as it provides a common basis for information that needs to exchanged between domains.
- we made some contributions to the area of self-adaptive middleware, which is at the core of future Cyber-Physical Systems. Patterns like the proposed Situation-Goal Policies, in combination with the dynamically loadable obligations will become even more important in CPS, where manual management of configurations is hardly

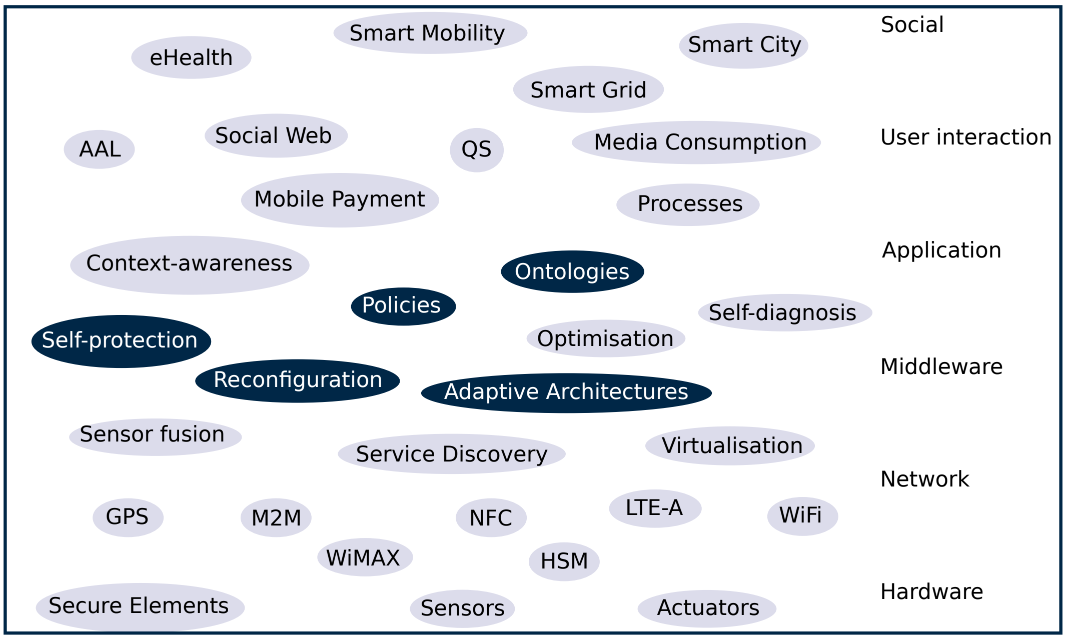


Figure 8.1.: Aspects of Cyber-Physical Systems and related areas of this thesis

possible and systems rather need to be able to autonomously adapt themselves to changing environments or requirements.

– cross-domain interaction and the handling of multilateral security requirements will be the rule, rather than the exception in CPS. The concept of a "system of systems" requires that different domains can be combined and are able to interact with each other. This raises challenges in terms of agreeing on a common security level. Here, this thesis has made contributions in form of a meta policy based merging algorithm for security policies from different, equitable domains, as well as in the form of a multilateral policy negotiation module, which autonomously achieves a pareto-optimal compromise between the security requirements of different domains.

Thus, both, the core policy framework itself, as well as concepts like autonomous policy negotiation, Situation-Goal policies, and cross-domain merging of decisions will be suitable starting points for future developments and research on the security of CPS.

Bibliography

[1] r^3 : Resourceful Reactive Rules. EU REWERSE IST-2004-50677. URL: `http://centria.di.fct.unl.pt/~rewerse/wg-i5/r3/` [cited November 15, 2012].

[2] REWERSE – Reasoning on the Web with Rules and Semantics. EU-FP6 IST 506779, 2004-2008. URL: `http://rewerse.net/` [cited November 15, 2012].

[3] AMIGO. Ambient intelligence for the networked home environment. http://www.amigo-project.org/, 2007. URL: `http://www.amigo-project.org/` [cited November 15, 2012].

[4] Ponder 2. http://www.ponder2.net/, December 2008. URL: `http://www.ponder2.net/` [cited November 15, 2012].

[5] wider3 – WIDER Resourceful Reactive Rules, August 2009. URL: `http://code.google.com/p/wider3/wiki/WIDER` [cited November 15, 2012].

[6] OSAmI – Open Source Ambient Intelligence Commons for an Open and Sustainable Internet. Project funded by the Spanish Ministry of Industry (AVANZA I+D programme) and FEDER, 2010. TSI-020400-2009-92.

[7] Hydra. Middleware for networked devices. Project funded by the EU-FP7 programme, May 2012. IST 2005-034891. URL: `http://www.hydramiddleware.eu` [cited November 15, 2012].

[8] Projekt simTD – Sichere Intelligente Mobilität. Testfeld Deutschland, January 2012. URL: `http://www.simtd.org` [cited November 15, 2012].

[9] José Júlio Alferes and Ricardo Amador. r^3: A Foundational Ontology for Reactive Rules. In *Proc. the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 933–952, Berlin, Heidelberg, 2007. Springer.

[10] Ricardo Amador and José Júlio Alferes. Knowledge Resources Towards a RESTful Knowledge. KR2RK Manuscript submitted for journal publication, June 2009.

[11] Marco Avitabile. An examination of requirements for meta-policies in policy-based management. Diploma thesis, Technical University of Munich, 1998. URL: `http://www.mnm-team.org/pub/Diplomarbeiten/avit98/`.

[12] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. ISBN 978-0-521-87625-4.

[13] Franz Baader, Ian Horrocks, and Ulrike Sattle. *Handbook of Knowledge Representation*, chapter 3 Description Logics, pages 135–180. Elsevier, 2007. ISBN 0444522115.

[14] D. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *Computer Aided Verification*, pages 1–18. Springer, 2010.

[15] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proc. 5th IEEE Int'l Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 159–168. IEEE Computer Society, 2004.

[16] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, The MITRE Coporation, Bedford, MA, USA, March 1976.

[17] Andras Belokosztolszki and Ken Moody. Meta-policies for distributed role-based access control systems. In *Third International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 106–115, 2002.

[18] Tim Berners-Lee. Giant global graph. http://dig.csail.mit.edu/breadcrumbs/node/215, November 2007.

[19] Tim Berners-Lee, Dan Connolly, Eric Prud'hommeaux, and Yosi Scharf. Experience with n3 rules. In *Rule Languages for Interoperability*, April 2005.

[20] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web: a new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 284(5):34–43, May 2001.

[21] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. Trust negotiations: Concepts, systems and languages. *IEEE Computing in Science & Engineering*, 4:27–34, 2004.

[22] Xander Bighorn and Robert Mines. JSR 8: Open Services Gateway Specification. Java Specification Request, May 1999. (withdrawn).

