Policy and Security Configuration Management in Distributed Systems

Simone Mutti
Ph.D. course in Mechatronics, Information Technology,
New Technologies and Mathematical Methods
XXVII Cycle

Advisor: Prof. Stefano Paraboschi
Department of Management, Information and Production Engineering
Università degli Studi di Bergamo, Italy

Università degli Studi di Bergamo, Italy
Department of Engineering and Applied Sciences

February, 2015

Contents

1 Introduction **9**
1.1 Document Structure . . . 10
1.2 Publications arising from this thesis . . . 11
1.3 Acknowledgments . . . 12

I Security Management in ICT Systems **13**

1 Introduction **15**
1.1 Vision and Solution Overview . . . 17
1.2 Contribution to the PoSecCo project . . . 20

2 Policy based Security Management **23**
2.1 Functional System metamodel . . . 24
2.1.1 The Business Layer . . . 26
2.1.2 The IT Layer . . . 27
2.1.3 The Infrastructure layer . . . 28
2.2 Security metamodel . . . 30
2.2.1 Business Security metamodel . . . 30
2.2.2 IT layer Security metamodel . . . 32
2.2.3 Configuration metamodel . . . 43
2.3 Model specialization . . . 46

3 Policy analysis and harmonization **51**
3.1 State of the Art . . . 52
3.1.1 Policy Harmonization checking . . . 52
3.2 Checking techniques . . . 58
3.2.1 Standard DL reasoner . . . 58
3.2.2 Ad-hoc reasoning methods . . . 58
3.2.3 Rule based inferencing . . . 61
3.3 Reasoning patterns . . . 62
3.3.1 Property verification rules . . . 63
3.3.2 Violation detection rules . . . 65
3.3.3 Custom module pattern . . . 66
3.4 Types of check . . . 67
3.4.1 Structural Checks . . . 67

3.4.2 Consistency Checks . . . 69
3.4.3 Separation of Duty Checks . . . 83
3.5 Policy Incompatibility . . . 85
3.6 Policy Minimization . . . 87
3.7 Separation of Duty . . . 88
3.8 Implementation and Experiment . . . 89
3.8.1 Experimental results . . . 91

4 Policy Minimization 97
4.1 Model . . . 99
4.1.1 Conflict resolution strategies . . . 104
4.2 Redundancy . . . 106
4.3 Implementation . . . 110
4.3.1 MIPP to Weighted SAT . . . 110
4.3.2 Heuristic Algorithm for MIPP . . . 111
4.3.3 MPP to Weighted SAT . . . 112
4.3.4 Heuristic Algorithm for MPP . . . 113
4.4 Experimental Results . . . 114

5 Policy Refinement 119
5.1 Overall workflow . . . 119
5.2 Enrichment of IT Policies . . . 121
5.2.1 Enrichment process and enrichment modules . . . 123
5.2.2 Types of enrichment techniques . . . 124
5.3 Relation with harmonization modules . . . 128
5.4 Relation with refinement modules . . . 133
5.4.1 Enrichment of IT Policies . . . 134
5.5 Refinement of IT Policies . . . 135
5.5.1 Select, Create and Connect (SCC) . . . 135
5.5.2 Select and Construct (SC) . . . 138

6 Software for the IT Policy language 143
6.1 Requirements . . . 144
6.2 Architecture . . . 145
6.2.1 Source Code . . . 147
6.2.2 Input/Output ITPolicy Tool . . . 149
6.3 Editor . . . 149
6.4 Harmonization . . . 151
6.5 Refinement . . . 153
6.5.1 Enrichment . . . 154
6.6 Additional functionalities . . . 155
6.6.1 Quality of access control policies . . . 155

II Security Management in Mobile Systems 157

1 Introduction 159
1.1 Rationale of the approach 160

2 Android Security Architecture 161
2.1 SEAndroid 162
2.2 Threat Model 163
2.2.1 Example 164

3 SELinux Policy Model 167
3.1 Model specification 167
3.2 Requirements 170
3.3 Policy Module Language 174
3.3.1 Correctness 176

4 Mapping Android Permissions 179

5 Implementation 183
5.1 Changes to SELinux 184
5.2 Changes to Android 185
5.2.1 Update SELinux policy 185
5.2.2 Update seapp_contexts 185
5.2.3 Update mac_permissions.xml 186
5.3 Performance 186
5.3.1 Installation time 186
5.3.2 Runtime 188

6 Related Work 191

7 Conclusions 193

A Comparison with other proposals 195
A.1 XACML 195
A.2 PCIM 197
A.3 KAoS 200
A.4 Summary of the analysis 200

B Model specialization example 203
B.1 Model specialization for Operating Systems 203
B.2 Model specialization for Database 204
B.2.1 DatabaseInterface 205
B.2.2 DBMS 205
B.2.3 Database and SQLDatabase 205
B.2.4 SQLDataObject, SQLTable, SQLTableColumn and SQLView 206

C Policy Incompatibility **207**
C.1 Policy Incompatibility SWRL Rules 207
C.2 Preprocessing Algorithms 208
C.3 Permission-Based and Object-Based SoD 208
C.4 Optimization 210

D Technology **213**
D.1 Eclipse 213
D.2 Web Ontology Language 214
D.2.1 OWL API 215
D.2.2 Reasoner 215
D.2.3 Semantic Web Rule Language 216
D.2.4 Simple Protocol And RDF Query Language - Description Logic 216

E Refinement Modules **219**

F Enrichment Modules **223**
F.1 NeOn ToolKit 226
abstract

“It is paradoxical, yet true, to say, that the more we know, the more ignorant we become in the absolute sense, for it is only through enlightenment that we become conscious of our limitations. Precisely one of the most gratifying results of intellectual evolution is the continuous opening up of new and greater prospects.”

-Nikola Tesla

1
Introduction

The evolution of information systems sees a continuously increasing need of flexible and sophisticated approaches for the management of security requirements. On one hand, systems are increasingly more integrated and present interfaces for the invocation of services accessible through network connections. On the other hand, system administrators have the responsibility to guarantee that this integration and the consequent exposure of internal resources does not introduce vulnerabilities. The need to prove that the system correctly manages the security requirements is not only motivated by the increased exposure, but also by the need to show compliance with respect to the many regulations promulgated by governments and commercial bodies.

Given the critical role of security, concerns arise about the correctness of the policy. It is not possible anymore to rely on the security designer to have a guarantee that the policy correctly represents how the system should protect the access to resources. Tools are needed to support the analysis of the security policies, and a crucial element that signals problems in the policies is represented by the presence of conflicts, i.e., contradictions or ambiguities in the policy specification, which may lead to anomalies in the application of the policy.

The integrated management of security policies promises to produce significant benefits in this area. In a way similar to the evolution seen in the area of software development and database design, the model-driven engineering of security leads to the specification of security requirements at an abstract level, with the subsequent refinement of the abstract model toward a concrete implementation. It is then guaranteed that, when the high-level representation of the security requirements is correct, the security configuration of the system satisfies the requirements. The application of this approach requires the implementation of a rich collection of tools supporting, for each of the many

components in the system, the refinement from the high-level representation of security requirements to the concrete security configuration. A significant investment in research and development is needed to realize this vision, in order to manage the heterogeneous collection of devices and security services that information and communication technology offers to system administrator. Still, this large investment promises to produce adequate returns, considering the importance that the correct management of security requirements presents in modern information systems.

A crucial advantage of the model-driven approach described above is the possibility of an early identification of anomalies in the security policy. Security policies in real system often exhibit conflicts, i.e., inconsistencies in the policy that can lead to an incorrect realization of the security requirements, and redundancies, i.e., elements of the policy that are dominated by other elements, increasing the cost of security management without providing benefits to the users or applications. The availability of a high-level and complete representation of the security policies supports the construction of services for the analysis of the policies able to identify these anomalies and possibly suggest corrections.

1.1 Document Structure

The document is organized in two parts. Part I describes a set of approaches and tools that have been designed in order to create an environment for the design and management of security policies that follows the approach proposed by the European project PoSecCo (`http://www.posecco.eu`). The author's contributions to the PoSecCo project are manifolds. They range from the definitions of the metamodel to the policy analysis.

However, due to the complexity of the project, this document also includes contributions developed by other partner[1], in collaboration with the author, in order to give the reader the opportunity to understand the overall scenario and to better comprehend the work done by the author.

- Chapter 1 introduces the overall scenario, the current limitations and the vision proposed by the PoSecCo project to overcome these limitations.

- Chapter 2 presents the metamodel that has been designed for the representation of the security policy. The IT metamodel will be used as a foundation for the realization of the tools, and in the PoSecCo policy chain it represents the abstraction level that connects the representation of security requirements at the Business level to the configuration at the Infrastructure level.

- Chapter 3 discusses three policy properties: *Policy Incompatibility* (given a set of authorizations A, check whether exist pairs of authorizations (a_1,a_2) such that a_1 and a_2 apply to the same request and have opposite sign),

[1] The contributions coming from PoSecCo partner, included in this document will be duly highlighted.

Policy Minimization (given a set of authorizations A, check whether a subset of those authorizations R exists that is dominated by other authorizations), and *Separation Of Duty Satisfiability* (given a set of authorizations A, check whether they satisfy a set of Separation of Duty (SoD) constraints).

- Chapter 4 presents a formalization of the redundancy problem in access control policies that considers three different ways in which a security administrator can act on a policy containing redundancy.
- Chapter 5 describes the techniques for creating the policy chain with specific focus on the *enrichment* of the IT level and *refinement* from IT level to Infrastructure level.
- Chapter 6 describes the design principles of the *IT Policy Tool*, which has been designed for the creation of the security policy at the IT level. The IT Policy Tool supports the creation and the maintenance of the IT Policy and is implemented as an Eclipse plug-in and deployed as a RAP Application.

Part II describes the work done regarding policy management in the are of mobile systems

- Chapter 1 and Chapter 2 provides an overview of the Android security architecture, describing the role of the MAC model introduced by SEAndroid, and the threat to third-party apps that the policy modules want to mitigate;
- Chapter 3 presents a model of SELinux policies, used to formalize the requirements that policy modules have to satisfy;
- Chapter 4 illustrates how the use of appPolicyModules can improve the support of Android permissions;
- Chapter 5 describes the performance results;
- Chapter 6 provides a comparison with previous work in the area;
- Chapter 7 draws a few concluding remarks.

1.2 Publications arising from this thesis

This Section presents a list of the published and submitted papers that have arisen from the work in the thesis.

Papers in Proceedings of International Conferences and Workshops:

- Enrico Bacis, Simone Mutti, and Stefano Paraboschi. "AppPolicyModules: Mandatory Access Control for Third-Party Apps." in 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015) [Part II]

- Marco Guarnieri, Eros Magri, Simone Mutti, and Stefano Paraboschi. "On the notion of redundancy in access control policies." in 18th ACM Symposium on Access Control Models and Technologies (SACMAT 2013) [Chapter 4]
- Mario Arrigoni Neri, Marco Guarnieri, Eros Magri, Simone Mutti, and Stefano Paraboschi. "Conflict detection in security policies using semantic web technology." in 1st International IEEE-AESS Conference in Europe about Space and Satellite Telecommunications - Security and Privacy Special Track (ESTEL 2012) [Chapter 3]
- Marco Guarnieri, Eros Magri, and Simone Mutti. "Automated management and analysis of security policies using eclipse." in 7th Italian Workshop on Eclipse Technologies (Eclipse-IT 2012) [Chapter 3 and Chapter 6]
- Simone Mutti, Mario Arrigoni Neri, and Stefano Paraboschi. "An Eclipse plug-in for specifying security policies in modern information systems." in 6th Italian Workshop on Eclipse Technologies (Eclipse-IT 2011) [Chapter 2 and Chapter 6]

Short Papers in Proceedings of International Conferences and Workshops:

- Mario Arrigoni Neri, Marco Guarnieri, Eros Magri, Simone Mutti, and Stefano Paraboschi. "A Model-driven Approach for Securing Software Architectures." in 10th International Conference on Security and Cryptography (SECRYPT 2013) [Chapter 2]

Chapters in Books:

- Cataldo Basile, Matteo Maria Casalino, Simone Mutti and Stefano Paraboschi. "Detection of conflicts in security policies." Computer and Information Security Handbook (Morgan Kaufmann Series in Computer Security) [Chapter 2 and 3]

Under submission at the time of writing:

- Simone Mutti, Enrico Bacis, and Stefano Paraboschi. "SeSQLite: Security Enhanced SQLite." in 24th USENIX Security Symposium (USENIX 2015) [Part II]

1.3 Acknowledgments

The research leading to the results documented in this thesis was supervised by the Prof. Stefano Paraboschi (Università degli Studi di Bergamo) and has received funding from the European Community's Seventh Framework Programme within the 7FP and H2020, respectively, under grant agreements 257129 and 644579, by the Italian Ministry of Research within the PRIN projects "PEPPER" and "GenData 2020", and by a Google Research Award (Winter 2014).

Part I

Security Management in ICT Systems

1 Introduction

The *Future Internet* (FI) market of services can cover various degrees of standardized and individualized offerings: Commodity services are standardized, off-the-shelf services, typically billed for on an 'as-used' basis and with little means for customization (e.g., email, spam filtering or simple financial accounting services). Specific customer needs are addressed by highly customized services that address particular functional or non-functional requirements. Both standardized and custom business services will require service providers to operate IT, application and infrastructure services with multi-customer functionality, that is, the capability to serve multiple customers with appropriate data segregation [65].

Future Internet (FI) applications will be static or dynamic compositions of services of different kinds and layers, ranging from low-level infrastructure services for data storage or bandwidth up to high-level IT services that support common business processes such as invoicing or enterprise resource planning. The technical composition of such services is accompanied by a multitude of contractual, binding agreements between service providers and consumers, on functional and non-functional aspects.

Figure 1.1 shows the viewpoint of one particular service provider that offers a couple of business services to its customers. A business service in that context can be understood as a commercial or non-commercial offering, which bundles a set of IT services. Business services are the basis for contractual agreements, and are subject to auditing standards such as the "Statement on Auditing Standards No. 70: Service Organizations" (SAS70) [AICPA]. Business services do not only cover business to business services, but also services offered to end-users. Business services focus on the real-world problem domain and its related terminology and processes, as opposed to the technology focus of lower-level services. IT services support one or more business processes, such as

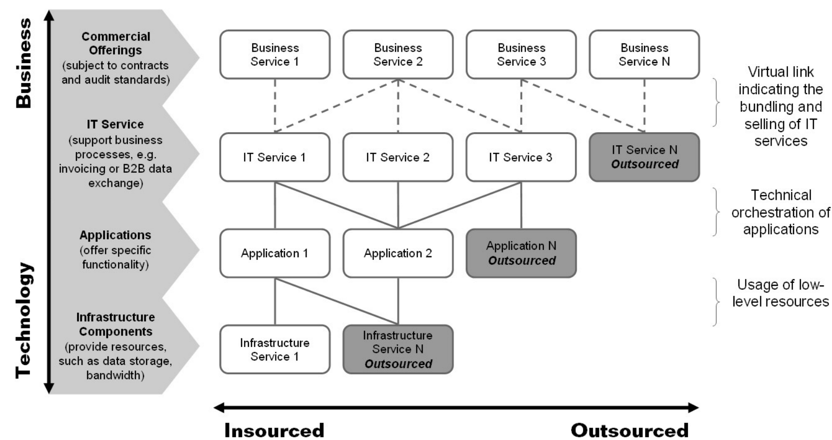


Figure 1.1: Architecture of FI applications from the viewpoint of a service provider.

human capital management. They are perceived by customers and users as self-contained, single, coherent entities [ITILv2] and are realized as orchestrations of technical application services which provide a single, specific functionality. The operation of application services relies on underlying infrastructure services that provide resources such as network bandwidth, application servers or data stores.

Service providers in such FI application scenarios have the freedom to integrate and mix services of all kinds for their specific offerings, according to their particular competencies and customer segments. As such, numerous combinations of *insourced* and *outsourced* services can be employed by future market participants, in its extremes ranging from service providers that operate entire service landscapes by themselves, to service providers that re-brand and re-sell IT services of suppliers.

The adoption and success of outsourcing in general and for FI applications in particular depends crucially on the market actors' capabilities to pursue and align two interdependent goals: (a) The implementation of each stakeholder's security requirements and compliance with all regulatory requirements, herewith establishing consumer trust in FI applications and (b) the profitable management and operation of its services.

Achieving and maintaining the required security level and providing the necessary evidence to customers, regulatory bodies and internal stakeholders in a cost-efficient manner is hindered by various issues:

- the multitude of security stakeholders with individual requirements;
- the increasing number and complexity of regulatory requirements that is multiplied by the number of countries a service provider and its customers are active in;

- overlaps or conflicts among high-level policies and enforcing security configuration settings of the various application and infrastructure services;

- the steady evolution and change of regulatory requirements, technology components and business relationships.

Even today, where most organizations operate the bigger share of the service landscape themselves, above issues lead to cost-intensive and error-prone security processes in which high-level policies are formulated and maintained in prose and manually translated into lower-level, service-specific configuration settings [22]. These processes have a well-known impact on the trustworthiness of IT infrastructures, many times confirmed by studies such as from the NSA, which found that '*inappropriate or incorrect security configurations (most often caused by configuration errors at the local base level) were responsible for 80 percent of Air Force vulnerabilities*' [70, 91, 94].

Such studies show that a correct configuration of compositions of application or infrastructure services is at least as important as the application security of the individual services. Besides creating exploitable security vulnerabilities, incorrect security configuration may have a significant impact on the availability of individual services and as such the overarching business services which are typically bound to corresponding service level agreements. Additional complexity arises in FI application scenarios, where larger shares of the service landscapes are outsourced. The process of breaking high-level requirements down to low-level settings spans across organizational boundaries, and service providers need to ensure that the outsourcing service providers properly reflect the policies that result from its own customers. A service provider must therefore check that a contractual agreement with an outsourcing service provider covers all its own customers' requirements.

1.1 Vision and Solution Overview

PoSecCo's vision is to establish and maintain a consistent, transparent, sustainable and traceable link between high-level, business-driven security and compliance requirements on one side and low-level technical configuration settings of individual services on the other side.

Such an end-to-end link shall be maintained in operating conditions, i.e. considering constant evolution that result in changes of two kinds:

- New or changing security and compliance requirements as a result of, for instance, new business service offerings, new customers or suppliers, changing security needs of existing customers or new legal regulations and security standards;

- *Landscape* changes as a result of, for instance, ordinary administration tasks that change configuration settings or changes of application and infrastructure services in the course of purchase, in- or outsourcing decisions.