[23] Chris Bizer and Andreas Schultz. The Berlin SPARQL Benchmark V3 Results, February 2011. URL: `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/index.html`.

[24] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[25] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL Reasoners. In *Workshop on Advancing Reasoning on the Web (ARea2008): Scalability and Commonsense*, June 2008.

[26] J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) Protocol. RFC 2748, January 2000. URL: `http://tools.ietf.org/html/rfc2748`.

[27] David F.C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[28] Jeremy Bryans. Reasoning about XACML policies using CSP. Technical Report CS-TR-924, School of Computing Science, University of Newcastle upon Tyne, July 2005.

[29] L. Capra, W. Emmerich, and Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, October 2003. `doi:10.1109/TSE.2003.1237173`.

[30] K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, and A. Smith. COPS Usage for Policy Provisioning (COPS-PR). RFC 3084, March 2001. URL: `http://tools.ietf.org/rfc/rfc3084.txt`.

[31] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. Internet Engineering Task Force, Internet-Draft, February 2011. URL: `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt`.

[32] S. Cheshire and M. Krochmal. Multicast DNS. Internet Engineering Task Force, Internet-Draft, February 2011. URL: `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`.

[33] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Version 3.0.2. Technical report, OASIS UDDI Spec TC, October 2004. URL: `http://uddi.org/pubs/uddi_v3.htm`.

[34] Juri L. De Coi and Daniel Olmedilla. A flexible policy-driven trust negotiation model. In *IEEE/WIC/ACM International Conf. Intelligent Agent Technology*, pages 450–453, Silicon Valley, CA, USA, 2007. IEEE Computer Society Press.

[35] P Costa, G Coulson, C Mascolo, G P Picco, and S Zachariadis. The runes middleware: a reconfigurable component-based approach to networked embedded systems. In *IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, pages 806–810. IEEE Computer Society Press, 2005. URL: `http://eprints.ucl.ac.uk/5656/`.

[36] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe, and Stefanos Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *IEEE International Conf. Pervasive Computing and Communications (PERCOM)*, 2007.

[37] N. Dai, W. Thronicke, A.R. Lopez, F.C. Latasa, E. Zeeb, C. Fiehe, A. Litvina, J. Krueger, O. Dohndorf, I. Agudo, and J. Bermejo. OSAMI commons – an open dynamic services platform for ambient intelligence. In *16th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–10, September 2011. `doi:10.1109/ETFA.2011.6059235`.

[38] E. Damiani, S. De Capitani di Vimercati, C. Fugazza, and P. Samarati. Extending policy languages to the semantic web. In *Proc. the International Conf. Web Engineering*, pages 330–343, 2004.

[39] Georg-August-Universität Göttingen DBIS Group, Institute for Informatics. Mars: Modular active rules for the semantic web. http://www.dbis.informatik.uni-goettingen.de/MARS/, 2005. URL: `http://www.dbis.informatik.uni-goettingen.de/MARS/`.

[40] Anand Dersingh, Ramiro Liscano, and Allan Jost. Context-aware access control using semantic policies. *Ubiquitous Computing and Communcation Journal*, Special issue on Autonomic Computing Systems and Applications:178–192, June 2008.

[41] P Dewan and S Joshi. Auction based distributed scheduling in a dynamic job shop environment. *International Journal of Production Research*, 40(5):1173–1191, 2002.

[42] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001. URL: `http://www.springerlink.com/content/kvd3xmw5q1jucder`, `doi:10.1007/PL00000013`.

[43] Anind K. Dey and Gregory Abowd. Towards a better understanding of context and context-awareness. In *Proc. Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the Conf. Human Factors in Computing Systems (CHI)*, 2000.

[44] Distributed Management Task Force (DTMF). Common Information Model (CIM) Specification v2.3, October 2005. DSP0004.

[45] Distributed Management Task Force (DMTF). CIM Simplified Policy Language (CIM-SPL). DMTF Standard DSP0231, July 2009.

[46] Nick Drummond, Alan Rector, Robert Stevens, Georgina Moulton, Matthew Horridge, Hai H. Wang, and Julian Seidenberg. Putting OWL in Order: Patterns for Sequences in OWL. In *OWLed*, 2006.

[47] Distributed Management Task Force (DTMF). CIM Policy Model White Paper Version 2.7. Technical report, 2003.

[48] Juergen Dunkel. On complex event processing for sensor networks. *Int'l Symposium on Autonomous Decentralized Systems*, 39:1–6, 2009. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5207376`.

[49] Claudia Eckert. *IT-Sicherheit*. Oldenbourg Wissenschaftsverlag, 5th edition, September 2008. ISBN 978-3-486-58270-3.

[50] Andreas Ekelhart, Stefan Fenz, Markus Klemen, and Edgar Weippl. Security ontology: Simulating threats to corporate assets. In Aditya Bagchi and Vijayalakshmi Atluri, editors, *Information Systems Security*, volume 4332 of *Lecture Notes in Computer Science*, pages 249–259. Springer Berlin / Heidelberg, 2006. URL: `http://dx.doi.org/10.1007/11961635_17`.

[51] M. R. Endsley. Situation awareness global assessment technique (sagat). In *Proc. the National Aerospace and Electronics Conference*, pages 789–795, New York, 1988. IEEE.

[52] M.R Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64, 1995.

[53] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. `doi:10.1145/857076.857078`.

[54] Joel Farrell and Holger Lausen (Editors). Semantic Annotations for WSDL and XML Schema. W3C Recommendation, World Wide Web Consortium (W3C), August 2007.

[55] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control, 2001.

[56] D.F. Ferraiolo and D.R. Kuhn. Role based access control. In *Proc. the 15th National Computer Security Conference*, pages 554–563, Baltimore MD, USA, 1992.

[57] Rodolfo Ferrini and Elisa Bertino. Supporting rbac with xacml+owl. In *Proc. the 14th ACM symposium on Access control models and technologies (SACMAT '09)*, pages 145–154, New York, NY, USA, 2009. ACM. doi:http://doi.acm.org/10.1145/1542207.1542231.

[58] Roy T. Fielding and David Singer. Tracking preference expression (DNT). W3C Working Draft, World Wide Web Consortium (W3C, March 2012. URL: http://www.w3.org/TR/tracking-dnt/.

[59] Tim Finin, Anupam Joshi, Lalana Kagal, Jianwei Niu, Ravi Sandhu, William H Winsborough, and Bhavani Thuraisingham. ROWLBAC - Representing Role Based Access Control in OWL. In *Proc. the 13th Symposium on Access control Models and Technologies*. ACM Press, June 2008.

[60] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Margrave, An API for XACML Policy Verification and Change Analysis, February 2009. URL: http://www.cs.brown.edu/research/plt/software/margrave/.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 21st printing edition, 2000. ISBN 0201633612.

[62] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl - the planning domain definition language. AIPS-98 Planning Competition Committee, 1998.

[63] Guido Governatori. Defeasible description logics. In Grigoris Antoniou and Harold Boley, editors, *Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Lecture Notes in Computer Science*, pages 98–112. Springer Berlin / Heidelberg, 2004.

[64] Peter Haase, Pascal Hitzler, Jürgen Angele, M. Krötzsch, Markus Krötzsch, and Rudi Studer. Practical reasoning with owl and dl-safe rules. PROWL-2006 tutorial at European Semantic Web Conference (ESWC), June 2006.

[65] Klaus Marius Hansen, Weishan Zhang, and João Fernandes. Flexible generation of pervasive web services using osgi declarative service and owl ontologies. In *Proc. the 15th Asia-Pacific Software Engineering Conference (APSEC'08)*, Bejing, China., December 2008.

[66] Almut Herzog, Nahid Shahmehri, and Claudiu Duma. An ontology of information security. *International Journal of Information Security and Privacy*, 1(4):1–23, 2007.

[67] Stephan Heuser. Entwicklung eines Frameworks zur sicheren mehrseitigen Aushandlung von Policies in Ambient-Intelligence Umgebungen. Diploma thesis, Technische Universität Darmstadt, Fachbereich Informatik, Fachgebiet Sicherheit in der Informationstechnik, Darmstadt, Germany, April 2010.

[68] Ian Hickson. Web storage. W3C Candidate Recommendation, World Wide Web Consortium (W3C), December 2011. URL: http://www.w3.org/TR/webstorage/.

[69] Ian Hickson. HTML5 web messaging. W3C Candidate Recommendation, World Wide Web Consortium (W3C), May 2012. URL: http://www.w3.org/TR/webmessaging/#web-messaging.

[70] Ian Hickson. The web sockets API. W3C working draft, World Wide Web Consortium (W3C), May 2012. URL: `http://www.w3.org/TR/websockets/`.

[71] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In Joachim Biskup and Javier López, editors, *Computer Security - ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer Berlin / Heidelberg, 2007.

[72] Mario Hoffmann, Julian Schütte, Tobias Wahl, Adedayo Adetoye, Atta Badii, Sebastian Zickau, Michael Crouch, and Hasan Akram. Deliverable d7.8 – security and privacy components for ddk prototype. Deliverable of the HYDRA project (IST 2005-034891), June 2009.

[73] Matthew Horridge and Peter F. Patel-Schneider. Owl 2 web ontology language: Manchester syntax. W3C Working Group Note, World Wide Web Consortium (W3C), October 2009. URL: `http://www.w3.org/TR/owl2-manchester-syntax/`.

[74] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *In Proceedings of the 10th International Conf. Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, 2006.

[75] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for shoiq. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 448–453, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1642293.1642365`.

[76] Hilary H. Hosmer. Integrating security policies. In *Proceeding of the Third RADC Database Security Workshop*, pages 5–7, Castile, N.Y., June 1990.

[77] Hilary H. Hosmer. Metapolicies I. *ACM SIGSAC Review – Special issue on Issues '91: data management security and privacy standards*, 10(2-3):18–43, 1992. Special issue on Issues '91: data management security and privacy standards. `doi:http://doi.acm.org/10.1145/147092.147097`.

[78] Hilary H. Hosmer. Metapolicies II. In *Proc. the 15th National Computer Security Conference*, pages 369–378. United States Government Printing Office: 1992-625-512:60546, October 1992.

[79] Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. Looci: a loosely-coupled component infrastructure for networked embedded systems. In *International Conf. Advances in Mobile Computing & Multimedia (MoMM)*, pages 195–203, Kuala Lumpur, Malaysia, 2009. ACM New York.

[80] Polar Humenn. The formal semantics of xacml. http://lists.oasis-open.org/archives/xacml/200310/msg00094.html, October 2003.

[81] S. Jiang and R. Kumar. Supervisory control of discrete event systems with ctl* temporal logic specifications. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 5, pages 4122–4127. IEEE Computer Society Press, 2001.

[82] Jan Jürjens. *Secure Systems Development with UML*. Springer Berlin / Heidelberg, 2005. ISBN 3540007016.

[83] L. Kagal, T. Finin, and Anupam Joshi. A policy language for pervasive computing environments. In *Proc. the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*, POLICY '03, 2003.

[84] Lalana Kagal. Rei Ontology Specifications, v2.0. http://www.cs.umbc.edu/ lkagal1/rei/. November 15, 2012. URL: `http://www.cs.umbc.edu/~lkagal1/rei/`.

[85] Lalana Kagal. Rei: A Policy Language for the Me-Centric Project. Technical Report HPL-2002-270, Enterprise Systems Data Management Lab., HP Lab. Palo Alto, September 2002.

[86] Lalana Kagal, Tim Berners-Lee, Dan Connolly, and Daniel Weitzner. Promoting interoperability between heterogeneous policy domains. W3C Workshop on Languages for Privacy Policy Negotiation and Semantics-Driven Enforcement, October 2006.

[87] Lalana Kagal, Tim Berners-Lee, Dan Connolly, and Daniel Weitzner. Using semantic web technologies for policy management on the web. In *Proc. 21st National Conf. Artificial Intelligence (AAAI)*, July 2006. URL: `http://dig.csail.mit.edu/2006/Papers/AAAI/rein@aaai.pdf`.

[88] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962. URL: `http://portal.acm.org/citation.cfm?doid=368996.369025`.

[89] Basel Katt, Xinwen Zhang, Ruth Breu, Michael Hafner, and Jean-Pierre Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *Proc. the 13th ACM symposium on Access control models and technologies*, SACMAT '08, pages 123–132, New York, NY, USA, 2008. ACM. `doi:http://doi.acm.org/10.1145/1377836.1377856`.

[90] Vladimir Kolovski. Formal semantics of xacml v3.0. XACML mailing list, March 2008.

[91] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proc. the 16th International Conf. World Wide Web (WWW)*, pages 677–686, New York, NY, USA, 2007. ACM. `doi:http://doi.acm.org/10.1145/1242572.1242664`.

[92] Vladimir Kolovski, James Hendler, and Bijan Parsia. Formalizing XACML Using Defeasible Description Logics. In *Proc. 16th Int'l Conf. World Wide Web (WWW)*, pages 677–686, New York, USA, May 2007. ACM. `doi:http://doi.acm.org/10.1145/1242572.1242664`.

[93] Vladimir Kolovski, Bijan Parsia, Yarden Katz, and James Hendler. Representing Web Service Policies in OWL-DL. In *4th International Semantic Web Conference (ISWC)*, 2005.

[94] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proceeding of Conf. Future of Software Engineering (FOSE '07)*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society. `doi:http://dx.doi.org/10.1109/FOSE.2007.19`.