PoSecCo maintains this end-to-end link by automated means where possible and offers decision support where human interaction is inevitable. To deal with these two kinds of changes, PoSecCo simultaneously thrives for a (i) top-down, policy-driven approach and a (ii) bottom-up, landscape-driven approach:

First, the top-down approach takes as input the various *laws*, *regulations*, *best practices* and *standards* for security and compliance, captures them by *policies* that are more detailed, prose descriptions of security and compliance objectives and translates them into *IT policies* which relate the high-level requirements to the actual IT landscape and infrastructure.

Second, the bottom-up approach builds on top of common *Change and Configuration Management* (CCM) and *Audit software*, whose industry adoption is steadily growing and which offers a common configuration interface for the various application and infrastructure services that compose FI applications - herewith extending the service-oriented architecture (SOA) concept of services being self-contained components with defined business interfaces towards defined configuration interfaces on the basis of, for instance, the OASIS Web Services Resource Framework [44].

Example 1. *By way of illustration, 'protect stored cardholder data' is a high-level business policy, defined by the PCI DSS (Payment Card Industry Data Security Standards) [35]. It states an objective in a declarative manner. The abstract business policy can be mapped to a set of declarative IT policies such as 'make the credit card database inaccessible from the Internet'. This IT policy can be mapped to the configuration 'block at the firewall all traffic between the Internet and the credit card database', which is imperative but abstract because no firewall can interpret this instruction. Settings are finally obtained by translating the configuration into a language that can be understood and executed by an actual firewall.*

PoSecCo solely relies on such configuration interfaces to read and write configurations - in particular, the changing of source code and rebuilding of a service is not a feasible option during operation time and as such out of the scope of PoSecCo. Whereas CCM software is used to (a) update representations of the actual landscape and (b) create change requests for configuration corrections of productive services, PoSecCo relies on common Audit software to (c) audit the productive landscape with help of standardized, comparable checklists and checks.

The establishment and maintenance of an end-to-end link between abstract requirements and technical configuration settings requires business service providers to perform the following three activities (see Figure 1.2):

1) Following the *top-down* approach: The capturing and harmonization of all stakeholders' policies to obtain a conflict-free set of IT policies.

This analytical activity involves various internal security stakeholders that firstly capture the relevant business policies and secondly transform these into formal descriptions of IT policies that allow reasoning about policy dependencies and particularly conflicts.

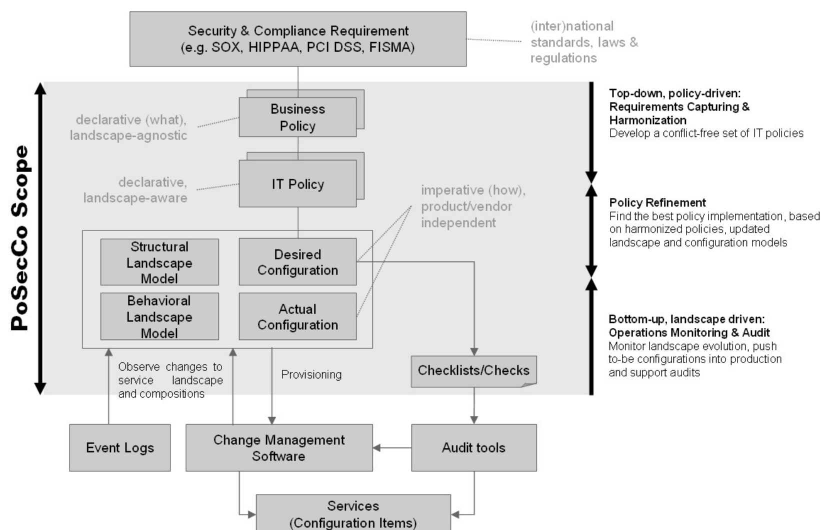


Figure 1.2: PoSecCo concepts and scope.

2) Following the bottom-up approach: The monitoring and audit of the operative service landscape and the configuration settings of its elements.

Currently, these actions are performed manually, resulting in error-prone and inefficient processes. As one result, PoSecCo turns these actions into automated processes where possible, and, thus, avoiding some of the risks for errors and increasing the overall efficiency.

As a prerequisite, the creation and maintenance of the service landscape model is required. A service landscape model is an abstraction of the set of all services and their compliance-relevant functionality. The structural service landscape model founds on simple inventory lists as maintained in CCM software and which describe component versions, patch levels, etc. as well as simple component dependencies such as the network topology. As such simple enumerations do not represent logical dependencies between the services that result from their composition in the context of specific business service offering. They need to be enriched with semantic, compliance-relevant information resulting from their usage context. Process mining and process modeling techniques will support the creation and update of such semantically enriched, behavioral service landscape models, herewith reducing manual efforts. In addition to landscape models, configurations abstract the active configuration settings of all relevant services. All models, structural and behavioral descriptions of service landscapes, and configurations will be the basis of the policy refinement activity.

The audit of operative landscapes shall happen through various means: Firstly, through the comparison of actual and desired configurations and secondly through the generation of standardized audit checklists and checks that may be processed by common Audit software and Vulnerability Scanners.

However, current languages lack some expressiveness with regard to the assessment of misconfigurations as well as automated corrections and focus on the audit of single services instead of entire landscapes. As it is hardly possible to keep actual settings and desired configurations synchronized, the assessment provides decision support for the selection of an appropriate remediation strategy.

3) Results of the previous activities are used in the refinement activity, which is an iterative process that transforms declarative IT policies into a consistent set of desired configurations that may serve as input for the generation and deployment of actual settings through CCM software.

The refinement of policies towards a deployable configuration relies on the description of the actual landscape (the service landscape model, created and updated during landscape monitoring), which supports the decision process by eliminating unfeasible configuration alternatives (reality check) and as such reducing the overall space of all possible configuration options.

The establishment and maintenance of an end-to-end link between high-level requirements and low-level settings through above-mentioned activities depends on the interaction and contribution of many different stakeholders within and outside a service provider's organization. Several of their tasks are hardly automatable (e.g., the identification and formalized capturing of policies that result from internal or external requirements, or the final decision on and approval of a given set of policy enforcing configurations). For that reason, PoSecCo aims to support the stakeholders' interaction and decision making processes by the analysis and development of suitable organizational processes and structures as well as models that investigate the costs of policy management in FI application scenarios.

1.2 Contribution to the PoSecCo project

Due to the complexity of the PoSecCo project and the involvement of several partner, in the following the author identifies the boundaries of his contributions (as already mentioned the other partner's contributions will be briefly presented in order to give to the reader an overall view of the entire project):

- The *Functional System metamodel* will be briefly presented and was developed by Politecnico di Torino and Bern University of Applied Science;
- The *Business Security metamodel* will be briefly presented and was developed by IBM and University of Innsbruck;
- The *IT Security metamodel* will be extensively presented and was developed by Università degli Studi di Bergamo;
- The *Configuration metamodel* will be presented and was developed by Bern University of Applied Science, Politecnico di Torino, Università degli Studi di Bergamo and SAP;

- The *Policy analysis and harmonization* and *Policy minimization* will be presented and was developed both theoretically and practically (i.e., IT Policy tool) by Università degli Studi di Bergamo;
- The *Policy Refinement* and *Policy Enrichment* process will be presented and was developed by Università degli Studi di Bergamo. This document only includes the approaches used in the *topological independent* scenario. The *topological dependent* scenario was developed by the Politecnico di Torino but will not be presented in this document.

2

Policy based Security Management

A crucial aspect for the evolution of security management is a better integration in security policy management. Configuring such a policy for a specific system in isolation is not trivial, but it is not at all problematic, since we can benefit from sophisticated access control models that have been developed for a variety of systems, from relational database management systems to application servers. The integration and harmonization of security policies specified in different systems at different levels are undoubtedly a much harder obstacle. Three clear integration perspectives can be identified:

- *Conceptual integration*: security policies have to be described at different levels of abstraction, from the business level to the concrete configuration of modules and devices. Separate models are required for the different levels, as shown by software engineering practice in many areas. Also, there is the need for support in the translation of the high-level policy to a more concrete specification. A precise description of the correspondence between the policies at different levels allows for a more effective and efficient verification of the compliance of the concrete policy with the high-level security requirements. In addition, a structure with different abstraction levels greatly facilitates the maintenance of a security policy.

- *Vertical integration*: the structure of a modern information system presents several components that can be represented in a vertical stack: physical hardware, virtual hardware, operating system, network, DBMS, application server, application. Security policies can be supported at each of these layers, and they are typically defined independently, but there exists the possibility for an integration process, which can lead to a greater level of security and flexibility.

- *Horizontal integration*: Compared to the classical scenarios considered in access control, where a policy is assumed to be enforced by a specific reference monitor, modern information systems present a variety of computational devices cooperating in the response to a specific user request. Components of the computational infrastructure can be owned by independent parties. In these scenarios, the management of security policies requires to carefully define models and mechanisms able to map a security requirement to a coordinated policy enforced by such varied parties. This aspect is particularly difficult when few hypotheses can be made about the specific security management functionality supported by the service providers.

The PoSecCo project plans to investigate these three aspects. Conceptual integration will rely on the design of metamodels structured with three levels: *Business*, *IT*, and *Infrastructure*. Vertical integration will specifically focus on the harmonization between access control and network configuration. Horizontal integration will be tackled in a Future Internet scenario, where applications are realized integrating the services of a variety of providers. A common objective will be the detection and resolution of conflicts in the policies.

2.1 Functional System metamodel

The *Policy based security management* paradigm is emerging on the security scene, because it decouples the security requirements, expressed using Service Level Agreements (SLA), proofs of compliance with a variety of high-level directives, from the mechanisms that will actually enforce them and the environment where resources to protect and security mechanisms are displaced. The translation process, named *policy refinement*, is manually performed by security administrators. Today, policy refinement asks administrators a huge investment in term of needed effort, but due to its complexity, this investment is not repaid with a good security level. According to recent studies, a significant number of system vulnerabilities, as well as actual security breaches, are due to inappropriate security configurations [42, 98, 99].

Many decisions need to be made when refining a policy to configurations. Due to the nature of policy enforcement and conflict analysis tasks, that do not depend only on the chosen policy, these decisions are affected by the environment where the policy will be enforced. Analogously, to perform discrepancy analysis a clear picture of the underlying system is needed. For example, by knowing that two virtual machines are on the same physical host, an administrator may decide not to configure a secure channel, or by knowing that there is a reverse proxy in front of a web server he may decide to configure authentication functionalities and terminate the secure connections at the proxy. Many more examples can be made to illustrate the dependence of the policy refinement and conflict analysis on the landscape.

The representation of the landscape must be expressive enough to enable automatic refinement activities, to recognize policy-enabled elements and iden-

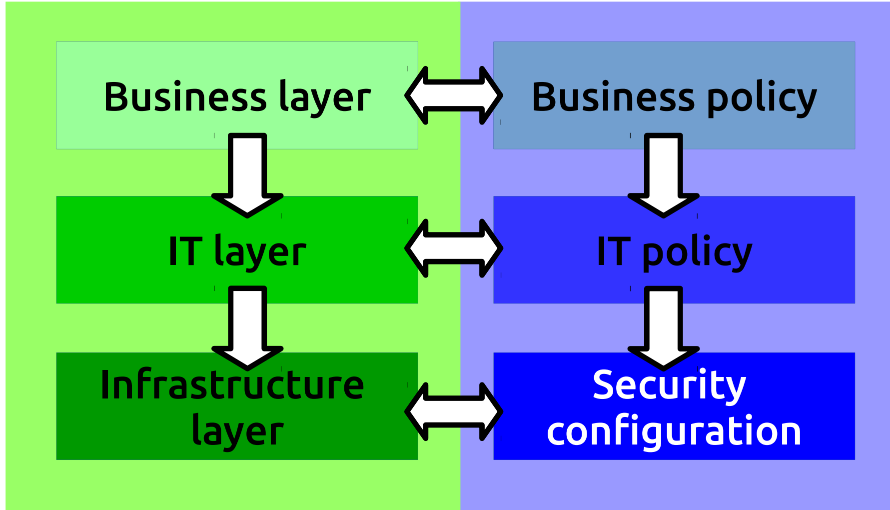


Figure 2.1: PoSecCo architecture.

tify their security capabilities in order to decide which need to be configured. Additionally, as evident from previous example, it is also necessary to know the topological arrangement of resources and security mechanisms. Since policies are often stated as having the data to protect in mind, a valid landscape description must be able to convey information about the data (used, stored, exchanged, etc.).

Since an excessive amount of information cannot be requested from administrators, PoSecCo relies on a CMDB to provide details about the infrastructure. Still, an appropriate level of description needs to be identified to limit the effort. This is the main reason why PoSecCo decided to develop a new landscape meta-model instead of using the Common Information Model. However, the availability of a metamodel does not exclude the possibility of having different model specializations that may be defined by external parties in order to model concepts useful for their scenarios.

The Functional System metamodel (see Figure 2.1) is vertically organized according to the three PoSecCo abstractions: business, IT and infrastructure layer. Going from business to infrastructure layer, the level of detail on the landscape increases both at the organization and technological level, while the terminology moves from a business-oriented to a technical vocabulary. These abstractions are horizontally connected to the corresponding policy abstractions, the business policy, the IT policy, and, at the infrastructure layer, both the logical associations and the configurations.

We present here a brief description of the content of each layer:

- *Business layer* contains the description of the business services, the busi-

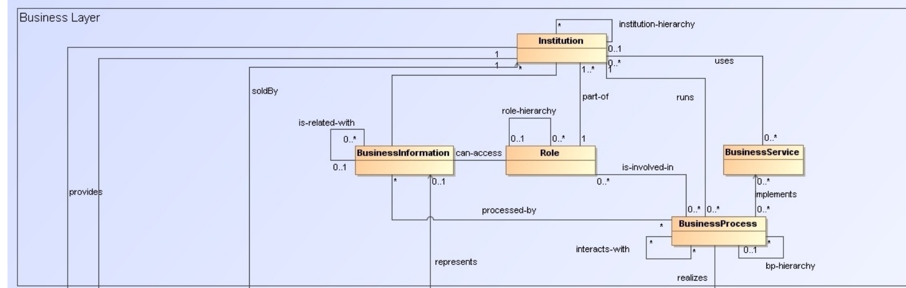


Figure 2.2: The Business abstraction layer.

ness processes and the institutions, along with other business oriented concepts;

- *IT layer*, gathers all the classes that are used to describe an IT service choreographically, describing the interaction between all the components, which are represented by abstract resources which expose different interfaces. Moreover, in order to satisfy the service providers' needs, we introduce a set of classes that can be used to describe the different implementations of each service that can be offered to a customer, which constitute a service provider catalog;

- *Infrastructure layer*, bundles the classes necessary to describe networked ICT systems and software which run actual service implementation sold to the customers. The networks topology of these systems can be described resorting to the node class which can be used to represent any type of physical or virtual network node (e.g., computer, router, switch). Then it contains the description of all the software components used to provide the different services and running on nodes. Moreover, this layer contains classes necessary to describe the functionality and security configuration of each hardware or software component in details, e.g. authentication logging or filtering capabilities. Finally the location where the ICT system are located is represented by the location class, which can be used to identify different country-related regulations.

2.1.1 The Business Layer

For a complete description of the business layer of the metamodel, depicted in Figure 2.2, we refer to the Deliverable [59]. In this section we recall only a few classes, which are used in the rest of this document.

The *Institution* class represents an organizational unit. It can be used to describe either a customer that buys services from a service provider, or a supplier that delivers services to the service provider, e.g., an outsourcing provider for IT services (that contributes in terms of business functionality to the service

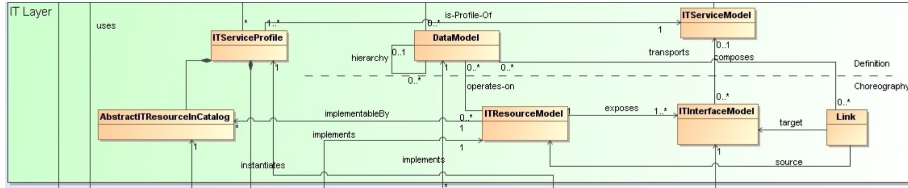


Figure 2.3: The IT abstraction layer.

provider's own offering) or infrastructure services (e.g, application platforms, storage, bandwidth).

The *Role* represents an actor within an institution. Several roles may be arranged hierarchically and assigned to an *Institution*.

The *BusinessInformation* class is used to represent any type of information at business level and can be associated to *Institution*s or *Role*s, in order to specify access rights. Several *BusinessInformation*s may be associated to represent complex information types.

The *BusinessProcess* class can be used to represent interactions between *Institution* and/or *Role* instances. Several *BusinessProcess*es may be correlated hierarchically or describing interactions, they may involve a set of roles and process a set of *BusinessInformation* objects.

2.1.2 The IT Layer

The main class of the IT layer (see Figure 2.3) is the *ITServiceModel*. A service model is an abstract description of an **IT Service** that supports one or more business processes (defined in the Business Layer see Section 2.1.1). This is modeled with the relation *realizes*. An IT Service is characterized by its interfaces and implemented by abstract software components.

An **Interface** is a point of interaction, exposed by a resource that may contribute in providing an IT service and used by customers via business processes or by other IT services. The functionality associated to an interface is actually provided by a resource model (class *ITResourceModel*), which represents an abstract (software) component.

The interfaces provided by an *ITServiceModel* are described using the *ITInterfaceModel* class and attached using the *composes* relation. Each of these interfaces is exposed using the *exposes* relation by an *ITResourceModel*, which represents an abstract description of a service component.

The **communication relationships** between abstract service components are described with the class *Link*. Each service interface (*ITInterfaceModel*) exposed by a component (*ITResourceModel*) can be linked to another component using this class. This result is achieved using the relations *source*, between the classes *Link* and *ITResource*, and *target* between the classes *Link* and *ITInterfaceModel*. Using *ITResourceModel* and *Link*, we can describe the relations among abstract software components, thereby defining their usage hierarchy

(i.e., the choreography).

Resource interaction is not only considered (and modeled) for functional elements of an IT service (i.e., elements that contribute to the original purpose of the service consumed by a customer) but also for management elements of the IT infrastructure. Such elements (e.g., the administration console of an application server) result from an implementation choice of the service provider, e.g., the decision of deploying a functional IT resource X in the application server Y. Even though management interfaces are primarily used by the service provider and as such transparent to the consumer, they are an important part of a service provider's security policy, and they are also communicated and guaranteed to the consumer as part of SLAs or other contractual agreements. Only representing both kinds of interactions inside the IT layer allows specifying PoSecCo's security policies for them. Looking at the metamodel (see Figure 2.3), the support of both functional and non-functional service interaction is visible from the 0..1 cardinality between *ITInterfaceModel* and *ITServiceModel*: functional interfaces and resources are referenced by the *ITServiceModel*, since they are known and provide recognizable added-value to the customer, whereas non-functional, management interfaces and resources are not referenced.