[95] Prachi Kumari and Alexander Pretschner. Deriving implementation-level policies for usage control enforcement. In *Proceedings of the second ACM conference on Data*

and Application Security and Privacy, CODASPY '12, pages 83–94, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2133601.2133612`, `doi:10.1145/2133601.2133612`.

[96] Winfried E. Kühnhauser and Michael von Kopp Ostrowski. A framework to support multiple security policies. In *Proceedings of the 7th Canadian Computer Security Symposium*, 1995.

[97] Lalana Kagal. The Rein Policy Framework for the Semantic Web, October 2006. http://dig.csail.mit.edu/2006/06/rein/. URL: `http://dig.csail.mit.edu/2006/06/rein/` [cited November 15, 2012].

[98] Steffen Lamparter and Sudhir Agarwal. Specification of policies for automatic negotiations of web services. In Lalana Kagal, Tim Finin, and James Hendler, editors, *Proc. the Semantic Web and Policy Workshop, held in conjunction with the 4th International Semantic Web Conference*, pages 99–109, Galway, Ireland, November 2005.

[99] Steffen Lamparter, Anupriya Ankolekar, Daniel Oberle, Rudi Studer, and Christof Weinhardt. Semantic specification and evaluation of bids in web-based markets. *Electronic Commerce Research and Applications*, 7(3):313–329, Autumn 2008.

[100] Adam Lee, Jodie P. Boyer, Lars E. Olson, and Carl A. Gunter. Defeasible security policy composition for web services. In *Proc. 4th ACM Workshop on Formal Methods in Security (FMSE)*, FMSE '06, pages 45–54, New York, NY, USA, 2006. ACM.

[101] Choonhwa Lee, Seungjae Lee, Eunsam Kim, and Wonjun Lee. Cross-domain service composition in osgi environments. In *IEICE TRANSACTIONS on Information and Systems*, volume E92-D, pages 1316–1319, June 2009. `doi:10.1587/transinf.E92.D.1316`.

[102] Dave Locke. MQ Telemetry Transport (MQTT) V3.1 Protocol Specification. IBM developerWorks, August 2010.

[103] Seng Loke. *Context-Aware Pervasive Systems – Architectures for a new Breed of Applications*. Auerbach Publications, 2006. ISBN 978-0849372551.

[104] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. Using CIM to realize policy validation within the ponder framework. In *Proc. the IFIP/IEEE Network Operations and Management Symposium (NOMS 2004)*, November 2004.

[105] MASSIF. Management of security information and events in service infrastructures. EU FP7-ICT-2009-5 Project, May 2012. URL: `http://www.massif-project.eu/`.

[106] Deborah L. McGuinness and Frank van Harmelen (Editors). OWL Web Ontology Language. OASIS, February 2004. URL: `http://www.w3.org/TR/owl-features/`.

[107] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology, September 2001.

[108] Deepak Mishra. SNOOP: An Event Specification Language for Active Databases. Master thesis, University of Florida, 1991.

[109] B. Moore. Policy Core Information Model (PCIM) Extensions. RFC 3460 (Proposed Standard), January 2003. URL: `http://www.ietf.org/rfc/rfc3460.txt`.

[110] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. RFC 3060 (Proposed Standard), February 2001. Updated by RFC 3460. URL: `http://www.ietf.org/rfc/rfc3060.txt`.

[111] Marie-Luise Moschgath. *Kontextabhängige Zugriffskontrolle für Anwendungen im Ubiquitous Computing*. PhD thesis, Technical University Darmstadt, 2002.

[112] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau (Editors). OWL 2 Web Ontology Language Direct Semantics. W3C Recommendation, World Wide Web Consortium (W3C), 2009. URL: `http://www.w3.org/TR/owl2-direct-semantics/`.

[113] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist (Editors). WS-SecurityPolicy 1.2. OASIS Standard, July 2007. URL: `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html`.

[114] Naval Research Lab. NRL Security Ontology, 2007. URL: `http://chacs.nrl.navy.mil/projects/4SEA/ontology.html`.

[115] Wolfgang Nejdl, Daniel Olmedilla, and Marianne Winslett. Peertrust: Automated trust negotiation for peers on the semantic web. In *Workshop on Secure Data Management in a Connected World (SDM'04)*, pages 118–132, 2004.

[116] Wolfgang Nejdl, Daniel Olmedilla, Marianne Winslett, and Charles C. Zhang. Ontology-based policy specification and management. In *2nd European Semantic Web Conference (ESWC)*, pages 290–302. Springer, 2005.

[117] Natasha Noy and Alan Rector (Editors). Defining n-ary relations on the semantic web. W3C Working Group Note, World Wide Web Consortium (W3C), April 2006. URL: `http://www.w3.org/TR/swbp-n-aryRelations/`.

[118] Donald Nute. Defeasible logic. In Oskar Bartenstein, Ulrich Geske, Markus Hannebauer, and Osamu Yoshie, editors, *Web Knowledge Management and Decision Support*, volume 2543 of *Lecture Notes in Computer Science*, pages 151–169. Springer Berlin / Heidelberg, 2003.

[119] OASIS. Core and hierarchical role based access control (RBAC) profile of XACML v2.0. OASIS, February 2005.

[120] OASIS. eXtensible Access Control Markup Language (XACML), Version 2.0. OASIS Standard, February 2005.

[121] OASIS. Privacy policy profile of XACML v2.0. OASIS Standard, February 2005.

[122] OASIS. SAML 2.0 profile of XACML v2.0. OASIS Standard, February 2005.

[123] OASIS. eXtensible Access Control Markup Language (XACML), Version 3.0, Draft Are. OASIS, May 2009.

[124] OASIS. Web services coordination v1.2. OASIS Standard, February 2009. URL: `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec.html`.

[125] OASIS. Web Services Reliable Messaging v1.2. OASIS Standard, February 2009.

[126] Martin J. Osborne. *An Introduction to Game Theory*. Oxford University Press, 2004. ISBN 0195128958.

[127] OSGi Alliance. Listeners considered harmful: The "whiteboard" pattern. Technical whitepaper, August 2004.

[128] OSGi Alliance. OSGi Service Platform – Service Compendium, Release 4, Version 4.3, January 2012.

[129] Jaehong Park and Ravi Sandhu. The ucon$_{ABC}$ usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, February 2004. URL: `http://doi.acm.org/10.1145/984334.984339`, `doi:10.1145/984334.984339`.

[130] Fernando S. Parreiras. *Semantic Web and Model-Driven Engineering*. John Wiley & Sons, 2012. ISBN 1118004175.

[131] B. Parsia, E. Sirin, B. C. Grau, E. Ruckhaus, and D. Hewlett. Cautiously approaching SWRL. Technical report, University of Maryland - College Park, December 2004.

[132] Bijan Parsia, Vladimir Kolovski, and James Hendler. Expressing WS-Policies in OWL. In *WWW2005 Workshop on Policy Management for the Web*, May 2005. URL: `http://cs.umbc.edu/pm4w/papers/parsia18.pdf`.