The *DataModel* class represents any piece of **data** which is handled by an `ITResourceModel` (attached using the *operates-on* relation) or transmitted through a *Link* (described using the *transports* relation). The *DataModel* class has a self relation *hierarchy* which can be used to describe complex datatypes which aggregate several piece of data (e.g., a person description can be composed by its name, address, etc.). Instances of the class *DataModel* can be linked via the *represents* relationship to the *BusinessInformation* in the Business Layer (see Section 2.1.1).

2.1.3 The Infrastructure layer

Figure 2.4 displays the portion of the Functional System metamodel corresponding to the Infrastructure Layer.

A central concept in the Infrastructure Layer is the *ITResource*. This class represents a wide and heterogeneous collection of software that is directly or indirectly involved in the delivery of IT services. The complex relationships between the different software components, can be described with the self relation *hierarchy*. The *provides* relation from the class *Institution* in the Business Layer (see Section 2.1.1) names the organization which is responsible to run the *ITResource*. This can be the service provider itself or one of its suppliers.

For each *ITResource*, we can model capabilities being supported by this software component. The class *Capability* denotes a security capability supported by an *ITResource*, e.g., the capability of communicating via SSL/TLS, the capability of enforcing authorization policies, or the capability of encrypting data at rest. As such, the class is of particular importance for the *policy refinement process*, as it indicates whether a given *ITResource* can act as a potential policy enforcement point, and as such can be presented to a decision maker who must select the preferred, to-be-implemented enforcement mechanism on

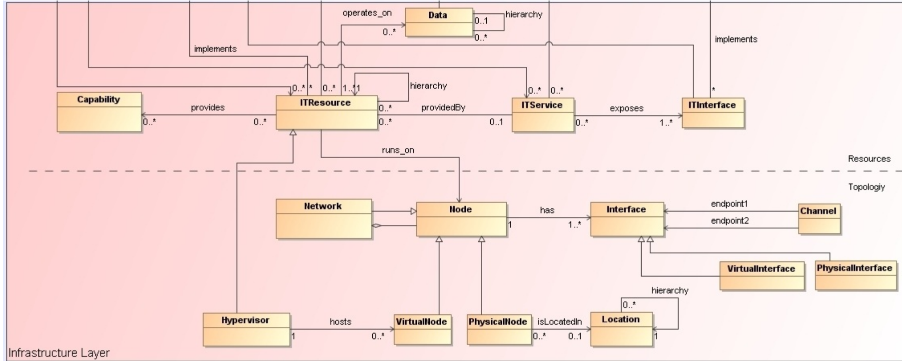


Figure 2.4: The Infrastructure layer.

the basis of different evaluation criteria. In many cases, a capability must be supported not only by one *ITResource* but by several ones in order to be considered as a alternative mechanism. The use of SSL/TLS, for instance, can only be considered as an alternative mechanism for ensuring the confidentiality of a given communication channel if all communication partners support these protocols in compatible versions, e.g., a Browser and a Web server both supporting TLS v1.0. Furthermore, the class *Capability* represents the link between the infrastructure described in the functional system model, and the security configurations that are part of the security model. Once a capability has been selected to become implemented, security configuration settings that activate and configure this capability must be created and deployed into all affected *ITResource*s. If a security manager decides, for instance, the use of TLS v1.0 for securing HTTP communication between a Web client and server, the Web server must be configured to only use TLS with a set of intended cipher suites (and to ignore the different versions of SSL), and must be equipped with an appropriate certificate. The generation of security configuration for a selected *Capability* and its deployment by means of CMS software finalizes the policy chain.

Each instance of *ITResource* is executed on a *Node*. This is represented via the relation *runs-on*. The *Node* class represents either a virtual or physical network node running some resources. Virtual nodes are described using the subclass *VirtualNode* and physical nodes using the subclass *PhysicalNode*.

A connection between two nodes is represented using the *Channel* class. This class maps any type of connection, including wired, wireless and virtual connections from/to virtual interfaces. The actual endpoints of a channel are described using the *ITInterface* class. In this way, the model is able to describe physical/virtual nodes that resort to physical/virtual network interfaces (e.g., Ethernet interface cards) to connect to each other.

A network is composed of *Node* instances. Since the *Network* is a subclass of *Node*, we can define nested networks (or subnets) and we can assign interfaces

to a network. In this way we can, for example, define the "internet" network and its interfaces, abstracting its internal complexity.

With the class *ITResource*, we can describe all deployed and configured software components that belong to an IT service implementation for a single customer (*ITService*) using the relation *providedBy*. The *ITResource* represents the implementation of an abstract *ITResourceModel* from the IT Layer (see Section 2.1.2), which is attached to it by the relation *implements*.

An instance of the class *ITService* describes a service implemented for one customer modeled by the *uses* relation to the *Institution*. It exposes *ITInterfaces*, linked by the homonymous relation. The *ITInterface* class represents an implementation of the *ITInterfaceModel* class in the IT Layer.

The class *Data* provides details about the representation of data, allowing the aggregation of complex data structures via the self relation *hierarchy*. It implements the corresponding class *DataModel* of the IT layer, which in turn is linked to *ITResourceModel* and to *Link*. Information about the storage location and type is required to configure the enforcement of various policies, e.g., the enforcement of AC policies by means of database authorizations, or the enforcement of data protection policies by means of WS-Security.

2.2 Security metamodel

According to the abstraction levels defined in the Functional System metamodel also the policy related concept will be organized according to the same levels and related to the element of the landscape metamodel that they involve using the following formats:

- *Business policy* describes the security requirements, which the service provider must guarantee, in a declarative human readable format;

- *IT policy* maps the required behavior of the IT services, ignoring topological details. It uses a formal model since it must be automatically processed by the refinement engine;

- *Security configurations* are imperative policies detailing the implementation of an IT security policy by an enforcement mechanism, e.g., firewall, routers or authentication servers. They are topologically aware as they consider all the nodes of the network connecting the service components.

2.2.1 Business Security metamodel

The Business Security metamodel is illustrated in Figure 2.5.

Security Requirements are the central component of this model. Each requirement can be composed of sub-requirements through the relation "realizes" allowing to build an arbitrary dependency-tree. Any requirement may be derived from a *Source* being either another requirement, a law, a policy or some other origin (e.g., a service level agreement with business partner). Taking the

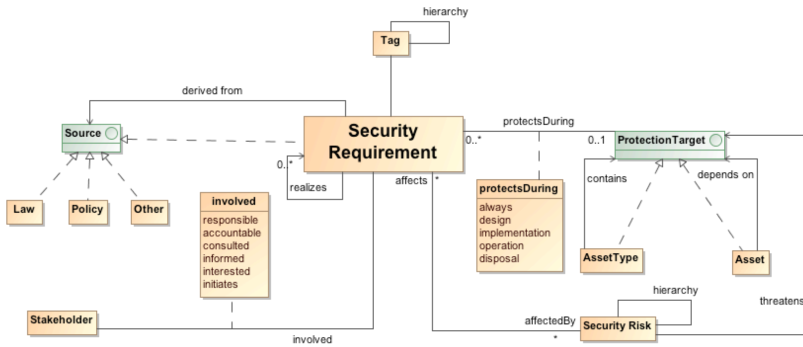


Figure 2.5: Business Security metamodel.

derived-from-relation into account it might seem that the realization-relation would not be needed at all. But that's only so at first glance. Many requirements especially at a more technical level are depending on various other requirements, they might not be directly derived from. For instance the requirement "Ensure availability of customer database" additionally depends on "Provide timely backups for customer database" which itself is derived from the requirement "Implement a data backup plan". With both relations present, it is possible to develop a very expressive model not only considering hierarchical dependencies of requirements but also allowing connections between requirements with regard to contents. Another difference between those two connections is also evident: derived requirements might need to be revalidated if the superordinate requirement is changed, i.e., loosening the superordinate requirement may make one or more subordinates unneeded.

Further every requirement is related to one or more *Stakeholders* holding various functions. This relation does not only consider certain kinds of responsibilities but additionally allows interested parties (e.g., customers, media, competitors) to be related with certain requirements.

Tags are used to provide another way of organizing requirements apart from the relations "realizes" and "derived from". By tagging requirements it is possible to build up any number of orthogonal arrangements. With this feature at hand it becomes possible to completely adopt the structure of arbitrary standards without interfering with the structure needed to actually manage security requirements. In a special sense, tags can provide shortcuts to a set of requirements that otherwise would not be directly related via protection targets or some inter-requirement relations. The great benefit from tagging is the way in which this technique allows users to successively enhance requirement categorizations through addition of new tags without the need to change major parts of the model.

The metamodel includes *Security Risks* but does not provide any additional concepts related to risk management. This approach has been chosen to keep the model as simple as possible while still being able to incorporate threat scenarios

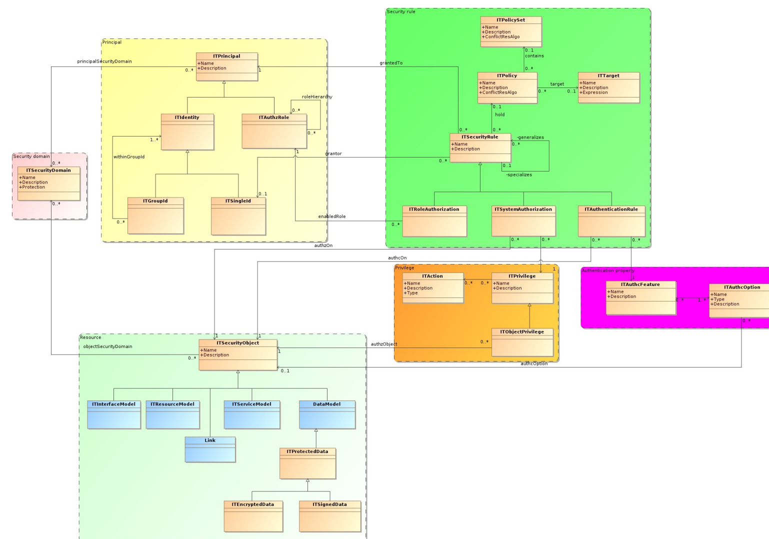


Figure 2.6: IT level Security metamodel.

often considered together with requirements. However, threats are not the main focus of this meta model and are therefore treated only as additional source for classification of security requirements (e.g., find requirements addressing human errors or technical shortcomings).

2.2.2 IT layer Security metamodel

The IT layer Security meta-model supports the representation of authentication and access control properties. Figure 2.6 shows a graphical representation of the complete model. The model is structured in parts, where each part contains all the entities describing one aspect of the model.

The IT metamodel consists of six related blocks, each focused on a specific aspect of the security policy.

- the *Principal* meta-model is used to describe the organization of identities, how users are structured into groups and how roles are assigned to users;
- the *Security rule* meta-model describes a taxonomy of authorization and authentication rules, together with some basic properties of the rules themselves;
- the *Authorization privilege* meta-model is used to describe the structure of privileges that are specified for system authorizations;
- the *Authentication property* meta-model is used to describe the structure of properties that are associated with authentication rules;

- the *Resource* meta-model describes the structure of the static, dynamic and communication resources that are associated with the privileges in the authorizations and with the authentication properties.

- the *Security domain* meta-model contains the entity that denotes the concept of security domain, supporting the realization of policies.

An important feature that is not explicited in the graphical representation of the metamodel is the possibility for each of the entities in the metamodel to be involved in a many-to-many relationship with the *Security Requirement* entity at the Business level. This relationship is critical for the explicit representation of the connections among the representations at the different levels. Similar relationships will be introduced in the Infrastructure metamodel to describe which are the connections between the IT policy and its realization at the Infrastructure level.

In the remainder of this section we briefly describe each class in each model block, as well as the fundamental relationships between them.

Furthermore, the metamodel is enriched by the support for ontologies. A significant advantage offered by ontologies is the possibility of checking the consistency of the model instances, going well beyond what can be offered by classical modeling tools. Ontologies support a variety of functions, using several approaches.

The proposed model will be compared in Appendix A with previous solutions. The analysis shows that the proposed model has a greater expressive power than other solutions. There are a few aspects that characterize other models, like the flexible representation of conditions of XACML, which have been omitted due to the obstacles they would have imposed in the realization of the harmonization services.

Principal metamodel The classes in this model are ITPrincipal, ITIdentity, ITGroupId, ITSingleId, ITAuthzRole. The goal of the model is to describe all the elements that can appear as beneficiaries of authorizations or be involved in an authentication rule. A variety of IT concepts can be represented by this portion of the meta-model. The beneficiary can be a real person, or it can be an account on a system, or it can be a group of identities, or it can be a role. This variety of concepts matches with the variety of configurations that are observed in real IT systems. Figure 2.7 shows the structure of this portion of the IT level meta-model.

ITPrincipal ITPrincipal is the root of the hierarchy of entities that support the representations of principals. It is an abstraction of whoever person or whichever entity can be the beneficiary of an authorization or authentication rule; for negative authorizations, the principal is the party who is forbidden from receiving a specific privilege. It has two subclasses: ITIdentity and ITAuthzRole. It is an abstract class, then no instance is associated with it that is not associated with one of the children entities;

ITIdentity ITIdentity is the root of a hierarchical user classification. In general, a principal can correspond to any user or to any specific organizational unit that corresponds to a group of identities. Properties that describe the configuration of authentication services will mostly be associated with this class. This association will specify how possession of an ITIdentity can be proved. For instance, we may have identities that are proved by providing a correct password. In other cases, identities can be proved by presenting a certificate and successfully passing a challenge/response step where access to the private key associated with a certificate is verified;

ITGroupId An instance of ITGroupId is a specialization of the ITIdentity concept and denotes (directly or indirectly) a collection of lower level identities. Groups are organized in a taxonomy via the *contains* relationship to instances of ITIdentity. This means that a group can contain one or more groups as well as one or more specific identities;

ITSingleId Each instance of ITSingleId corresponds to a single identity in the system. An ITSingleId could have been considered as a particular kind of ITGroupId, containing a single identity. It was preferred to model the single identities as a separate class because modern access control models permit the distinction between single and group identities. This choice offers a more precise description and an easier mapping to the configuration. This is also consistent with security best practices, which force the use of explicit user identifiers always leading to a specific person, in order to be able to assign a concrete responsibility for every action on the system. ITSingleId instances can also represent legal entities;

ITAuthzRole The ITAuthzRole entity offers a variety of interpretations, adaptable to the specific features of the access control models implemented in the functional system metamodel. ITAuthzRole instances can correspond to “organizational roles”, to “collections of privileges”, and to “accounts”.

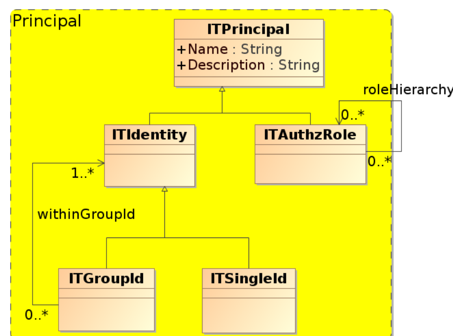


Figure 2.7: Principal Meta-model.

There is no clear line of separation between these different interpretations in the scenarios is going to consider, thus it is considered appropriate to keep a single IT level concept for the representation of this variety of situations. The critical feature that characterizes the model is that roles are separated from the representation of the real identities and the rigid organization described under the identity hierarchy. We can shortly analyze the interpretations considered above.

- *Organizational role*: this is the use of the ITAuthzRole entity that better corresponds to the *BusinessRole* entity appearing at the Business level in the Functional system metamodel. The representation provided at the IT level is more concrete and specifies the collection of privileges that have to be available to the members of an organization that have the responsibility to execute a specific function. Note that the model does not support a direct connection between the BusinessRole and ITAuthzRole entities. The connection is mediated by the use of a *Security Requirement* that specifies the security requirements associated with a specific role. The assumption here is that typically the roles will have to be "enacted" before they can be used, and the use of the concept at run-time will be strictly connected with the idea of a "session". Systems may put restrictions on the number of roles that can be used at the same time (e.g., in SQL there is the restriction that only one role at a time can be active in a session, and each SQL statement has to be executed within the context of a session). The concept of session is crucial for the run-time evaluation of the Access Control restrictions, but this is not required for PoSecCo, which deals only with the static configuration of the policy and does not consider its enforcement.
- *Collection of privileges*: An ITAuthzRole instance is the container for a set of privileges that have to be assigned to an active party in the system. This is similar to the organizational role, except that no restriction is imposed on the dynamics, and potentially many collections of privileges can be given to a user and exercised at the same time. Historically, the concept of role in RBAC models was associated with the first interpretation, but currently the emphasis has shifted to the "security management" aspects, rather than the run-time aspects, and the interpretation of a role as a collection of privileges is probably today the most common one. At the IT level in PoSecCo there is no need to distinguish between the two interpretations. This interpretation is expected to be the basis for the realization of harmonization services that support the specification of separation of duty constraints.
- *SystemAccount*: An instance of ITAuthzRole can represent an account available on a system (Application/DBMS/OS). The advantage of putting this concept within ITAuthzRole is the possibility to

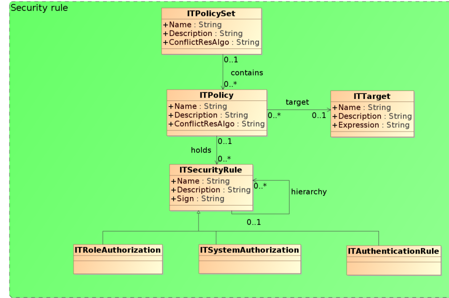


Figure 2.8: Security rule Meta-model.

manage as authorizations the assignment to identities of the privilege of activating a specific account in a system. This is a specific aspect that has to be considered in PoSecCo, because the security policy has to look at a complex infrastructure. Common access control models refer to a single component and do not have this need, as they essentially combine the concept of user with the concept of account on the system. This is an aspect that supports the definition of models where authentication is managed by a specific node in the network, and all the other nodes in the network belonging to the same security domain "trust" the decisions of the authentication servers. The verification that a user asking to access a system has successfully completed the authentication can be realized in a number of ways, e.g., with tokens or signed assertions. The designer can choose to consider the specific solution used to communicate the positive outcome of authentication as an internal aspect;

Security rule metamodel The security rule meta-model contains classes ITPolicySet, ITPolicy, ITTarget, ITSecurityRule,ITRoleAuthorization, ITSystemAuthorization, and ITAuthenticationRule. Figure 2.8 shows the structure of the portion of the IT level meta-model describing security rules and policies.

ITPolicySet A policy set represents a collection of ITPolicy elements. The motivation for the introduction of the policy set is to offer a level of aggregation of policies that supports the integrated representation of the business requirements. This approach corresponds to the solution used in XACML, that offers these two levels of aggregation. A property that characterizes the ITPolicySet is the conflict resolution algorithm that has to be used to manage conflicts in the policies;

ITPolicy A policy consists of a set of authorizations. It is associated with a target that specifies restrictions valid over all the authorizations in the policy. This entity supports the organization of security associations in sets that contribute to the description of a specific policy goal. This

entity also permits the specification of the conflict resolution algorithm used, since the policy can contain both positive and negative rules;

ITTarget The target permits to specify for each policy a set of restrictions that have to be applied to all the authorizations in the policy. The target can specify restriction over the principal and the privilege/resource managed by the authorizations in the policy;

ITSecurityRule The ITSecurityRule entity is an abstract class that permits the representation of rules that specify the structure of authorization and authentication. Instances of the entity apply to principals and are possibly tracked to be granted by a specific person. The choice of the ITSingleId as the endpoint of the relationship is consistent with the security principle that the realization of auditing requires to keep track of the specific identity of the person behind each administrative operation; the explicit representation of the user responsible for a security rule is supported by some access control models, like in relational DBMSs.

Three different specializations of ITSecurityRule are foreseen: ITRoleAuthorization, ITSystemAuthorization, and ITAuthenticationCondition. ITSecurityRules have a sign, so there is a total partitioning in Positive ITSecurityRules and Negative ITSecurityRules. The choice of the conflict resolution algorithm occurs at the level of policies and policy sets. In general, there is the interesting option to enforce a model where negative authorizations (prohibitions) can only be defined as exceptions to previously defined positive authorizations, but this property requires support in the harmonization phase.

Optionally an authorization can be marked with a "grant option", which means that the assignee can assign the same authorization to another user. The SQL access control model offers this capability; in an environment describing the security management of a complex infrastructure, there are significant opportunities for the use of this feature;

ITRoleAuthorization ITRoleAuthorizations represent the authorization to enable a role. An ITRoleAuthorization permits to specify which principals are associated with a role. Negative ITRoleAuthorizations require a particular interpretation, supported by the ontology, possibly with adaptations to the specific scenario. For instance, the support for separation of duty can rely on the introduction of negative ITRoleAuthorizations that specify the incompatibility between two different roles. It is then the responsibility of the harmonization phase to verify the consistentcy (for static separation of duty) of the policy wrt the specified constraint. The idea of having explicit role authorizations was presented in the original proposals for RBAC models. The XACML profile for RBAC uses the same approach, with two suggested kinds of XACML rules. Roles are disjoint from identities and they cannot belong to groups. The hierarchy among roles can be represented by a set of authorizations, but it should be

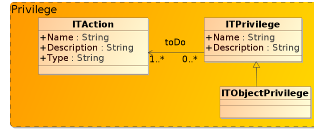


Figure 2.9: Privilege Meta-model.

clear that the containment relationship has a semantics that differs from that characterizing the definition of ITRoleAuthorizations;

ITSystemAuthorization ITSystemAuthorizations correspond to the classical authorizations/rules of most access control models. Authorizations are often considered as <subject,resource,action> triples. The model proposed relies on a different approach, where an authorization associates a subject (the principal) with a privilege. The privilege is described in the next section. This choice increases the flexibility, because it can be immediately used to describe resourceless privileges, like the “auditing”, “shutdown”, or “backup” privileges that appear in operating systems and DBMSs access control model, not associated with a specific resource. The sign is associated with the authorization and no hybrid authorization is possible, that is, a system authorization cannot mix positive and negative privileges;

ITAuthenticationRule The ITAuthenticationRule entity permits to specify the configuration of authentication for a given principal and with respect to the access to a given resource. The authentication rule will typically have as target a resource that natively supports authentication services (e.g., an ITResourceModel that corresponds to nodes with direct access to operating systems). Authentication rules can also specify restrictions on the way authentication has to be supported when accessing the communication channel (ITLink entity) specified as a target. The choice of presenting at the same level is not common in the representation of access control models, where typically the emphasis is on the precise description of authorizations, but it is consistent with the architecture of the CIM model and other solutions that focus on the description of the concrete security configuration of an information system;

Privilege metamodel The meta-model consists of entities ITPrivilege and ITAction. Figure 2.9 shows the structure of the portion of the IT level meta-model describing the two entities.

ITPrivilege An ITPrivilege instance represents the right of performing some action. In access control models there is commonly a distinction between privileges, which are resourceless, and access rights, which appear in the DACL of a resource. In these models, a privilege with no resource is interpreted as a generic privilege and indicates some system level action.

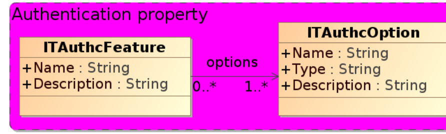


Figure 2.10: Authentication property Meta-model.

For example “log access” can be granted in those models without the reference to a specific resource. The PoSecCo metamodel assumes that all the privileges, associated with a resource or not, correspond to instances of the ITPrivilege entity;

ITObjectPrivilege This entity is a specialization of ITPrivilege and it denotes all the privileges associated with a specific resource. Classical authorizations are often denoted as the elements of an ACL. These will correspond to instances of this entity, where the object associated with the ACL is the object which each instance refers to;

ITAction An ITAction instance represents a generic business or technical action. Internally, the action may present a set of privileges (as in common ACLs). The structure of the action is strictly related with the characteristics of the underlying access control system. At the IT level it is possible to use rather descriptive actions, that will be made concrete in the refinement when producing the configuration. Dependencies can be modeled among the privileges (e.g., a rule may specify that an IT level “update” privilege requires an IT level “read” privilege), with the use of ontologies;

Authentication property metamodel The meta-model consists of entities ITAuthcFeature and ITAuthcOption. Figure 2.10 shows the structure of the portion of the IT level meta-model describing the two entities.

ITAuthcFeature The entity permits the description of properties that the authentication service has to satisfy. By way of the authentication rule the property can be associated with a subject. In many cases it is expected that the principal will be a large group of users, since in most cases the authentication properties do not need very fine granularity and general restrictions applying to categories of users are normally sufficient. The authentication option may specify additional details about certificates, like, e.g., where is the repository of the trusted root authorities;

ITAuthcOption The entity ITAuthcOption permits to specify the configuration options of the authentication features, illustrating the admissible values for each parameter. The definition of the values is part of the ontology and offers a level of flexibility, in order to adapt to the specific options that are needed in the different landscapes where the model will be used. The relationship with an ITSecurityObject permits to specify

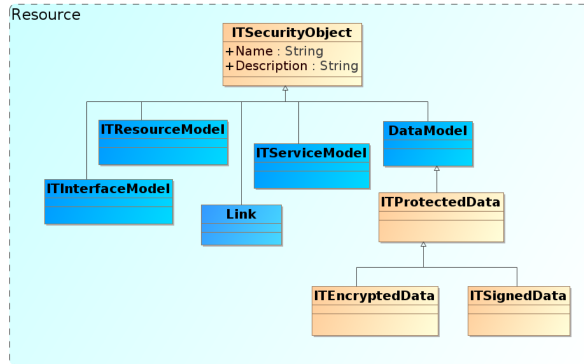


Figure 2.11: Resource Meta-model.

references to elements of the landscape that are needed for the realization of the authentication service. For instance, an LDAP server can be used to store the credentials of the users accessing the application. The LDAP server will be referenced by an instance of ITAuthcOption that details the structure of the authentication rule for the application.

An advantage of the introduction of this entity rather than putting all this information within ITAuthcFeature is that the structure for authentications becomes symmetric to the structure used for system authorizations. It also offers greater expressive power and more flexibility than what would be offered by a single entity.

Resource meta-model The Resource meta-model consists of entities ITSecurityObject, ITInterfaceModel, ITResourceModel, Link, ITServiceModel, DataModel, ITProtectedData, ITEncryptedData, and ITSignedData. Figure 2.11 shows the structure of the portion of the IT level metamodel describing the structure of resources. Entities imported from the Functional System Model have a different color.

ITSecurityObject The class is an abstraction of all the IT level entities in the functional meta-model that can be the target of an authorization. The PoSecCo environment sees a large variety of resources to manage. The advantage of the introduction of the abstract class is that a single regular format can be used for the representation of access control privileges over different kinds of resource. The main distinction within resources is between static resources, active resources, and network links. The first two kinds match the classification of access control models, which are typically distinguished in "classical" access control models (e.g., OS, DBMS) and access control models for services and object-oriented systems (e.g., Java, Application servers), which describe privileges for software modules. This separation is justified by the separation between services and data that

appears in the IT level Functional System metamodel. The representation as a target of authorizations of network links is one of the crucial innovations of PoSecCo;

ITInterfaceModel The entity is imported in the IT level Security metamodel from the IT level Functional System metamodel. It describes any active resource that can be the target of an authorization. This entity is used in the representation of authorizations specifically designed for a Service Oriented Architecture. If we look at a classical OS, a program is stored in a file, and the user can be authorized to execute it, with two alternative modes in terms of privileges (executing the process in the user environment and exploiting the privileges associated with it, or using a special "setuid" mode that applies in the execution of the service the privileges of the owner of the program). For these scenarios, there is the possibility to describe the authorizations using a specific privilege over a static resource. For the services considered in a SOA environment, both with a WS-* or RESTful invocation paradigm, the use of a different kind of resource appears appropriate, as it better captures the nature of the protected resource and the required support;

ITResourceModel This entity is also imported from the IT level Functional System metamodel. The entity describes at the IT level the nodes and software modules used to implement a service. Authorization and authentication rules that reach this entity will produce in the refinement a number of concrete authorization and authentication restrictions that will be supported by the access control services offered by the element that will represent in the landscape the ITResourceModel instance. If the IT Policy mostly consists of authorization and authentication rules that derive from an analysis of what the system will be able to directly support, it can be expected that most of the rules will use ITResourceModel instances as a target;

Link This entity is imported from the IT level Functional System metamodel. The Link entity is reported in the metamodel to specify authentication and authorization rules that involve a network channel. A Web application may have requirements that specify that the network communication between the clients and the application has to use a specific set of protocols (e.g., SSL, with mutual authentication based on X.509 certificates and RSA-1024 signatures), with a set of protocols that have to be excluded. These requirements correspond to positive and negative authentication rules that use as resource the instances of the Link entity that represent the connections used by the clients to access the application;

ITServiceModel This is also an entity that appears in the IT level Functional System metamodel. The ITServiceModel entity provides an abstract representation of a collection of fine-grained services that are represented by the ITInterfaceModel entity. This resource can then be used as a target

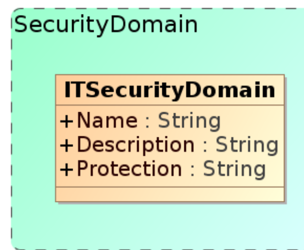


Figure 2.12: Security domain Meta-model.

for authorization and authentication rules that specify permissions and restrictions on the access to the collection of service interfaces characterizing an application;

DataModel This entity is also imported from the IT level Functional System metamodel. It describes a variety of static resources, like a file or a directory in a file system, an attribute or a view or a table in a relational database, or a specific piece of information in a business application;

ITProtectedData This specialization of Data has the goal of managing protection requirements that have been defined at the Business level. The class is specialized in the two classes described below.The hierarchy is not exclusive, as the same piece of data can be both the subject of confidentiality and integrity requirements;

ITEncryptedData This further specialization of ITProtectedData is introduced to specify the encryption properties of the information that is subject to encryption requirements, when stored or in transit;

ITSignedData This specialization of ITProtectedData permits to specify the signature details for information that has to satisfy integrity requirements;

Security domain meta-model The metamodel consists of a single entity, ITSecurityDomain. Figure 2.12 shows the structure of the portion of the IT level metamodel describing the single entity.

ITSecurityDomain The entity has the goal of describing at the IT level the existence of separate security domains. Security domains have a relationship with principals and with resources. Security domains may characterize principals and make explicit their location or the organization the principal belongs to. A security domain for resources may organize resources depending on the application domain. In an application, there may be customers that require that their data are physically segregated from the data of other applications. The definition of these constraints depends on the availability of a special representation of the security domain;

2.2.3 Configuration metamodel

The Configuration metamodel is on one hand an extension of the Functional System metamodel and details capabilities of software components together with their configurations in form of settings or rules. On the other hand the Configuration metamodel is the lowest level of the *policy chain* which links business requirements on the highest level over IT policies down to configurations of single components. The configuration metamodel therefore represents the link between the Infrastructure layer of the Functional System metamodel and the policies of the policy chain.

The development of the Configuration metamodel has been based on existing standards, like CIM and CIM Policy. The Configuration metamodel combines the approach of configurations settings in CIM with the rule based configurations from the CIM Policy Model, as such allowing the modeling of a broader range of configurations from arbitrary hierarchies of key/value pairs to complex rule sets with Boolean conditions and different actions.

Similar to the Common Information Model (CIM), in the context of this document, a capability denotes any kind of common functionality that can be provided by a software component. We focus on an extensible set of security capabilities that fits the selected security domains of the PoSecCo project: *Access control* and *Communication protection*. A security capability describes security abilities and/or potential for use (e.g., the capability of communicating via SSL/TLS, the capability of encrypting data or the capability of enforcing authentication and authorization policies). The concept of a security capability tied to an IT Resource is crucial and has a particular importance for the policy refinement process, as it indicates whether a given IT Resource can act as a potential enforcement mechanism to satisfy a given IT policy. Beside the security capabilities, we define also a set of supporting capabilities, like Routing or Logging, which serve as input for the policy refinement process.

On the basis of a general Configuration metamodel, we could specify several configuration specializations for the following capabilities fitting the selected security domains:

- *Data Protection*;
- *Authentication*;
- *Authorization*;
- *Filtering*.

In the following sections we will provide a brief introduction about the configuration specializations of (a) Data Protection, (b) Authentication and (c) Authorization. We refer to [14] for a complete explanation about Filtering configuration.

Figure 2.13 sketches the high-level elements of the Configuration metamodel.

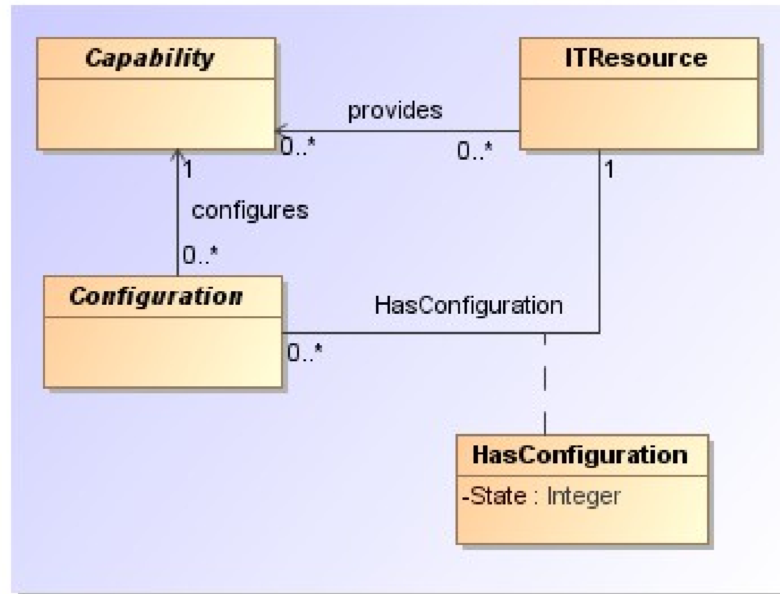


Figure 2.13: High-level of the Configuration metamodel.

ITResource An ITResource, known as the central concept in the Infrastructure Layer (see Section 2.1.3 represents a piece of software that is directly or indirectly involved in the delivery of IT services. For each ITResource, we can assign the capabilities being supported by this software component with the help of the relation provides.

Each ITResource can have zero or more Configurations which are assigned to exactly one capability. The assignment of configurations to an ITResource is done by means of the relation *HasConfigurations*. The association class with the same name allows the qualification of different types of configurations for an ITResource instance in the model. It can be used to represent the current configuration of an ITResource and the golden configuration as a result of the policy refinement process, at the same time.

Capability A Capability denotes any kind of (security) functionality that can be provided, e.g., filtering, authorization, authentication). In PoSecCo, mainly common security functionalities will be considered, i.e., capabilities that are typically supported by a class of software products. A further description of the Capability class and its subclass hierarchy can be found in Section 3.

Configuration A Configuration corresponds to abstract configuration settings, which are independent from a given vendor or product. The abstract class Configuration can be subclassed by configurations dedicated to deliver the

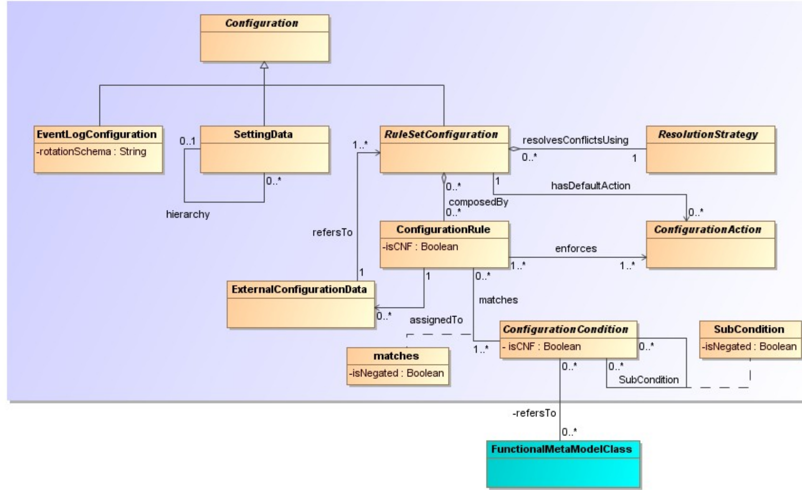


Figure 2.14: Configuration Specializations.

different capabilities, e.g., class AuthorizationConfiguration (shown in Figure 9 Configuration Hierarchy) to describe settings for access control.

To ensure the identification of the affected infrastructure element for a given configuration, each Configuration is assigned by the relation *HasConfiguration* to exactly one ITResource.

We distinguish three kinds of configuration, which are all product- and vendor-independent (see Figure 2.14)

- EventLogConfiguration;
- SettingData;
- RuleSetConfiguration.

The *EventLogConfiguration* provides information where the event logs and traces are located and describe the format of the event logs and their semantics. This kind of configuration reflects normally the current landscape configuration and is used to access logs for process mining or to collect audit evidences.

SettingData describes configurations as key/value pairs in arbitrary hierarchical structures. In general this representation can be used to specify to PEPs where it is located the PDP in charge for making the decisions it has to enforce.

The *RuleSetConfiguration*s are the preferred type of configuration within PoSecCo and it will be used to assign a configuration (i.e., a policy) to a PDP. This kind of configurations is the expected outcome of the policy refinement process, that is, a rule set is the representation of a policy at landscape configuration layer of the Functional System metamodel, the lowest abstraction layer within PoSecCo. Each RuleSetConfiguration consists of a set of *ConfigurationRule*s.