[133] Siani Pearson and Marco Casassa Mont. Sticky policies: An approach for managing privacy across multiple parties. *IEEE Computer*, 44(9):60–67, September 2011.

[134] Alexander Pretschner. An overview of distributed usage control. In *Proceedings of the 2nd Conference on Knowledge Engineering: Principles and Techniques*, pages 17–25, Cluj, 2009.

[135] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Communications of the ACM*, 49(9):39–44, 2006. `doi:http://doi.acm.org/10.1145/1151030.1151053`.

[136] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. Representation-independent data usage control. In *Proc. 6th Intl. Workshop on Data Privacy Management*, 2011.

[137] Alexander Pretschner, Judith Rüesch, Christian Schaefer, and Thomas Walter. Formal analyses of usage control policies. In *Proceedings of the 4th International Conerence on Availability, Reliability, and Security (AReS)*, Fukuoka, March 2009.

[138] T. Priebe, W. Dobmeier, and Kamprath N. Supporting attribute-based access control in authorization and authentication infrastructures with ontologies. In *Proceedings of the 1st International Conf. Availability, Reliability and Security (AReS)*, pages 465–472, Vienna, Austria, April 2006.

[139] Marek Psiuk, Daniel Żmuda, and Krzysztof Zieliński. Distributed OSGi built over message-oriented middleware. *Software: Practice and Experience*, 2011. URL: `http://dx.doi.org/10.1002/spe.1148`, `doi:10.1002/spe.1148`.

[140] E. Rasmusen. *Games and information: An introduction to game theory*. Blackwell Publishing, 4th edition edition, 2006. ISBN 978-1405136662.

[141] D. Recordon and D. Hardt. The OAuth 2.0 Authorization Framework. Internet-Draft, IETF, June 2012. URL: `http://tools.ietf.org/html/draft-ietf-oauth-v2-27`.

[142] REWERSE Working Group I2. Protune: PROvisional TrUst NEgotiation. http://policy.l3s.uni-hannover.de:9080/policyFramework/protune/, May 2008. URL: `http://policy.l3s.uni-hannover.de:9080/policyFramework/protune/`.

[143] Roland Rieke, Julian Schütte, and Andrew Hutchison. Architecting an security strategy measurement and management system. In *Proceedings of the Model Driven Security Workshop (MDSec) at the ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems (MODELS)*, 2012.

[144] Taufiq Rochaeli and Claudia Eckert. Using patterns paradigm to refine workflow policies. *1st International Workshop on Secure systems methodologies using patterns (SPattern'07)*, 2007.

[145] Taufiq Rochaeli and Ruben Wolf. Policy Generator. SicAri Consortium, 2006.

[146] Dumitru Roman, Holger Lausen, and Uwe Keller (Editors). Web Service Modeling Ontology (WSMO). Technical report, Conceptual Models for Web Service (CMS) Working Group, 2005. URL: `http://cms-wg.sti2.org/doc/D1v1.0%20Web%20Service%20Modeling%20Ontology%20(WSMO).html`.

[147] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74 – 83, 2002. `doi:10.1109/MPRV.2002.1158281`.

[148] G. Russello, Changyu Dong, and N. Dulay. Authorisation and conflict resolution for hierarchical domains. In *Proc. the 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 201–210, 13–15 June 2007. `doi:10.1109/POLICY.2007.8`.

[149] Julian Schütte, Hervais Simo Fhom, and Mark Gall. Security policies in dynamic service compositions. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, Rome, Italy, July 2012. SciTePress.

[150] Julian Schütte and Stephan Heuser. Auctions for secure multi-party policy negotiation in ambient intelligence. In *Proceedings of Workshops of the International Conference on Advanced Information Networking and Applications (WAINA)*. IEEE Computer Society Press, March 2011.

[151] Julian Schütte, Nicolai Kuntze, Andreas Fuchs, and Atta Badii. Authentic refinement of semantically enhanced policies in pervasive systems. In Kai Rannenberg, Vijay Varadharajan, and Christian Weber, editors, *IFIP Advances in Information and Communication Technology, Proc. 25th International Information Security Conference (IFIP SEC)*, volume 330, pages 90–102. Springer Boston, 2010.

[152] Julian Schütte, Roland Rieke, and Timo Winkelvos. Model-based security event management. In *Proceedings of the International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS)*, volume 7531 of *Lecture Notes in Computer Science*. Springer, October 2012.

[153] Julian Schütte. Apollon: Towards a semantically extensible policy framework. In *Proc. the Int'l Conf. Security and Cryptography (SECRYPT)*, pages 391–395, Seville, Spain, July 2011. SciTePress.

[154] Julian Schütte. Goal-based policies for self-protecting systems. In *Proc. the 26th IEEE International Conf. Advanced Information Networking and Applications (AINA)*, Fukuoka, Japan, March 2012. IEEE Computer Society Press.

[155] Julian Schütte and Tobias Wahl. Description logics-based handling of inter-domain policy conflicts. *IEEE Vehicular Technology Magazine*, 5(3), 2010.

[156] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobson, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 68–79. IEEE Computer Society, 2002.

[157] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979. URL: `http://doi.acm.org/10.1145/359168.359176`, `doi:http://doi.acm.org/10.1145/359168.359176`.

[158] Chetan Shankar and Roy Campbell. Ordering management actions in pervasive systems using specification-enhanced policies. In *IEEE International Conf. Pervasive Computing and Communications*, pages 234–238, Los Alamitos, CA, USA, 2006. IEEE Computer Society. `doi:http://doi.ieeecomputersociety.org/10.1109/PERCOM.2006.41`.

[159] Chetan Shankar, Vanish Talwar, Subu Iyer, Yuan Chen, Dejan Milojicić, and Roy Campbell. Specification-enhanced policies for automated management of changes in it systems. In *Proc. the 20th conference on Large Installation System Administration (LISA)*, pages 103–118, Berkeley, CA, USA, 2006. USENIX Association.

[160] SicAri. Policy patterns and its applications, 2004. Work package Pol 3.

[161] SicAri. Requirements on SicAri Security Policies, Mai 2004. Work package Pol 1, 25. Mai 2004.

[162] simTD. Deliverable D21.2 – Konsolidierter Systemarchitekturentwurf, October 2009.

[163] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.

[164] Sebastian Speiser. Semantic annotations for WS-Policy. In *IEEE International Conf. Web Services (ICWS)*, pages 449–456, 2010.

[165] Andrew Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms*. Pearson Prentice Hall International, second edition, 2007.

[166] Kerry Taylor and Lucas Leidinger. Ontology-driven complex event processing in heterogeneous sensor networks. In Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and JeffEditors Pan, editors, *The Semanic Web: Research and Applications*, volume 6644 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2011.