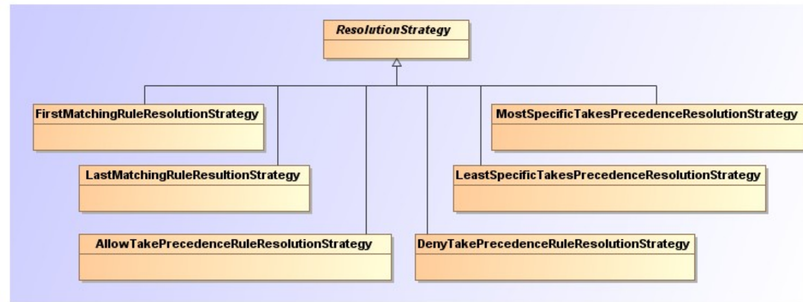


Figure 2.15: Hierarchy of ResolutionStrategy.

A ConfigurationRule is enforced by *ConfigurationActions* (association enforces) and can use a set of *ConfigurationConditions*. A ConfigurationCondition is a Boolean predicate, which allows the expression of complex conditions using AND- and OR-connections. Several ConfigurationConditions can be assigned to the ConfigurationRule via the association matches. More complex configuration rules can be modeled using the self relation *SubCondition* of the class ConfigurationCondition. The association class with the same name has the attribute *isNegated* that indicates when a subcondition is used negated or unnegated.

A ConfigurationCondition can refer to elements of the infrastructure layer of the Functional System model, like the ITInterface or Data. This is modeled by the association *refersTo* between ConfigurationCondition and the placeholder *FunctionalMetaModelClass*. The refersTo association permits to identify, which element in the Functional System metamodel is actually affected by a rule.

Practically, two additional cases need to be considered when describing a policy using a set of rules: what happens if no rules apply and what to do when more than one rule applies. The latter is often named *policy conflict*.

With the association *HasDefaultAction* a default ConfigurationAction is linked to the RuleSetConfiguration in order to specify what to do when no rule can be applied. If more than one ConfigurationRule can be applied, the ResolutionStrategy assigned to the RuleSetConfigurations via the association *resolvesConflictsUsing* is used to solve the conflict. We predefine the most common resolution strategies (see Figure 2.15). The presented specializations of the class ResolutionStrategy can be further refined and extended when needed.

2.3 Model specialization

Once the metamodel is defined at every level, it is necessary to refine it to an "actual" model, which can be used to describe the services to be handled. We decided to impose some constraints to the process that defines a model starting from the metamodel in order to avoid that changing or extending the landscape models will affect the refinement process. These constraints will assure that

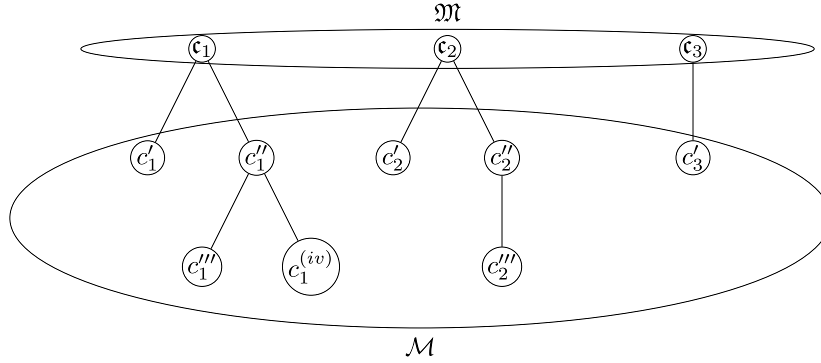


Figure 2.16: Graphical representation of metamodel and model classes.

extending the models will not impose changes in the refinement algorithms and in their implementations in the PoSecCo prototypes.

First of all, all the classes of the models will be subclasses of metamodel classes. Formally, given the set of classes of the metamodel:

$$\mathfrak{C} = \{\mathfrak{c}_1, \dots, \mathfrak{c}_t\}$$

any class of the model will be a subclass of exactly one element of $\mathfrak{C}$, therefore the set of classes of the model will be defined as:

$$\mathcal{C} = \{c_1', \dots, c_1^{(n_1)}, c_2, \dots, c_2^{(n_2)}, \dots\}$$

where $c_i^{(n)}$ is subclass of $\mathfrak{c}_i$. Figure 2.16 depicts this concept. For example, by subclassing the class `node`, which represents a generic network node, in $\mathfrak{M}$, we can define the class `firewall` in $\mathcal{M}$.

Then, associations between classes in the models are possible only if a relationship exists between the superclasses in the metamodel. Formally, given the set of relationships of the metamodel, defined as:

$$\mathfrak{R} = \{(\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{r}_1), \dots, (\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{r}_n), (\mathfrak{c}_2, \mathfrak{c}_4, \mathfrak{r}_3), \dots\}$$

where $(\mathfrak{c}_i, \mathfrak{c}_j, \mathfrak{r}_n)$ is a relationship between $\mathfrak{c}_i$ and $\mathfrak{c}_j$, a relationship $(c_i^{(k)}, c_j^{(l)}, r_n^{(m)})$ exists in the set of relationships of the model

$$\mathcal{R} = \{(c_1', c_2'', r_1'), \dots, (c_i^{(k)}, c_j^{(l)}, r_n^{(m)})\}$$

if and only if an element $(\mathfrak{c}_i, \mathfrak{c}_j, \mathfrak{r}_n)$ exists in $\mathfrak{R}$. where r_n^m is a specialization of the relationship $\mathfrak{r}_n$. This situation is illustrated in Figure 2.17.

Model specialization for File System

An *OperatingSystem* can host *FileSystem*s that consist of *LogicalFile*s. The homonymous relationship hosts is hereby a refinement of the more general self relationship *hierarchy* of class *ITResource*.

A *FileSystem* can be further specialized indicating:

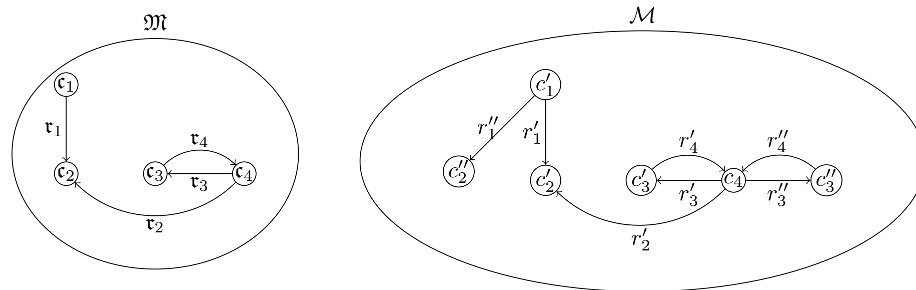


Figure 2.17: Graphical representation of metamodel and model relationships.

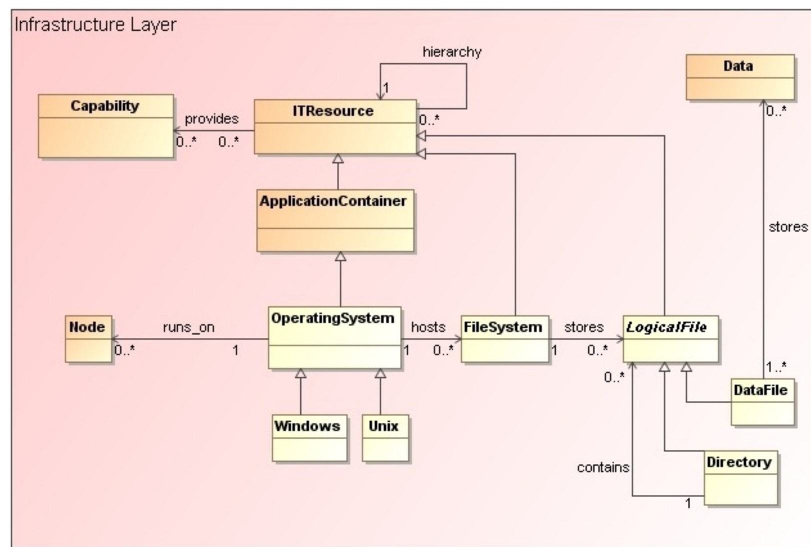


Figure 2.18: Example Model Operating System

- its FileSystemType, e.g. ”NTFS” or ”S5”;
- its Root, the path name or other information defining the root of the FileSystem;
- its EncryptionMethod, a string indicating the algorithm or tool used to encrypt the FileSystem.

A *LogicalFile* is a named collection of data or represents a directory. It is located within the context of a FileSystem described by the relationship *stores*, which subclasses the self relationship of class *ITResource*. The class *LogicalFile* is abstract and can be subclassed by *DataFile* and *Directory*.

A *LogicalFile* can be further specialized indicating:

- its Name, a unique identifier (such as a full path name) within the *FileSystem* it belongs to;
- its parameters, like "Readable" or "Writable" indicating that the file can be read or written;
- its EncryptionMethod, a free form string indicating the algorithm or tool used to encrypt it.

A *Directory* is a type of file that logically groups the directories and files 'contained' in it, and provides path information for the grouped files. The grouping of *LogicalFiles* is described by the relationship *contains*, also a specialization of the self relationship *hierarchy* of *ITResource*.

A *DataFile* is a type of *LogicalFile* that is a named collection of data. It can be linked to the *Data* by the relationship stores, a specialization of the *operates_on* relationship between *ITResource* and *Data*. This relationship links the concept of *Data* to the physical placement of this information in a *FileSystem* on a piece of hardware.

We can assign the following capabilities (subclasses of class *Capability*) to the *FileSystem*, *DataFile* and Directory:

- *Authorization*;
- *DataProtection*.

In Figure 2.18, an example system is modeled. On a PC that runs Windows 7 as operating system and on the NTFS file system on drive D: a data file is stored that contains credit card information. Appendix B introduces several examples about model specialization.

3

Policy analysis and harmonization

A crucial advantage of the model-driven approach, i.e., the specification of security requirements at an abstract level, with the subsequent refinement of the abstract model toward a concrete implementation, is the possibility of an early identification of anomalies in the security policy. Security policies in real system often exhibit conflicts, i.e., inconsistencies in the policy that can lead to an incorrect realization of the security requirements, and redundancies, i.e., elements of the policy that are dominated by other elements, increasing the cost of security management without providing benefits to the users or applications.

Although security administrators can handle manually simple access control policies (e.g., ACLs), the manual approach cannot scale well with bigger policies. For instance, maintaining and modifying complex ACLs is not an easy task, because it may be difficult to identify all the possible side effects of a change in the policy. Another problem of bigger ACLs is that they usually contain redundancies [82], which introduce further complexity in the policies, which leads to higher management costs and less efficient behaviours of access control mechanisms. Removing redundancies can also improve the efficiency of the access control mechanism [46].

Although some current access control languages, e.g. XACML, implements techniques that automatically solve conflicts between rules in the policy at run-time using a particular criterion, anomalies detection at compile-time can provide a valuable help to security administrators, because it is difficult for them to clearly understand what the results of the run-time conflict resolution algorithm will be, at least in complex policies, and this can lead to unexpected problems and misconfigurations. Due to these needs the identification of conflicts and redundancies in policies has been the subject of a significant amount of research in the last years.

Specifically, Semantic Web and ontology management technology [53] today offers a variety of interesting solutions. In the recent past, the application of logical models to the analysis of security policies required to use logical tools that had no clear integration with common information system tools; security administrators in most cases administrators were unfamiliar to them and they perceived the risk of limited scalability [100]. Modern Semantic Web tools are instead well integrated with existing XML and Web technology, are used in many scenarios to efficiently obtain solutions for several optimization problems, and for this reason are often already familiar to administrators. The familiarity of computer science practitioners with Semantic Web tools is likely going to increase, as the diffusion of these tools is currently increasing [41].

Semantic Web technology has made available an extensive collection of tools. Several approaches can be adopted, relying on different tools, with different profiles in terms of abstractness and efficiency. In general, tools and approaches are available that offer a simple and direct representation of the policy, but may present scalability problems when applied to complex analysis tasks, and other tools may require greater effort in the representation of the problem, but offer scalability and support the analysis of complex properties in large-scale scenarios.

This section will present a variety of approaches that will confirm that the use of Semantic Web technology for policy analysis can bring great advantages in the identification and resolution of conflicts.

3.1 State of the Art

The following Section reviews some existing work of interest in policy harmonization and reasoning. The state of the art in policy harmonization checking is presented, illustrating the research performed in specification, management and analysis of access control policies with existing reasoning techniques. Concerning specification, different approaches for representation of positive and negative authorizations and obligations as well as static and dynamic segregation of duties constrains are included.

3.1.1 Policy Harmonization checking

Kolovski [66] presents an in-depth analysis of several works on the specification, management and analysis of access control policies by means of logical formalisms.

The main purpose of policy analysis includes verification of conflicts, modality conflicts, but also "controlled system or application" specific conflicts such as separation of duty. Therefore, a complete approach to policy harmonization must cover both.

The survey done by Kolovski in [66] covers both language proposals that have formal semantics and therefore that provide algorithms for policy analysis out of the box, and formalizations of already existing policy languages, such

as WS-Policy, XACML, XrML and ODRL, that provide a formal semantics and analysis services previously unavailable for the particular language. In 2.1.1 different ways are summarized to exploit semantic web reasoning tools for policy analysis of the main used policy models, which are RBAC or ABAC based. In 2.1.2 other logical formalisms are presented to model and analyze policies, keeping in mind that policies representations must be as expressive as possible to fit the global requirements of the controlled system.

Semantic web reasoning

The idea of adapting security models to take advantage of consolidated and efficient reasoning from semantic web tools is well studied in literature. However, semantic web technologies are often used one at a time; our approach, instead, is based on the consideration that different technologies (DL reasoning, rule engines and *SPARQL* querying) should be used in an organic and organized framework.

Finin et al. [40] use OWL [80] to formalise Role Based Access Control (RBAC) [101]. They propose two approaches, in the first one they represent roles as classes and the role hierarchy is expressed by means of subclassing, while in the second one they represent roles as values and the role hierarchy is expressed by means of the superRole property. They also express static and dynamic separation of duty constraints, in the first approach by means of the owl:disjointWith property, while in the second approach they express the constraints by means of N3 rules [13]. Their approach can be used also to express attribute-based access control.

Zhao et al. [120] model RBAC using the description logic language ALCQ. They represent roles as classes and the role hierarchy is expressed by means of subclass relations. They also represent static and dynamic SoD using disjoint axioms. They allow also the representation of role cardinality constraints.

Heilili et al. [50] presents an OWL-based approach to RBAC that considers also negative authorizations, which represent the denial of covering a particular role. They represent roles as classes, and the role hierarchy is expressed using subclassing. They also analyze further modality conflicts due to negative authorizations and how to detect them with reasoning.

In [68, 69] Kolovski et al. show how the WS-Policy language [117] can be mapped to OWL and how standard OWL reasoners can be used to check the conformance of policies and to analyse them. They map policy assertions and alternatives to OWL classes while they map web service requests to individuals. They analyze how operations of Merge and Intersection can be represented using OWL and default rules, and how to tackle the issue of the Open World Assumption.

The work of Cirio et al. [25] aims at representing RBAC and ABAC using semantic web technologies. They combine the two approaches, obtaining a new access control model that enables dynamic assignments of roles to users, taking into account also the values of the attributes. They model roles using OWL classes and they model constraints using SPARQL queries that are stored in OWL annotations.

Reasoning on huge security policies is time consuming and thus Cadenhead

et al. [21] propose a technique to do efficient reasoning on OWL ontologies. Their approach is based on a distributed knowledge base with one TBox and several ABoxes, to diminish the number of symbols in the actual knowledge base with the result of improving performance.

Kolovski et al. in [67] tackle the fact that complex XACML policies are hard to understand and evaluate manually, thus automated tools are needed. To do this they propose to map XACML to Description Logic, this allows them to use off-the-shelf DL reasoners. To capture the behaviour of XACML they use a Defeasible Description Logic [116]. They identify several reasoning services on policies: Policy Comparison, Policy Verification, Incompatibility check, Redundancy check and querying.

Other logical models

We will consider simple or pure access control policy models as "snapshot" access control policy models and more complex ones that take into account temporal statements, such as the number of previous by authorized accesses or even the evolution of the underlying application due to the access themselves, will be called " stateful flexible access control policies".

Pure access control policy Such a policy specification language encodes high level statements into a function from access requests to authorization decisions. This is a way to evaluate action requests to produce a decision taking into account information necessary from the underlying application.

The main notions attached to the analysis of a policy are its consistency its termination, its completeness. Consistency means that no more than one decision is deduced from a request. Completeness means that at least one decision is deduced from a request. Termination means that, eventually after several requests, an answer is obtained. Some formalisms allow also to compare policies.

In [114] Tschantz and Krishnamurthi clearly expose reasonability properties of access control policy languages, whatever the type of language, which can either be based on logics (restricting First Order Logic or augmenting Datalog) or be a custom language such as XACML or EPAL. The latter behave by rule evaluation without any dependence on a theorem proving capabilities. They define precisely the notion of policy language and the main properties of such a language: determinism and totality, which can be mapped respectively to consistency and completeness. They define also a safe policy language with respect to inclusion of requests, and policy combinations through monotonicity of combinators. One request is included in another one when it contains less information than the other, and safety of the language means that an included request does not get more permissions than the including request .

If a FOL based language is always safe, Tschantz shows than one must restrain CoreXACML by withdrawing "explicit denial", so that the resulting language is safe.

Logic based languages Datalog (often Datalog with constraints) is the foundation of many access-control languages and related frameworks [11, 28, 72, 95].

FOL (first order logic) based languages have been created by minimizing the restrictions on them so that they can be decidable and therefore assure consistency and completeness of described policies. Among the FOL based policy languages, Halpern and Weissman have created Lithium and L5, as defined in [48]. Both are safe as subsets of a safe language. L5 has independent composition property but is less expressive than Lithium, which does not have this property. These languages have a foundational approach and well defined complexity results. They have been created to be decidable. However, their bases in pure first-order logic imposes on the policy author the burden of specifying complete definitions (so that every request has a decision), since one is not able to have default decision policies.

Free Variable Tableaux method Concerning conflict detection in standard policy languages such as XACML with RBAC profile, Kamoda et al. in [63] propose a conflict and redundantcy policy detection based on free variable tableaux. Policies are translated into first order logic and all kinds of conflicting policies / redundant policies can be detected with the same algorithm. The method not only detects conflicts, but also provides information to detect redundant rules. This work generalizes the work performed in [62], where Kamoda et al. formalise a logical representation of access control policies for Web Services, which support the description of roles, authorizations and obligations, both positive and negative and the representation of several constraints, such as time constraints, Separation of duty and Chinese Wall. Several kinds of conflicts are supported, like modality conflicts, conflicts caused by propagation and action composition, Separation of duty conflicts and time constraint conflicts. Previously Massacci [77] had also proposed a decision method based on tableaux to detect consistency of a policy. He had defined a logic to express RBAC policies and a decision method to verify automatically the consistency and/or logical consequences of policies. He described also how to implement separation of duty in this logical framework.