[167] Mohammad-Reza Tazari, Francesco Furfari, Álvaro Fides Valero, Sten Hanke, Oliver Höftberger, Dionisis Kehagias, Miran Mosmondor, Reiner Wichert, and Peter Wolf. *Handbook of Ambient Assisted Living; Technology for Healthcare, Rehabilitation and Well-being*, chapter The universAAL Reference Model for AAL, pages 612–625. IOS Press, 2012. ISBN: 978-1-60750-836-6. `doi:doi:10.3233/978-1-60750-837-3-?`

[168] Kia Teymourian and Adrian Paschke. Towards semantic event processing. In *Proc. the Third ACM International Conf. Distributed Event-Based Systems (DEBS)*, DEBS '09, pages 29:1–29:2, New York, NY, USA, 2009. ACM.

[169] Stephan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. Phd thesis, Technische Hochschule Aachen, 2001 2001.

[170] Alessandra Toninelli, Jeffrey M. Bradshaw, Lalana Kagal, and Rebecca Montanari. Rule-based and ontology-based policies: Toward a hybrid approach to control agents in pervasive environments. In *Proc. the Semantic Web and Policy Workshop*, November 2005.

[171] Alessandra Toninelli, Antonio Corradi, and Rebecca Montanari. A quality of context-aware approach to access control in pervasive environments. In *MobileWireless Middleware, Operating Systems, and Applications*, volume 7, pages 236–251, Berlin, Germany, April 2009. `doi:10.1007/978-3-642-01802-2_18`.

[172] Alessandra Toninelli, Rebecca Montanari, Lalana Kagal, and Ora Lassila. A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *International Semantic Web Conference (ISWC)*, volume 4273, pages 473–486. Springer Berlin / Heidelberg, 2006. URL: `http://www.springerlink.com/index/1041673323243651.pdf`.

[173] Alessandra Toninelli, Rebecca Montanari, Lalana Kagal, and Ora Lassila. Proteus: A semantic context-aware adaptive policy model. In *Proc. the IEEE 2007 International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2007.

[174] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In *The Semantic Web (ISWC 2003)*, volume 2870/2003 of *Lecture Notes in Computer Science*, pages 419–437. Springer Berlin / Heidelberg, 2003. `doi:10.1007/b14287`.

[175] Dmitry Tsarkov and Ian Horrocks. Dl reasoner vs. first-order prover. In *Proc. the 2003 Description Logic Workshop (DL 2003)*, volume 81, pages 152–159. CEUR (http://ceur-ws.org/), 2003.

[176] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A policy system for autonomous pervasive environments. In *The Fifth International Conf. Autonomic and Autonomous Systems (ICAS)*, pages 330–335. IEEE Computer Society Press, April 2009. URL: `http://www.ponder2.net/cgi-bin/moin.cgi/Ponder2Publications`.

[177] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Inc., New York, NY, USA, 1988.

[178] Andrzej Uszok and Jeff Bradshaw. Kaos policies for web services. W3C Workshop on Constraints and Capabilities for Web Services, October 2004.

[179] Andrzej Uszok, Jeffrey M. Bradshaw, Renia Jeffers, Niranjan Suri, Patrick J. Hayes, Maggie R. Breedy, Larry Bunch, Matt Johnson, Shriniwas Kulkarni, and James Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Third International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 93–96, 2003.

[180] Anne van Kesteren. Cross-origin resource sharing. W3C Working Draft, World Wide Web Consortium (W3C), April 2012. URL: `http://www.w3.org/TR/cors/`.

[181] Asir S Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçinalp. Web Services Policy 1.5 - Framework (WS-Policy). W3C Recommendation, World Wide Web Consortium (W3C), September 2007. URL: `http://www.w3.org/Submission/WS-Policy/`.

[182] Kris Verlaenen, Bart De Win, and Wouter Joosen. Policy analysis using a hybrid semantic reasoning engine. In *International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 193–200, Washington, DC, USA, 2007. IEEE Computer Society. `doi:http://dx.doi.org/10.1109/POLICY.2007.33`.

[183] Kunal Verma, Rama Akkiraju, and Richard Goodwin. Semantic matching of web service policies. *Proc. the Second Workshop on Semantic and Dynamic Web Processes (SDWP)*, pages 79–90, 2005.

[184] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. In *The Journal of Finance*, volume 16, pages 8–37, 1961.

[185] Tomas Vitvar and Jacek Kopecky (Editors). hRESTS & MicroWSMO. Technical report, SOA4All D3.4.6. UIBK 2009, March 2009. URL: `http://cms-wg.sti2.org/doc/D12v0.1%20hRESTS%20&%20MicroWSMO.html`.

[186] Markus Völter, Michael Kirchner, and Uwe Zdun. *Remoting Patterns – Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley & Sons, 2004. ISBN 978-0-470-85662-8.

[187] W3C Member Submission. Owl-s: Semantic markup for web services. http://www.w3.org/Submission/OWL-S/, November 2004.

[188] W3C Member Submission. Web Services Policy 1.2 - Framework (WS-Policy). http://www.w3.org/Submission/WS-Policy/, April 2006. URL: `http://www.w3.org/Submission/WS-Policy/`.

[189] W3C P3P Working Group. Platform for privacy preferences (p3p) project, November 2007.

[190] W3C Semantic Sensor Network Incubator Group. Semantic Sensor Network XG Final Report. http://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628/, June 2011.

[191] F. Wang and K. J. Turner. Policy conflicts in home care systems. In *Proc. 9th International Conference on Feature Interactions in Software and Communication Systems*, 2008.

[192] Kewen Wang, David Billington, Jeff Blee, and Grigoris Antoniou. Combining description logic and defeasible logic for the semantic web. In Grigoris Antoniou and Harold Boley, editors, *Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Lecture Notes in Computer Science*, pages 170–181. Springer Berlin / Heidelberg, 2004.

[193] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–10, 1991.

[194] Mac Gregor William, Dutcher William, and Khan Jamil. An ontology of identity credentials. Information Technology LaboratoryNational Institute of Standards and Technology, October 2006.

[195] W.H. Winsborough, K.E. Seamons, and V.E. Jones. Automated trust negotiation. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX)*, volume 1, pages 88–102, Hilton Head, SC, USA, January 2000. IEEE Computer Society Press.

[196] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. RFC 2753, IETF, January 2000. URL: `http://www.ietf.org/rfc/rfc2753.txt`.

[197] Weishan Zhang, K.M. Hansen, and J. Fernandes. Towards open world software architectures with semantic architectural styles, components and connectors. In *IEEE International Conf. Engineering of Complex Computer Systems (ICECCS)*, pages 40 –49, june 2009. `doi:10.1109/ICECCS.2009.7`.