Rewriting based rule systems Authorization decisions can be computed by a set of rewrite rules that transform the input terms, representing access requests, into authorization terms. In order to take the raw computational power of term rewriting and to enhance the agility of the policy specification language, one uses strategies to explicitly control the rule application order. The policy specification and its environment are described as terms built over one single signature. The requests are a subset of ground terms. The result returned by a policy can be a ground term containing further information and not only a constant "permit" or "deny". The strategy allows to finely specify the evaluation of the policy rules, therefore the expressivity of the policy can be quite high compared to other logical models. Rewriting system formalisms allow also to properly define consistency, confluence and completeness. Moreover, composition operators of policies are properly defined, not "only over the set of decisions", but on the rules and rewrite strategy defined to derive them.

Didactic explanations are contained for example in [34].

The work of Armando et al. [6] proposes a symbolic framework to improve automated analysis techniques for the management of access-control policies expressed in ARBAC (Administrative RBAC). In [5] Armando et al. propose to apply their approaches in the Health sector and the access control for personal health records (PHR).

Standard Deontic Logic Cholvy and Cuppens [23] represent security policies using Standard Deontic Logic, where actions can be allowed, forbidden, obligatory or waived. In order to decide consistency of policies, they translate them into a set of first order formulae and then the problem of checking consistency can be reduced to a problem of consequence finding.

Lupu et al. [76] focus on conflict detection and resolution in authorization and obligation policies. They identify two kinds of conflicts. The first ones are modality conflicts, which are caused by positive and negative authorizations or obligations on the same target and subject/action. To repair modality conflicts several techniques exist, such as negative takes precedence, explicit priorities, more specific takes precedence or distance functions. The second kind of conflicts are application specific conflicts, such as Separation of Duty, which need meta-polices in order to be expressed and repaired.

Kagal et al. [60] propose Rein, which is a unifying framework that aims at allowing users to express security policies in several languages and to integrate them in the framework. A common representation of policy languages is useful because in this way reasoning services can run on different policies. Rein uses metapolicies to detect and solve conflicts.

Chomicki et al. [24] define the Policy Definition Language (PDL) to express policies with the Event- Condition- Action paradigm. Their language can describe also action constraints, which are sets of actions that cannot occur together. They then provide a technique to translate PDL policies and constraints in logic programs using non recursive Horn logic. They provide a way to identify and solve conflicts in PDL policies simply checking consistency of the resulting logic programs.
They also identify two ways of solving conflicts, deleting actions or deleting events. Computing the maximal action cancellation set is P-TIME, whereas computing the maximal event cancellation set is NP-TIME.

Adaptive access control policy In [12] Becker and Nanz present a logic for specifying policies where access requests can have effects on the authorization state. It leads to more expressive policies that take the history of access requests into account. They present a sound and complete proof system for reasoning about sequences of access requests. This gives rise to a goal-oriented algorithm for finding minimal sequences that lead to a specified target authorization state.

In [33] Dougherty et al. propose a new mathematical model of policies, their environments, and the interactions between them. Influenced by this work, Bandara et al. [26] built a framework to model system behaviour joined to models

of policy enforcement. On the other hand, because the enforcement of an obligation policy changes the state of a system, in addition to modelling the policy specification, it is useful to model system behaviour itself when developing a formal technique for analyzing policies with both modalities: authorization and obligation. Craven, Bandara et al. ([26] for example) use a formalism based on the standard Event Calculus to model both authorization/obligation specifications together with system behaviour. Event Calculus (EC) is a formal language for representing and reasoning about dynamic (event-based) systems which supports a representation of time independent event. EC supports deductive, inductive and abductive reasoning. From EC representations, the authors specify rules to detect both modality conflicts and application specific conflicts, such as separation of duty. Using abductive reasoning techniques, they can provide explanations on how conflicts might arise.

Security policy enforcement Very few authors approach the problem of enforcing security. This is a very ambitious goal, because it poses some serious issues at the infrastructure layer. So, most of these works limit their attention to some higher level abstraction.

The work of Ferrini et al. [39] aims at extending XACML [83] with reasoning capabilities in order to support RBAC and separation of duty constraints. They use OWL because it offers reasoning capability off-the-shelf, using standard Description Logic (DL) reasoners. They propose an extension to the XACML architecture in order to manage semantic functions. They formalize RBAC using a role-as-value approach and their framework supports the representation of both static and dynamic separation of duty constraints, using the owl:disjointWith property. They enforce DSoD constraints only at runtime when a specific XACML request happens, adding to the ontology the needed axioms, extracted from the request, to check the separation. To evaluate the constraints they check the ontology consistency.

Basin et al. [9] present an architecture that provides the capability of enforcing SoD constraints on workflows considering SoD enforcement as a service. They use RBAC and the Separation of Duty algebra, defined by Li et al. in [73], to model SoD on workflows. Their approach has some advantages over solutions that do not consider SoD enforcements as a service:

- it facilitates the separation of concerns,
- it is well suited to work with legacy systems,
- it looses the coupling between workflow engine and repositories.

Basin et al. [10] tackle the fact that access control is important to secure IT systems, but it usually makes harder achieving business goals. They define a formalization of workflows and access control constraints using CSP and consider static SoD constraints, dynamic SoD constraints and dynamic binding of duty constraints. They formalize the concept of *obstruction*, which is a deadlock in a system caused by the access control enforcement. They define the *Enforcement Process Existence Problem* which tries to find an obstruction free enforcement

mechanism. They show that this problem is **NP**-hard, but it can be solved in **P**-TIME using heuristics.

3.2 Checking techniques

3.2.1 Standard DL reasoner

The standard reasoner is one of the core elements of an ontology based system. Starting from the information contained in the ontology, it is able to perform several tasks. In general, it is able to check the consistency and validity of the ontology, classify its information, answer queries, and generate inferences, using a variety of techniques deriving from the work of the Artificial Intelligence community and enriching existing information.

In particular, standard *DL* reasoning performed w.r.t. a formal ontology can check complex consistency constraints in the model. Such constraints are different from the usual ones from database and *UML*-like systems. For example, using *DL*-based languages, like *OWL* (*Web Ontology Language*) and *DAML-OIL*[1], we can express:

- constraints on properties, domains, and ranges;
- definitions of concepts (classes) in terms of relationships with other elements;
- boolean operations on classes; in the simplest case, they allow for the representation of complete, total, and partial refinements.

One of the main differences between *DL* based schema definitions and *UML* or *ER* ones concerns the constraints on property domains and ranges. Properties can be defined in a general way and their behavior in terms of range type can be precisely described while refining the ontology concepts. This promotes the definition and reuse of high level properties, without losing the ability to force precise typing. For example an "*abstraction-of*" property can be used to support the traceability of the refinement process, by connecting one element with another one that represents the same node at a lower level. Generally speaking, such a property will be able to connect almost any element in the model. However, some homogeneity must be guaranteed on the types of element connected by an *abstraction-of* link. In this case we can describe at a lower level axioms like, for example, IT level machines must be refined only in some types of resource (physical servers and virtual machines). Such a description goes beyond classical property domains and range definitions.

3.2.2 Ad-hoc reasoning methods

Standard *Description Logics* reasoners can answer complex questions and can verify structural and non structural constraints. Furthermore, *Description Log-*

[1] `http://www.daml.org/2001/03/daml+oil-index`

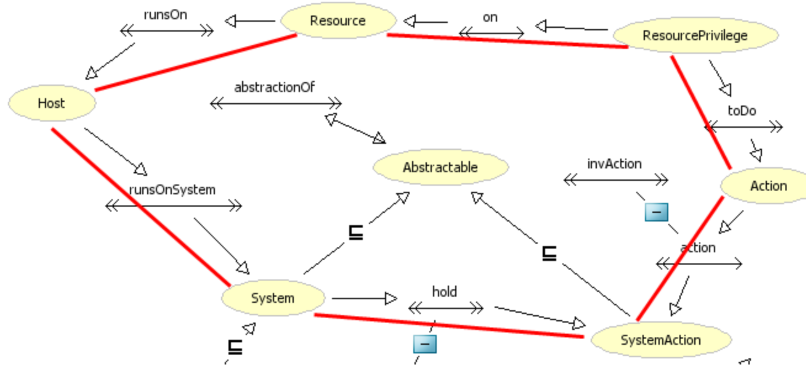


Figure 3.1: A simple consistency constraint

ics based language expressiveness often exceeds classical solutions (like UML for design and SQL for data storage models). This permits the description and verification of more complex structural constraints.

However, *DL* systems, as well as *Semantic Web* tools in general, are designed and implemented with a focus on knowledge management services, such as knowledge integration, schema matching and instance retrieval. Such a specialization poses some limitations on the use of pure *Description Logics* reasoning in scenarios like the one of PoSecCo, in which reasoning must be carried out on a well defined and complete description of a closed system.

In particular, some characteristics of a typical knowledge management oriented description that can be critical and must be carefully considered during the adoption in PoSecCo are:

***Closed World Assumption* (*CWA*) reasoning** is a generally accepted requirement in model driven systems. Conversely, *DL* reasoners usually work under the *Open World Assumption* (*OWA*). This means that the facts asserted in the model (e.g., about the layout topology or the authorization policies) are not assumed to be complete. Obviously, this can become a problem if model characteristics are described in terms of the existence of some properties or some relationships between model elements.

Reasoning on complex property paths (e.g., commutativity of non trivial graphs) rapidly leads to the undecidability of the formal logics the language is based on. Checking the closure of complex paths is beyond the expressive power of classical database systems too, but unfortunately it is sometimes necessary to check structural constraints. This is the case, for example, for the consistency loop in Figure 3.1 stating that

> a *privilege* of executing an *action* must be assigned to a *resource* that runs on a *system* compatible with the *action type*.

Unique Name Assumption (UNA) is a commonly accepted assumption in most model driven tools. It consists in assuming that different names will always denote different elements in the model. This is usually not true in *Description Logics* reasoners, because of the essential nature of knowledge integration problems. In fact, in the *Semantic Web* scenario, different authors may describe the same entities (both shared conceptualizations and physical objects) assigning a new name, generally in the form of a *URI*, defined independently from other *Semantic Web* users.

In order to solve these issues, classical *DL* reasoning must be integrated and completed by adding a set of specific reasoning modules.

Application-level closure

The first and probably simplest point to address is the necessity to use both *Open World Assumption* and *Closed World Assumption* reasoning in the same system. In general, and in the PoSecCo system in particular, reasoning can be used for at least two tasks:

Schema reasoning operates on the *ontological* level of the semantic model. This level comprises the schema (T-box in *Description Logics* terminology) and optionally a very limited number of fundamental instances. The reasoning at this level boils down mostly to consistency checks and concept classification. In this scenario, the *Open World Assumption* is very well suited to drive consequences (theorems) that are correct for any possible model of the schema.

Instance level reasoning involves both schema axioms (as background knowledge) and the individuals used to express actual models at the extensional level. Constraint checking is a typical reasoning task at this level, as well as semantic model completion through the derivation of implicit properties between specific instances. At this level a more database-styled semantics can be used, which interprets the instance level assertions with a closed world semantics.

In order to integrate the two semantics, some metadata must be added at the ontology level to describe which properties must be interpreted under *CWA* at the extensional level. In other words, a set C of properties (or *roles* in *Description Logics* terminology) must be kept, which will be "closed" before any reasoning can be performed. This means that the compatible models that are extensions of the explicit model represented by the user cannot contain links of type R that are not explicitly included by the user and that cannot be derived (proved to exist) starting from the original model.

For each property R in C, $Links_R$ is the set of all the predicates of type R (that is, all the *RDF* triples having R as predicate). $Links_R$ must be extracted. After that:

1. an axiom is added, stating that the individuals that are not listed as a source in any predicate in $Links_R$ must have no R links to any individual;

2. a new axiom is added for each individual i that appears as a source in $Links_R$. The axiom states that i can be connected through R to an individual j if and only if j is an R-successor of i in $Links_R$.

A similar solution can be adopted to obtain the *Unique Name Assumption* semantics. The *Unique Name Assumption* can be considered a sort of *Closed World Assumption* applied to the "*same-as*" relationship; in fact, no individual can be considered as representing the same object if it is not explicitly stated in the model or it can be proved starting from the model itself. However, if we assume that no individual equivalence can be derived from the model, an even simpler solution can be implemented by stating explicitly a "*different-from*" property between all the possible couples of individuals.

3.2.3 Rule based inferencing

Rule inference reasoning is widely used in knowledge management systems. Recently some combinations of theorem proving systems (like *Description Logics* ones) and rule inference systems have been proposed to address some limitations of decidable theorem proving systems.

Semantic Web Rule Language (SWRL) is the *W3C* standard proposal for integrating rule-based inferencing into systems that represent knowledge as a set of *RDF* triples and introducing some limitations to the use of the rules, in order to preserve joint system decidability.

In the PoSecCo scenario, the main advantage of combining rules and classical theorem proving systems is the support for complex property chains. In fact, even if some *OWL2* profiles introduce the support for property (a.k.a *role* in *DL* terminology) chaining, this can be not sufficient to express some complex topological properties. For example, the simple consistency loop shown in Figure 3.1 involves six different properties and requires that a loop must exist for each *toDo* link between a *ResourcePrivilege* and an *Action*. The existence of such a loop is not enforceable at the schema level using only *Description Logics* axioms. Furthermore, as a general outcome of the adoption of the *Open World Assumption*, even if we could enforce the existence of the loops, we would not be able to require that such loops are explicitly stated into the assertional part of the semantic model (*A-box*).

At the opposite end, due to decidability issues (*DL* safe rules) the rule based component of the language operates in a sort of *Closed World Assumption (CWA)* limited to the nodes. This means that a forward chaining rule can be triggered by any property derived by the reasoning, but involving only nodes that are explicitly named in the *A-box*. Then, we can operate only on nodes and properties explicitly stated in the semantic model.

Furthermore, rules can freely combine as antecedents triple patterns to capture complex topological structures, and this solves the lack of complex property chains of *Description Logics*. This means that we can check for loops, or for the absence of loops, by adding custom rules to the PoSecCo ontology.

However, *SWRL* safe rules can consume only positive knowledge, so they can be used to directly detect errors that consist in the existence of some structure in the ontology, that is the existence of a loop. However, if the property that we want to detect involves the non existence of some links, an application-level closure is required. So we have to distinguish between two kinds of rule:

Error detection rules apply when a misconfiguration or an error in the model is detectable by checking positive knowledge. In this case a *SWRL* rule can directly check such a configuration and mark a node as "misconfigured", both by classifying it as an instance of a technical concept (*Misconfigured*) or, even more directly, by making it and the whole model unsound by adding it to a non satisfiable concept (like *owl:Nothing*). In this case the adoption of rule based inferencing can perform the error detection task without any impact at the *DL* and application level.

Property checking rules apply if the error condition consists in the lack of some links. For example, the link shown in Figure 3.4 represents a mandatory situation; if the loop cannot be closed, then a misconfiguration must be revealed. Since inference rules cannot be triggered by the non existence of a link, the rule will detect the correct configuration and will mark the node or the link as "correct". After role execution terminates, a custom application level reasoner will check that all the suitable links and nodes have been marked and verified, if any element is not checked, then a misconfiguration error will be detected.

As an example, the following rule can check if the system on which an action is defined holds an *ActionType* incompatible with the one of the action itself:

$$Action(?a), onSystem(?a, ?s), holds(?s, ?t1), hasType(?a, ?t2), incompatibleWith(?t1, ?t2) \rightarrow \bot(?a)$$

In the same scenario, the following property checking rule verifies that the action is supported by the system. For simplicity the action is classified as *"Ok"*, while in practice a more complete representation will be necessary to enable multiple criteria verification.

$$Action(?a), onSystem(?a, ?s), holds(?s, ?t), hasType(?a, ?t) \rightarrow Ok(?a)$$

After that, the custom reasoner can use the verification result to query for all the actions that are not marked as *Ok*.

3.3 Reasoning patterns

As told in the previous section, there is not a unique reasoning technique that can solve the harmonization problems in all the cases. However, each technique can be useful in some situations and can provide simple and effective solutions to some subproblems.

In spite of the large variety of possible configurations, we can identify some recurrent patterns in verification processes. In this section we will introduce four recurrent patterns, which represent reference scenarios during the design of verification tools. The tools should be flexible enough to model each harmonization module as a very simple workflow, that is the orchestration of atomic verification steps.

Simple SAT

Figure 3.2: Simple satisfiability pattern

Often simple *Description Logics* reasoning is enough to take evidence of some problems in policy models, like in the simple scenario in figure 3.2. *Description Logics* reasoning is dedicated to the verification of semantic constraints and to derive all the entailment of the model.

If the constraint violations are detectable under the *Open World Assumption* a reasoner can be an efficient and simple support for their recognition. This specifically applies to most of the *structural checks*, as described in section 3.4.

3.3.1 Property verification rules

Description Logics axioms can describe quite complex properties on the intended models, however the corresponding logic can lack the necessary power to describe complex constraints defined on paths of properties (property chains) and requiring the commutativity of some subgraphs (namely the agreement operator). In this perspective *Description Logics* are advocated to be *asymmetric*, in the sense that they provide much more expressiveness in concept definition and less expressiveness in property constraints. This typically results in problems while describing topological related properties.

In such cases the user is typically able to describe the constraints via one or few desired configurations. This ends up with the need for a mechanism for:

1. detecting arbitrary interconnected patterns in the model; each detected pattern checks the subgraph as correct w.r.t. the specific constraint;

2. highlight all the violations of the desired constraints, through the difference between all the involved subgraphs and the validated ones.

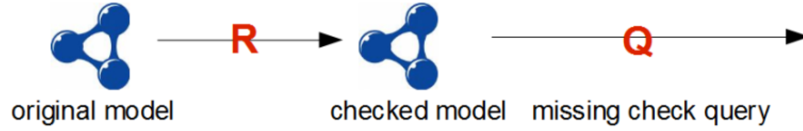


Figure 3.3: Property verification rules pattern

So, we can apply the pattern from figure 3.3; here the original model is enriched by one or more inference rules (typically *SWRL* rules), which add some validation flags to the nodes that match the subgraph. In the second stage a query (or more generally a verification module) performs a sort of closure by extracting all the unchecked relevant nodes.

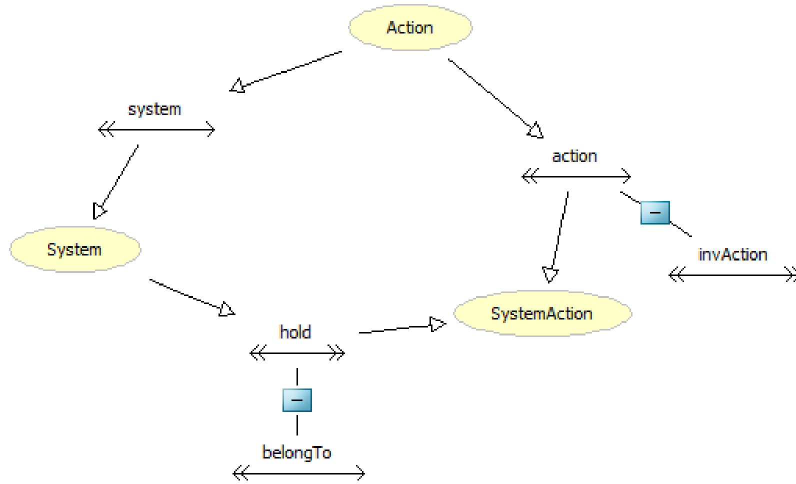


Figure 3.4: An example for property verification rules pattern

Figure 3.4 provides a very simple example of the application of the *property verification pattern*. In this example we want to verify that each *Action* must be involved in a commutative subgraph in which it is relevant to a *System* that is compatible with its *SystemAction* classification. The simple verification rule that verifies the action element is shown in Listing 3.1.

```
type(?x, Action), type(?y, SystemAction), type(?z, System),
action(?x,?y),belongTo(?y,?z),system(?x,?z)
->
type(?x, Correct)
```

Listing 3.1: Verification rule

We can notice that, if the reasoner is available during rule execution, from our reference ontology (and thanks to property domains and range definition)

the types of involved nodes can be inferred and this results in the simplification of rule deduction.

3.3.2 Violation detection rules

Sometimes it is easier to define misconfigurations than the correct desired property. Assuming again that the expressive power of a *Description Logics* reasoner is not sufficient for our goals, we can slightly modify the previous pattern to use subgraphs topology to directly detect conflicts or misconfigurations.

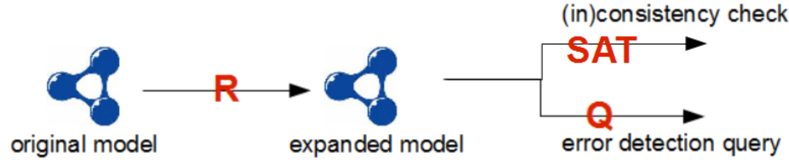


Figure 3.5: Violation detection rules pattern

The violation detection rules pattern, shown in figure 3.5, is slightly simpler than the property verification one and probably more efficient. The improvement involves both the stages of the workflow and it is due to three considerations:

1. statistically, in real scenarios we foresee that the number of misconfigurations will be dramatically smaller than the correct subgraphs. This results in less fires of enrichment rules.

2. we are not required to execute closure, since the decorations introduced by the rules are directly revealable as misconfigurations. So the query in the second stage will not require to evaluate all the interested nodes. Another effect of this consideration is that the query will be usually very simple, and it will simply extract all the marked nodes.

3. just like in the verification pattern, the first stage rules can highlight misconfigured nodes by classifying them under a technical concept. However, in this case we can simply define such a technical concept to be unsatisfiable; in this way we can even replace the second stage query with a simple satisfiability check, which will reveal inconsistencies for each marked node.

Figure 3.6 presents a very simple property that we can check with the misconfiguration detection rule from Listing 3.2.

```
on(?p, ?res), toDO(?p, ?action), system(?action, ?sys1)
,onSystem(?res,?sys2), differentFrom(?sys,?sys2)
->
type(?p, Error)
```

Listing 3.2: Error detection rule

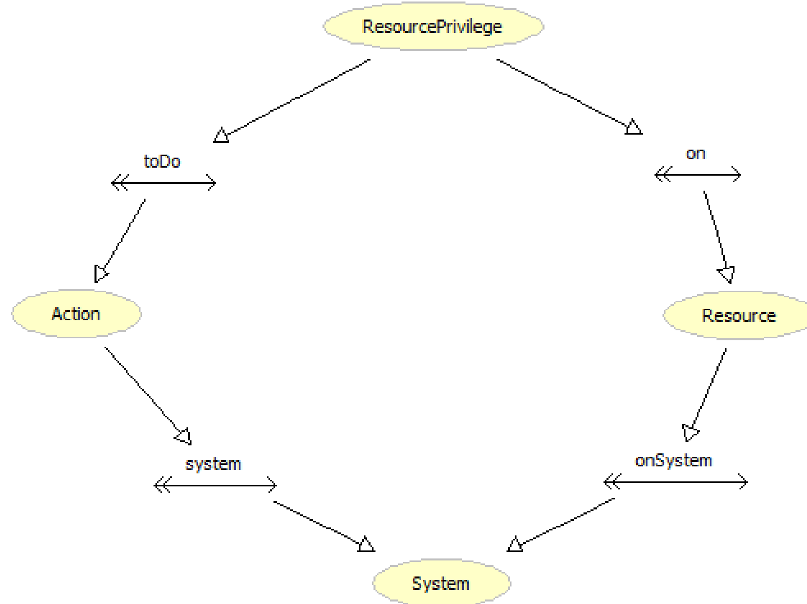


Figure 3.6: An example of violation detection rules pattern

The constraint is very similar to the one from the previous section, but now we want to check that *all* the systems on which the privilege is granted on are compatible with the *Action*. The two verifications are equivalent only if

1. the ontology defines all the properties to be functional and serial (on the correspondent domains);

2. we have no missing links in the model (ABox), since *SWRL* rules are applied under the *Closed World Assumption*.

3.3.3 Custom module pattern

In the most complex cases the first phase of harmonization process cannot be reduced to model enrichment via forward chaining inference rules. Some highly specialized elaborations are required to prepare the model for classic *Description Logics* reasoning.

After the model preparation different techniques for harmonization can be followed, including *SPARQL queries* and *Description Logics* reasoning.

A very common application of this pattern is the reduction of *Closed World* reasoning to *Open World* reasoning as shown in figure 3.7. Generally speaking, the *Closed World Assumption* can be reduced to the *Open World Assumption* by adding closure axioms, given that the reference logics allows for nominals or the definition of concepts (classes) by instance enumeration. This means that we

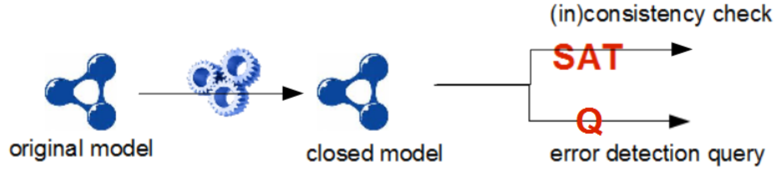


Figure 3.7: Custom module pattern

can obtain a *Closed World Assumption* reasoning service using an *Open World Assumption* reasoner. The converse is not true in the general case.

So, every time we have *positive* constraints, that is constraints that requires the existence of some individuals or some relationships between individuals, we can use an ad hoc closure module that adds closure axioms to the model before passing it to a classic *Description Logics* reasoner.

3.4 Types of check

3.4.1 Structural Checks

The ontological representation of the ITSecurity meta-model permits to define several models. However, some models cannot be considered correct from a structural point of view, because they break some semantic constraints that can not be expressed using only the ontology. Structural checks aim at verifing if the policy under analysis is structurally incorrect. Policies that do not pass structural checks represent wrong policies, and thus they do not represent a useful information. In the following we will present two methods in order to perform structural checks.

Structural Checks on resource privilege The aim of this check is to verify if a resource privilege is consistent. We want to check if the resource assigned to the privilege is in the same system on which the action allowed in privilege can act.

```
on(?resPriv, ?res), onSystem(?res, ?syst1),
system(?action, ?syst2),toDo(?resPriv, ?action)
-> SameAs (?syst1, ?syst2)
```

Listing 3.3: Structural Check

The query that implements the check is presented in Listing 3.3. The assumption behind the right behaviour of the query is the Unique Name Assumption (UNA) on the System class, and thus each individual of the System class must be explicitly declared as *DifferentFrom* the others. In this way, when we have a resource privilege that is related to an action on a first system and a resource on a second one, the reasoner infers that the two systems are the same, in conflict with the UNA, and thus it identifies an inconsistency in the ontology.

Structural Checks on hierarchy The PoSecCo ontology allows the representations of several kinds of hierarchies, i.e. the hierarchy of principals, the hierarchy of identities, the hierarchy of roles, a containment hierarchy of resources and an abstraction hierarchy of resources. These hierarchies represent partially ordered sets, built upon the set of all roles, principals or resources. We can, thus, define the $\leq$ relation over all hierarchies in such a way:

- for the hierarchy of principals and roles the partial order relation is represented by the relation *canActAs* because if *a canActAs b* then $a \geq b$,
- for the hierarchy of roles the partial ordered relation is represented by the relation *canBe*, because if *a canBe b* then $a \geq b$,
- for the hierarchy of identities the partial order relation is represented by the relation *contains*, because if *a contains b* then $a \geq b$,
- for the containment hierarchy of resources the partial order relation is represented by the relation *containsResource*, because if *a containsResource b* then $a \geq b$,
- for the abstraction hierarchy of resources the partial order relation is represented by the relation $abstractionOf$, because if *a* $abstractionOf$ *b* then $a \geq b$.

The fact that these hierarchies are partial orders means that they can not contains cycles. These checks can be done using the rules in Listing 3.4, assumed that the UNA is valid for principals and resources and the transitive closure is computed on all the partial order relations. The relation $PosetError : Thing \rightarrow Thing$ is created in order to keep trace of this kind of error.

```
canActAs(?p1,?p2),canActAs(?p2,?p1),
    DifferentFrom(?p1,?p2)
-> posetError(?p1,?p2)

canBe(?r1,?r2), canBe(?r2,?r1) -> posetError(?r1,?r2)

contains(?id1,?id2), contains(?id2,?id1) ->
    posetError(?id1,?id2)

containsResources(?r1,?r2),
    containsResources(?r2,?r1) -> posetError(?r1,?r2)

abstractionOf(?r1,?r2), abstractionOf(?r2,?r1) ->
    posetError(?r1,?r2)
```

Listing 3.4: Partial orders checks

3.4.2 Consistency Checks

A crucial advantage of the use of Semantic Web technology is the possibility of an early identification of anomalies in the security policy. Security policies in real systems often exhibit conflicts, i.e., inconsistencies in the policy that can lead to an incorrect realization of the security requirements, and redundancies. The availability of a high-level and complete representation of the security policies supports the construction of services for the analysis of the policies able to identify these anomalies and possibly suggest corrections. In the following we will see several approach in order to identify inconsistencies:

- Consistency Checks on Actions;
- Consistency Checks on Requirements;
- Consistency Checks on containsResource property (as requirements);
- Consistency Checks on containsResource property (as conflicts);
- Consistency Checks on authorizations.

Consistency Checks on Actions Consistency between actions can be seen as a sort of separation of duty, but related to actions instead of roles. The idea is to check that a principal can not hold two Actions, on the same resource, that belong to two SystemActions that are explicitly declared in conflict.
In order to do this we have introduced two object properties $ConflictWith : SystemAction \rightarrow SystemAction$, which represents conflicts between SystemActions that cannot be hold concurrently by a principal, and $ConsistentWith : SystemAction \rightarrow SystemAction$, which represent the consistency between two SystemActions. $ConsistentWith$ is derived by means of SWRL rules.
We have also introduced the constraint that $ConflictWith \cap ConsistentWith = \emptyset$. The $ConflictWith$ property must be stated explicitly in the ontology, while the values for the $ConsistentWith$ property can be inferred using the two SWRL rules presented in Listing 3.5.

```
action(?a1, ?action1), action(?a2, ?action2),
    grantedTo(?auth1, ?principal),
grantedTo(?auth2, ?principal),  on(?p1, ?r1), on(?p2, ?r2),
    privilege(?auth1, ?p1),
privilege(?auth2, ?p2),  toDo(?p1, ?a1), toDo(?p2, ?a2),
    SameAs (?r1, ?r2)
-> consistentWith(?action1, ?action2)
```

Listing 3.5: Consistency on actions

When there is a problem in the consistency of actions the reasoner detects it, because the properties $ConflictWith$ and $ConsistentWith$ are not disjoint anymore, violating the constraint expressed before and thus making the ontology inconsistent.

To correctly handle the hierarchy of resources the property $abstractionOf$ must be transitive or the SWRL rule in Listing 3.6 must be used.

```
abstractionOf(?r1, ?r2), abstractionOf(?r2, ?r3) ->
    abstractionOf(?r1, ?r3)
```

Listing 3.6: abstractionOf transitive closure

Using the SWRL rules presented above the hierarchy of *Principals* is ignored. In order to consider also this hierarchy, another transitive property must be added to the ontology, $canActAs : Principal \rightarrow Principal$. It represents a sort of transitive closure on the hierarchy of *Principals*, and it tells, for each principal, what are the other principals he can act as.

The rules presented in Listing 3.7 initialize the reasoning process, which is kept on automatically by the reasoner, because *canActAs* is a transitive property.

```
Principal(?p) -> canActAs(?p,?p)
contains(?group,?id) -> canActAs(?id,?group)
canHaveRole(?id,?role) -> canActAs(?id,?role)
canBe(?role1,?role2) -> canActAs(?role1,?role2)
```

Listing 3.7: *canActAs* transitive property

Now the rules that detect conflicts can be modified as in Listing 3.8.

```
action(?a1, ?action1), action(?a2, ?action2),
    grantedTo(?auth1, ?principal2),
grantedTo(?auth2, ?principal1),
    canActAs(?principal1,?principal2),   on(?p1,?r1),
on(?p2, ?r2), privilege(?auth1, ?p1), privilege(?auth2, ?p2),
     toDo(?p1,?a1),
toDo(?p2, ?a2), SameAs (?r1, ?r2)
-> consistentWith(?action1, ?action2)
```

Listing 3.8: Consistency on actions, considering Principal hierachy

Consistency Checks on Requirements In order to define prerequisites on actions (e.g., in order to do a Write on a resource we must have the permission to do Read on the same resource), we have introduced the relation $requires : Action \rightarrow Action$. The *requires* property is transitive (otherwise the rule in Listing 3.9 can be used to compute the transitive closure).

```
requires(?action1,?action2), requires(?action2,?action3) ->
    requires(?action1,?action3)
```

Listing 3.9: requires transitive closure

To keep trace of conflicts, the class *principalResourcePair* has been created in the ontology. This class represents the reification of a pair of a *Principal* and a *Resource*. It is involved in two properties, $requiresAction : principalResourcePair \rightarrow SystemAction$, which represents pending requirements, and $satisfy : principalResourcePair \rightarrow SystemAction$, which represents the satisfied requirements. The consistency check can be done in three steps:

1. First we compute, for each pair in *principalResourcePair*, the requirements, using the SWRL rule in Listing 3.10.

```
PositiveSystemAuthorization(?posAuth1), action(?a1,
    ?action1),
grantedTo(?posAuth1, ?principal), on(?p1, ?r1),
principalInPair(?pair, ?principal), privilege(?posAuth1,
    ?p1),
requires(?action1, ?action2), resourceInPair(?pair,
    ?r1), toDo(?p1, ?a1)
->
requiresAction(?pair, ?action2)
```

Listing 3.10: First step of requirements evaluation

2. In the second step we compute, for each pair in *principalResourcePair*, all the satisfied requirements, using the rules in Listing 3.11. Using the property *canActAs* the requirements computation can span over the *Principal* hierarchy graph.

```
PositiveSystemAuthorization(?posAuth1), action(?a1,
    ?action1),
grantedTo(?posAuth1,?principal1),
    canActAs(?principal,?principal1),
on(?p1,?r1), principalInPair(?pair, ?principal),
privilege(?posAuth1, ?p1), requiresAction(?pair,
    ?action1),
resourceInPair(?pair, ?r1), toDo(?p1, ?a1)
->
satisfy(?pair, ?action1)
```

Listing 3.11: Second step of requirements evaluation

3. In the third step we can compute all the unsatisfied requirements simply making a difference between *requiresAction* and *satisfy*, as shown in the SPARQL 1.1 query in Listing 3.12.

```
PREFIX pos_ont:
    <http://www.posecco.eu/ontologies/pos_ont#>
PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?pair
FROM <http://www.posecco.eu/ontologies/pos_ont.owl>
      WHERE {
        ?pair rdf:type pos_ont:principalResourcePair .
        ?pair pos_ont:requiresAction ?action1 .
               NOT EXISTS { ?pair pos_ont:satisfy
                   ?action1 . }
      }
```

Listing 3.12: Third step of requirements evaluation

For this check, the main contributor to performance issues is the class $principalResourcePair$, because, while the other classes depend strictly on the policy and the security domain to represent, in the worst case $principalResourcePair \equiv Principal \times Resource$. Thus a good choice of the pairs to represent in the ontology can give an improvement to the performance. Given the ontology, before the execution of the three steps described above, we can execute some Java code in order to create a greedy optimized set of individuals of the $principalResourcePair$ class. The idea is to create some temporary groups and resources in order to minimize the number of pairs. This can be done visiting the policy representation in the memory of the Eclipse plugin.

The optimization can be one of the following:

- Given a group g and action act, if there exists in the policy the authorization $a_i : grantedTo(a_i, u_i) \wedge privilege(a_i, p_i) \wedge on(p_i, r) \wedge toDo(p_i, ac_i) \wedge action(ac_i, act)$ such that $contains(g, u_i) \forall u_i \in g$, we can delete from the ontology all the authorizations a_i and create a new authorization a such that $grantedTo(a, g) \wedge privilege(a, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, act)$. In this way we moved the authorization from the single members of the group to the group itself. This process can be done iteratively, until it reaches a fixed point, when we can stop (the fact that the hierarchy of principal is a poset means that the process will always reach a fixed point). At the end of this process we can create all the individuals in $principalResourcePair$. For all the PositiveSystemAuthorization instances we can create a pair of a Principal, the grantor of the authorization, and a resource, the resource on which the ResourcePrivilege is given. Although this is a conservative optimization, it can improve the performance.

- For each resource r and action act, the module creates a temporary group g that contains all the users $u_1, \ldots, u_n$ such that $\exists a \in PositiveSystemAuthorization : grantedTo(a, u_i) \wedge privilege(a, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, act)$. In this way we can have, for each resource, a temporary group that contains all the principals that are allowed to execute a specific system action on it. Then we delete, for each user in the temporary group g, the authorizations that have the resource privilege to do the *SystemAction* *act*. After this we create a new authorization a such that $grantedTo(a, g) \wedge privilege(a, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, act)$. In this way we move the authorization from the single individuals to the temporary group just created. This process can be done iteratively, until it reaches a fixed point, when we can stop. At the end of this process we can create all the individuals in $principalResourcePair$. For all the PositiveSystemAuthorization instances we can create a pair of a Principal, the grantor of the authorization, and a resource, the resource on which the ResourcePrivilege is given. This is a less conservative optimization than the first one and can be too strict. A possible solution is to use a branch and bound approach. We execute the optimization as illustrated before,

then we execute the reasoning, thus for the pairs that have unsatisfied requirements and involve a temporary group as principal, we execute a new reasoning phase, without considering the temporary group. In this way we can have a big improvement if in the first phase all the requirements are satisfied.