[198] Weishan Zhang, Julian Schütte, Mads Ingstrup, and Klaus M. Hansen. A Genetic Algorithms-based approach for Optimized Self-protection in a Pervasive Service Middleware. In *Proc. 7th Int'l Joint Conf. Service Oriented Computing (ICSoC)*, pages 404–419. Springer Berlin / Heidelberg, November 2009.

[199] Zhijiao Zhang, Naizheng Wang, Yu Chen, Yongqiang Lu, Yuanchun Shi, Rui Wang, and Xiaojuan Lu. Srengine: An osgi-based context-aware inference engine for smart room. In *6th International Conf. Pervasive Computing and Applications (ICPCA)*, pages 267–271, October 2011. `doi:10.1109/ICPCA.2011.6106515`.

[200] Evgeny Zolin. Complexity of reasoning in description logics. http://www.cs.man.ac.uk/ ezolin/dl/, 2008. URL: `http://www.cs.man.ac.uk/~ezolin/dl/` [cited November 15, 2012].

[201] Qiong Zou, Buğra Gedik, and Kun Wang. Spamwatcher: a streaming social network analytic on the IBM wire-speed processor. In *Proc. 5th ACM International Conference on Distributed Event-Based Systems*, pages 267–278, 2011.

Acronyms

ABAC	Attribute Based Access Control
AJAX	Asynchronous JavaScript and XML
BAN	Body Area Network
CEP	Complex Event Processing
CIM	Core Information Model
COPS	Common Open Policy Service
CORS	Cross-origin resource sharing
CPS	Cyber-Physical System
CTL	Computational Tree Logic
CWA	Closed World Assumption
DL	Description Logics
dl	Defeasible Logic
DoS	Denial of Service
DRBAC	Dynamic Role Based Access Control
DSL	Domain Specific Language
ECA	Event Condition Action
ESB	Enterprise Service Bus
FOAF	Friend of a Friend
GUI	Graphical User Interface
HMAC	Hash Based Message Authentication Code
HMIs	Human-machine interface
JVM	Java Virtual Machine

LP	Logic Programming
LTL	Linear Time Logic
MDA	Model-driven architecture
MFOTL	Metric First-Order Temporal Logic
MOOP	Multi Objective Optimisation Problem
MPN	Multi Party Negotiation
N3	Notation 3
OOP	Object Oriented Programming
OSGi	Open Services Gateway Initiative
OWA	Open World Assumption
OWL	Web Ontology Language
PDDL	Planning Domain Definition Language
PDP	Policy Decision Point
PEP	Policy Enforcement Point
RBAC	Role Based Access Control
RCP	Rich Client Platform
RDF	Resource Description Framework
RPC	Remote Procedure Call
SaaS	Software as a Service
SAWSDL	Semantic Annotations for WSDL
SLA	Service Level Agreement
SLP	Service Location Protocol
SOA	Service Oriented Architekture
SoD	Separation of Duty
SWRL	Semantic Web Rule Language
SWT	Semantic Web Technology
TLS	Transport Layer Security
UML	Unified Modeling Language
UNA	Unique Name Assumption
UPnP	Universal Plug and Play
VM	Virtual Machine
WSDL	Web Services Description Language
XACML	eXtensible Access Control Markup Language

APPENDIX A

Defeasible Logic proof theory

The proof theory of defeasible logic, as described by [118] is as follows:

$+\Delta$: If $P(n+1) = +\Delta q$ Then
(1.1) $q \in F$ or
(1.2) $\exists r \in R_s[q]\, \forall a \in A(r) :\ +\Delta a \in P(1..n)$

$-\Delta$: If $P(n+1) = -\Delta q$ Then
(2.1) $q \notin F$ and
(2.2) $\forall r \in R_s[q]\, \exists a \in A(r) :\ -\Delta a \in P(1..n)$

$+\delta$: If $P(n+1) = +\delta$ then either
(3.1) $+\Delta q \in P(1..n)$ or
(3.2.1)$\exists r \in R_{sd}[q] \forall a \in A(r) :\ +\delta a \in P(1..n)$ and
(3.2.2) $-\Delta \neg q \in P(1..n)$ and
(3.2.3) $\forall s \in R[\neg q]$ either
(3.2.3.1) $\exists a \in A(s) :\ -\delta a \in P(1..n)$ or
(3.2.3.2) $\exists t \in R_{sd}[q]$ s.t. $\forall t \in A(t) :\ +\delta a \in p(1..n)$ and $t > s$

$-\delta$: If $P(n+1) = -\delta$ then either
(4.1) $-\Delta q \in P(1..n)$ and
(4.2.1)$\forall r \in R_{sd}[q] \exists a \in A(r) :\ -\delta a \in P(1..n)$ or
(4.2.2) $+\Delta \neg q \in P(1..n)$ or
(4.2.3) $\exists s \in R[\neg q]$ s.t.
(4.2.3.1) $\forall a \in A(s) :\ +\delta a \in P(1..n)$ or
(4.2.3.2) $\forall t \in R_{sd}[q]$ either $\exists a \in A(t) :\ -\delta a \in p(1..n)$ or $t \not> s$

APPENDIX B

Code Snippets

Example template for Situation-Goal policies The following template file illustrates how a human-friendly representation (*input* section) is mapped to an ontology representation which can be loaded into the PDP (*rendering* section).

```
## ########################################################
## This is a policy template. It defines three parts:
## a name, an input and a rendering.
##
## Author: Julian Schütte
##
## Do not remove the following line, it defines the name.
## Name: Situation-based Policy
## ########################################################


## BEGIN INPUT

#set($rule="TestRule")
#set($situation="SOME sbac:Situation")
#set($goal="SOME sbac:Goal")
$rule:
in situation $situation achieve goal $goal

## END INPUT


## BEGIN RENDERING
Prefix: : <http://www.linkality.org/policy/sbac.owl#>
Prefix: sbac: <http://www.linkality.org/policy/sbac.owl#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>

Ontology: <http://www.linkality.org/policy/sbac.owl>

ObjectProperty: sbac:achieve
    Domain:
        sbac:SituationGoalPolicy
    Range:
        sbac:Goal
```

```
ObjectProperty: sbac:in
    Domain:
        sbac:SituationGoalPolicy
    Range:
        sbac:Situation

########### Definition of classes ################
Class: owl:Thing
Class: sbac:Goal
Class: sbac:ProtectionGoal
    SubClassOf:
        sbac:Goal

Class: sbac:Situation
Class: sbac:SituationGoalPolicy
    EquivalentTo:
        (sbac:achieve some sbac:Goal)
         and (sbac:in some sbac:Situation)

########### Definition of individuals ################

Individual: sbac:confidentiality
    Types:
        sbac:ProtectionGoal

Individual: sbac:anomalySituation
    Types:
        sbac:Situation
    Facts:
     sbac:startedBy  sbac:anomalyDetectionEvent

################# Rules Template #####################

Individual: sbac:$rule
    Types:
        sbac:SituationGoalPolicy
    Facts:
     sbac:in  ($situation),
     sbac:achieve  ($goal)

## END RENDERING
```

APPENDIX C

Publications in the context of this thesis

During the work on this thesis, the author has published numerous peer-review publications, presenting intermediate results of this thesis, or being otherwise related to the topic. The following list gives an overview of the publications and states their relation to the thesis.

Atta Badii, Mario Hoffmann, Marco Tiemann, Julian Schütte, Daniel Thiemert, Konstantinos Banitsas. **Dynamic Resource Access Control in the LinkSmart IoTS Middleware Framework**
Submitted to IEEE Transactions on Dependable and Secure Computing. In review at the time of writing this thesis.

Relation to thesis
The article gives an overview of the LinkSmart middleware which has been developed in the EU-funded project HYDRA. The section on *Semantic and Context-Aware Access Control* describes ideas which grew out of this thesis and have been tested against the LinkSmart middleware. The otherwise XACML-based policy framwork of LinkSmart is different from the one we present in this thesis.

Julian Schütte, Roland Rieke, Timo Winkelvos. **Model-based Security Event Management**
Proceedings of the Sixth International Conference Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS-2012), St. Petersburg, 2012

Relation to thesis
The paper presents the event model which is used in the core ECA policy model of the thesis. In the context of the paper, the model is used to detect cross-layer attacks on a dam infrastructure, as an example of a SCADA system.

Roland Rieke, Julian Schütte, Andrew Hutchison. **Architecting an Security Strategy Measurement and Management System**
Workshop on Model-Driven Security (MDSec) at the 15th ACM/IEEE International Conference on Model Driven Engineering Languages & Systems (MODELS), Innsbruck, 2012

Relation to thesis
The paper presents the software architecture for evaluating the event model of the aforementioned paper in context of an automated security measurement and management system. While the software architecture is mainly specific to the EU-MASSIF project, the event model and its evaluation has been created by the author of the thesis and is also found in the thesis' policy framework.

Julian Schütte, Mark Gall, Hervais Simo Fhom. **Security Policies in Dynamic Service Compositions**
Proceedings of the International Conference on Security and Cryptography (SECRYPT 2012), Rome, 2012

Relation to thesis
This paper presents the metapolicy-based merging of policy decisions in multi-domain settings, as it is supported by the policy framework.

Julian Schütte. **Goal-based Policies for Self-Protecting Systems**
Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA 2012), Fukuoka, 2012

Relation to thesis
In this paper, the author of the thesis presents a novel "Situation-Goal" policy pattern, and describes how it can be easily realised with the policy framework, by extending concepts of the traditional ECA pattern in the framework.

Julian Schütte. Apollon: **Towards a Semantically Extensible Policy Framework**
Proceedings of the International Conference on Security and Cryptography (SECRYPT 2011), Sevilla, 2011

Relation to thesis
This paper is a presentation of the overall software architecture of the policy framework.

Julian Schütte, Stephan Heuser. **Auctions for Secure Multi-Party Policy Negotiation in Ambient Intelligence**
Proceedings of the Seventh International Symposium on Frontiers in Networking with Applications (FINA 2011), Singapore, 2011

Relation to thesis
This paper presents the policy negotiation module which allows to find an optimal selection of multiple applicable obligations. It has been realised as a module for the policy framework which allows a multi-party agreement on obligations. In this publication, we moved away from Genetic Algorithms and applied instead a more generic and flexible auction-based approach.

Weishan Zhang, Klaus Marius Hansen, João Fernandes, Julian Schütte, Francisco Milagro Lardis. **QoS-aware Self-adaptation of Communication Protocols in a Pervasive Service Middleware**
Proceedings of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom 2010), Hangzhou, 2010

Relation to thesis
In this paper, we used Genetic Algorithms to find an optimal configuration of a middleware. Later, this approach has been modified and transferred into a generic policy negotiation strategy (c.f the FINA2011 paper).

Julian Schütte, Nicolai Kuntze, Andreas Fuchs, Atta Badii. **Authentic Refinement of Semantically Enhanced Policies in Pervasive Systems**
Proceedings of the 25th International Information Security Conference (IFIP SEC 2010), Brisbane, 2010

Relation to thesis
In this paper we show how Semantic Web based policies in combination with a respective refinement process can be used to automatically negotiate obligations which match the policies and capabilities of different services. This refinement process has been incorporated into the policy framework described in the thesis. Additional strategies to attest the correctness of the announced service capabilities have been contributed by Kuntze and Fuchs and were not included in the design of the framework's core architecture, as they put strong demands on specific security hardware. Nevertheless, they could of course be added in form of supplementary modules in case such hardware is available.

Julian Schütte, Tobias Wahl. **A Description Logic based Approach on Handling Inter-Domain Policy Conflicts using Meta-Policies**
IEEE Vehicular Technology Magazine, Volume 5, Number 3, pp. 68-74, September 2010

Relation to thesis
This paper presents a first version of the conflict resolution approach described in the thesis, which allows to combine policy decisions from multiple domains, regulated by metapolicies. The paper demonstrates how metapolicies are used to classify strict and weak policy decisions and shows how conflicts between metapolicies themselves can be detected by reasoning over their Description Logic representation. In contrast to the final version described in the thesis, the paper does however not yet consider obligations.

Weishan Zhang, Julian Schütte, Mads Ingstrup, Klaus M. Hansen. **A Genetic Algorithms-based Approach for Optimized Self-protection in a Pervasive Service Middleware**
Proceedings of the Seventh International Conference on Service Oriented Computing (ICSoC 2009), 2009

Relation to thesis
The main contribution of the paper is a "self-protection workflow" which orchestrates an event detection engine, a Genetic Algorithm optimisation engine, a planning engine and an obligation enforcement layer. By the application of this workflow to a pervasive system middleware, we illustrate how it is possible to autonomously achieve an optimal trade-off between conflicting criteria like security and performance. Sections 2.2, 3.1 and 4 have been written by the author of the thesis and describe how protection goals and -mechanisms can be modelled in ontologies and how this information can be used to find appropriate obligations matching abstract requirements and platform limitations.

Tobias Wahl, Julian Schütte. **Security Mechanisms for an Ambient Environment Middleware**
Workshop Proceedings of the 32nd Annual Conference on Artificial Intelligence (KI), International Workshop on Distributed Computing in Ambient Environments (DiComAe), 2009

Relation to thesis
In this paper, we state the necessity of semantically modelling security capabilities of services. Later, this approach led to the development of DL-based policies.

M. Hoffmann, M. Mattheß, J. Schütte, A. Badii, R. Nair, D. Thiemert, S. Engberg. **Towards Semantic Resolution of Security in Ambient Environments**
Ambient Intelligence Developments (Ami.D), 2007

Relation to thesis
This publication has been the first one by the author of this thesis on the overall area of Ambient Intelligence and Pervasive Systems.