Consistency Checks on containsResource property (as requirements)
The ontology allows the representation of the fact that a *Resource* can be contained in another one, such as a folder that contains a file. This fact is represented by means of the transitive property $containsResources : Resource \rightarrow Resource$.

In order to check the consistency of this property, a separated check is needed because the semantics of the *containsResource* property is slightly different from the one of the $abstractionOf$ property. As an example, in order to do a 'write' on a file, where 'write' is a *SystemAction*, we must have 'access' to the folder that contains the file, where also 'access' is a *SystemAction*, thus the *containsResource* property cannot be considered as the $abstractionOf$ property because it can involve a different *SystemAction* on the container resource.

In order to manage this semantics we have introduced in the ontology the relation $actionNeededOnContainer : Resource \rightarrow SystemAction$, which, for a given *Resource* gives back the *SystemAction* that a Principal must hold on the container in order to do anything on the resource.
The example presented before can be mapped in this way: we have two resources, file and folder
$Resource('file')$, $Resource('folder')$, such that $containsResource('folder','file')$ and
$actionNeededOnContainer('file','access')$. The assumption is, despite the principal has an authorization on the contained resource, the authorization on the container must be given explicitly.

The schema of the check is, then, quite similar to the one used for the check on requirements, because the consistency on the *containsResource* property can be seen as a requirement. We thus add two properties $requiresActionOnContainer : principalResourcePair \rightarrow SystemAction$, which represents pending requirements on the container, and $satisfyOnContainer : principalResourcePair \rightarrow SystemAction$, which represents the satisfied requirements on the container.

The consistency check can be done in three steps:

1. First we compute, for each pair in *principalResourcePair*, the container requirements, using the SWRL rule in Listing 3.13.

```
containsResources(?r1,?r2),
    PositiveSystemAuthorization(?auth),
grantedTo(?auth,?princ),privilege(?auth,?priv),
    on(?priv,?r2),
actionNeededOnContainer(?r2,?sysact),
    resourceInPair(?pair,?r2),
principalInPair(?pair,?princ)
```

```
–>
requiresActionOnContainer(?pair,?sysact)
```

Listing 3.13: First step of containment requirements evaluation

2. In the second step we compute, for each pair in *principalResourcePair*, all the satisfied requirements, using the rules in Listing 3.14. Using the property *canActAs* the requirements computation can span over the *Principal* hierarchy graph.

```
PositiveSystemAuthorization(?posAuth1), action(?a1,
    ?action1),
grantedTo(?posAuth1,
    ?principal1),canActAs(?principal,?principal1),
on(?p1,?r2), containsResource(?r2,?r1),
principalInPair(?pair, ?principal), privilege(?posAuth1,
    ?p1),
requiresActionOnContainer(?pair, ?action1),
resourceInPair(?pair, ?r1), toDo(?p1, ?a1)
–>
satisfyOnContainer(?pair, ?action1)
```

Listing 3.14: Second step of requirements evaluation

3. In the third step we can compute all the unsatisfied requirements simply making a differentiation between *requiresActionOnContainer* and *satisfyOnContainer*, as shown in the SPARQL 1.1 query in Listing 3.15.

```
PREFIX pos_ont:
    <http://www.posecco.eu/ontologies/pos_ont#>
PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?pair
FROM
    <http://www.posecco.eu/ontologies/pos_ont.owl>
WHERE {
?pair rdf:type pos_ont:principalResourcePair .
        ?pair pos_ont:requiresActionOnContainer
            ?action1 .
        NOT EXISTS { ?pair
            pos_ont:satisfyOnContainer ?action1 .
            }
}
```

Listing 3.15: Third step of requirements evaluation

The optimization step is the same as the one for the check on requirements consistency.

Consistency Check on the containsResource property (as conflicts) is another check that aims to control if the authorizations on resources involved in the *containsResource* relation are in conflict or not. Previously, we presented a way to solve the problem as a requirement, thus expecting that the authorization on the container must be explicitly given in order to hold an authorization on the contained resource; in this section we present another way to solve the problem, considering that if we have an authorization on the contained resource we also have implicitly an authorization to do the needed action on the container, if not explicitly specified otherwise. If we have the authorization to do a 'write' on the file, we have, implicitly, also the authorization to do 'access' on the folder that contains the file. The conflict exists only if we have also a negative authorization to do 'access' on the folder.

We can thus reuse the *pairOfAuthorizations* class, in which the positive authorization is always the explicit authorization on the contained resource and the negative one is the explicit authorization that forbids the needed action on the contained resource. In any other case the conflict does not exist.

Also here we can have several situations:

- If the negative authorization and positive authorization are granted to the same principal, then the negative one wins, as shown in Listing 3.16.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p2,?a2),
action(?a2,?action2), on(?p1,?r1), on(?p2,?r2),
actionNeededOnContainer(?r1,?action2),
    containsResources(?r2,?r1),
SameAs(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action2),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.16: Conflict on containsResources, negative authorization wins

- If the positive authorization is given at the more specialized level, then the positive authorization wins, as shown in Listing 3.17.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p2,?a2),
```

```
action(?a2,?action2), on(?p1,?r1), on(?p2,?r2),
actionNeededOnContainer(?r1,?action2),
    containsResources(?r2,?r1),
DifferentFrom(?pr1,?pr2), canActAs(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?posAuth),
involvingAction(?pair,?action2),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.17: Conflict on containsResources, positive authorization wins

- If the negative authorization is given at the more specialized level, then the negative authorization wins, as shown in Listing 3.18.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p2,?a2),
action(?a2,?action2), on(?p1,?r1), on(?p2,?r2),
actionNeededOnContainer(?r1,?action2),
    containsResources(?r2,?r1),
DifferentFrom(?pr1,?pr2), canActAs(?pr2,?pr1)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action2),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.18: Conflict on containsResources, more specialized authorization wins

Consistency Checks on authorizations Consistency on authorizations aims at checking that it does not exist any pair of System authorization that can create conflicts on the hierarchy of Principals.

To keep trace of conflicts we have created the class *pairOfAuthorizations* in the ontology. This class represents the reification of a pair of a *NegativeAuthorization* and a *PositiveAuthorization*. At the end of the check, if the pair is involved in a conflict it contains also the list of the actions and resources involved in a conflict. The assumption behind the check is that exists an individual that represents each pair of authorizations, and that for principals the Unique Name Assumption is valid.

Figure 3.8 shows a quite complete example of inconsistency. We have a positive authorization *auth 1* that grants the right to execute *action 1* on *resource 1*. However *auth 2* disallows the user the right to execute *action 2* on a higher

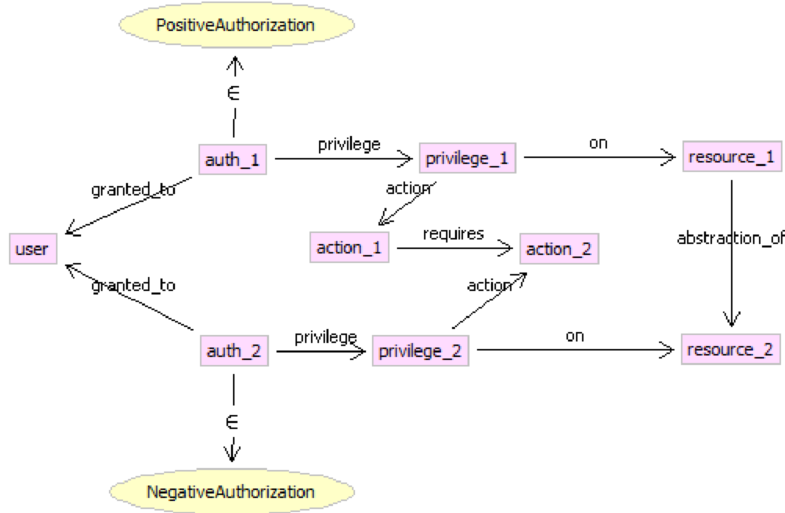


Figure 3.8: An example of an inconsistency on authorizations

level abstraction of the same resource. Since we know from the action model that *action 2* is a prerequisite to execute *action 1*, we conclude that the two authorizations are incompatible.

This example involves different fragments of the ontology:

authorization ontology describes the signs of the involved authorizations, and the fact that they are granted to the same individual;

cross levels traceability is necessary because the inconsistency involves elements at different levels of abstraction (the resources in this example);

action ontology provides the knowledge about action types dependencies.

Three different hierarchies are involved in this check: the hierarchy of resources, the hierarchy of roles and the hierarchy of identities. At first time we suppose that we can ignore the resource hierarchy. Thus we need an order of specialization to identify which principal is the more specialized. This information is expressed by the *canActAs* relation because, for construction, a *canActAs* b means that a is more specialised than b. The only special case is if a and b are the same individual.

We can thus have two different situations :

- If the conflict is on two principals a and b such that $a = b$, than the conflict is solved in favour of the negative authorization. This is done using the rule presented in Listing 3.19.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
```

```
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action), action(?a2,?action), on(?p1,?r),
    on(?p2,?r),
SameAs(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action), onResource(?pair,?r)
```

Listing 3.19: Conflict evaluation, same principal and no resources hierarchy

- If the conflict is on two principals a and b such that a *canActAs* b, then the conflict is solved in favour of the authorization held by a. This is done using the rules presented in Listing 3.20.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair,?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1),privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action),action(?a2,?action), on(?p1,?r),
    on(?p2,?r),
canActAs(?pr1,?pr2),DifferentFrom(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?posAuth),
involvingAction(?pair,?action), onResource(?pair,?r)
```

Listing 3.20: Conflict evaluation, two principals in the hierarchy and no resource hierarchy

Now if we consider also the resource hierarchy, we can have three additional situations:

- If the conflict is on two principals a and b such that $a = b$ and two resources $r1$ and $r2$ such that $r1$ $abstractionOf$ $r2$, then the conflict is solved in favour of the authorization involving $r2$ because is more specific. This is done using the rule presented in Listing 3.21.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1),grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
```

```
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action), action(?a2,?action), on(?p1,?r1),
    on(?p2,?r2),
SameAs(?pr1,?pr2), abstractionOf(?r1,?r2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.21: Conflict evaluation, same principal and resource hierarchy

- If the conflict is on two principals $pr1$ and $pr2$ such that $pr1$ *canActAs* $pr2$ and on two resources $r1$ and $r2$ such that $r2$ $abstractionOf$ $r1$, then the conflict is solved in favour of the authorization involving $pr1$ and $r1$ because both are the specialized side.
  This is done using the rule presented in Listing 3.22.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action), action(?a2,?action), on(?p1,?r1),
    on(?p2,?r2),
canActAs(?pr1,?pr2), abstractionOf(?r2,?r1),
    DifferentFrom(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?posAuth),
involvingAction(?pair,?action),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.22: Conflict evaluation, principals and resources are coherent

- If the conflict is on two principals $pr1$ and $pr2$ such that $pr2$ *canActAs* $pr1$ and on two resources $r1$ and $r2$ such that $r1$ $abstractionOf$ $r2$, then the conflict is solved in favour of the authorization involving $pr2$ and $r2$ because both are the specialized side.
  This is done using the rule presented in Listing 3.23.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1),grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
```

```
action(?a1,?action), action(?a2,?action), on(?p1,?r1),
    on(?p2,?r2),
canActAs(?pr2,?pr1), abstractionOf(?r1,?r2),
    DifferentFrom(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.23: Conflict evaluation, principals and resources are consistent

- If the conflict is on two principals $pr1$ and $pr2$ such that $pr1$ *canActAs* $pr2$ and on two resources $r1$ and $r2$ such that $r1$ $abstractionOf$ $r2$, we have the case in which there is a problem of variance and counter variance between the two hierarchies. In this case the principal hierarchy dominates the resource hierarchy and thus the conflict is solved in favour of the authorization that involves $pr1$, as shown in the rule in Listing 3.24.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action), action(?a2,?action), on(?p1,?r1),
    on(?p2,?r2),
canActAs(?pr1,?pr2),
    abstractionOf(?r1,?r2),DifferentFrom(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?posAuth),
involvingAction(?pair,?action),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.24: Conflict evaluation, hierarchy conflict ($r1$ $abstractionOf$ $r2$)

- If the conflict is on two principals $pr1$ and $pr2$ such that $pr2$ *canActAs* $pr1$ and on two resources $r1$ and $r2$ such that $r2$ $abstractionOf$ $r1$, we have the case in which there is a problem of variance and counter variance between the two hierarchies. In this case the principal hierarchy dominates the resource hierarchy and thus the conflict is solved in favour of the authorization that involves $pr2$, as shown in the rule in Listing 3.25.

```
PositiveSystemAuthorization(?posAuth),
    NegativeSystemAuthorization(?negAuth),
grantedTo(?posAuth,?pr1), grantedTo(?negAuth,?pr2),
containsAuthorization(?pair, ?negAuth),
    containsAuthorization(?pair, ?posAuth),
```

```
privilege(?posAuth,?p1), privilege(?negAuth,?p2),
    toDo(?p1,?a1), toDo(?p2,?a2),
action(?a1,?action), action(?a2,?action), on(?p1,?r1),
    on(?p2,?r2),
canActAs(?pr2,?pr1),
    abstractionOf(?r2,?r1),DifferentFrom(?pr1,?pr2)
->
solvedAuthorizationConflict(?pair),
    harmonizedAuthorization(?pair, ?negAuth),
involvingAction(?pair,?action),
    onResource(?pair,?r1),onResource(?pair,?r2)
```

Listing 3.25: Conflict evaluation, hierarchy conflict ($r2\ abstractionOf\ r1$)

The optimization phase can be done by diminishing the number of authorizations that exist in the ontology. In order to do this, an approach that removes redundant authorizations can be adopted. We can do the following optimization:

1. For a given group g, a given system action *action* and a resource r, if $\forall i \in contains(g, i)$ a SystemAuthorization exists $sysauth_i$ such that $grantedTo(sysauth_i, i) \wedge privilege(sysauth_i, p_i) \wedge on(p_i, r) \wedge toDo(p_i, ac_i) \wedge action(ac_i, action)$, then we can delete all the authorizations $sysauth_i$ and create a new authorization *sysauth* such that $grantedTo(sysauth, g) \wedge privilege(sysauth, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, action)$. In this way we move the authorization from the identities to the group.

2. For a given role *role*, a given system action *action* and a resource r, if $\forall i \in canHaveRole(i, role)$ a SystemAuthorization exists $sysauth_i$ such that $grantedTo(sysauth_i, i) \wedge privilege(sysauth_i, p_i) \wedge on(p_i, r) \wedge toDo(p_i, ac_i) \wedge action(ac_i, action)$, then we can delete all the authorizations $sysauth_i$ and create a new authorization *sysauth* such that $grantedTo(sysauth, role) \wedge privilege(sysauth, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, action)$. In this way we move the authorization from the identities to the role.

3. For a given role *role*, a given system action *action* and a resource r, if $\forall i \in canBe(role, i)$ a SystemAuthorization exists $sysauth_i$ such that $grantedTo(sysauth_i, i) \wedge privilege(sysauth_i, p_i) \wedge on(p_i, r) \wedge toDo(p_i, ac_i) \wedge action(ac_i, action)$, then we can delete all the authorizations $sysauth_i$ and create a new authorization *sysauth* such that $grantedTo(sysauth, role) \wedge privilege(sysauth, p) \wedge on(p, r) \wedge toDo(p, ac) \wedge action(ac, action)$. In this way we move the authorization from the low level roles to the high level role.

These optimizations can be done iteratively, until they reach a fixed point, i.e., a point after that another iteration of the optimizations cannot diminish the number of authorizations in the ontology.

A less conservative kind of optimization can be done creating temporary groups and roles, which group together principals that have the authorization

to do the same action on the same resource in order to delete the single authorizations creating a new temporary one, using a branch and bound approach.

Redundancy Checks

One of the problems in policy management is the presence of redundant authorizations. A redundant authorization (respectively authentication) is an authorization (authentication) that is dominated by other authorizations (authentications) and that does not contribute to the policy, i.e., its removal would not modify the behavior of the system.

The goal of redundancy analysis is to detect if a policy (or an authorization) is dominated by any other policy (authorization). A policy p_1 is dominated by a policy p_2 if and only if:

1. p_1 and p_2 have the same sign (allow or deny); in such a situation if both the policies apply no conflict arises;
2. p_1 is a specialization of p_2 with respect to all the available dimensions. This guarantees that p_2 applies in any situation in which p_1 does.

We define the property *dominate*: *SystemAuthorization* → *SystemAuthorization* to represent the case in which all the domains associated with an authorization contain the domains associated with another authorization. To identify these cases, we use the following SWRL rule:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1),
    on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr2),
canActAs(?pr2,?pr1), containsResource+(?r2,?r1),
    DifferentFrom(?a1,?a2)  -> dominates(?a1,?a2)
```

We can define that $auth_1$ is redundant with respect to $auth_2$, if $auth_2$ dominates $auth_1$, and the signs of $auth_1$ and $auth_2$ are the same, otherwise the pair of authorizations is not removable. We have thus defined the property *implies* : *SystemAuthorization* → *SystemAuthorization* and the symmetric property *unRemovable* : *SystemAuthorization* → *SystemAuthorization*. Then, we define the following SWRL rules to classify the pairs of authorizations involved in the *dominates* property:

```
dominates(?a1,?a2), sign(?a1,?s), sign(?a2,?s) ->
    implies(?a1,?a2)

dominates(?a1,?a2), sign(?a1,?s1), sign(?a2,?s2),
    DifferentFrom(?s1,?s2) -> unRemovable(?a1,?a2)
```

Redundant authorizations are those authorizations that are redundant at least with respect to another authorization, and are not involved in any instance of the *unRemovable* property. In this way only authorizations that are not involved in a conflict are considered redundant. This is a safe approach, because a redundant authorization involved in one or more conflicts can have an impact on the policy, depending on how the conflicts are solved. The safe removable authorizations can be detected using the following SPARQL query:

```
PREFIX rbac:
    <http://www.posecco.eu/ontologies/accesscontrol#>
SELECT ?auth2 WHERE { ?auth1 rbac:implies ?auth2 .
      NOT EXISTS { ?auth2 rbac:unRemovable ?auth3 . } }
```

Since SPARQL queries are evaluated under the closed world assumption, they can query the result of the execution of the SWRL rules, returning to the security administrator in a direct way the list of authorizations that are redundant and can be removed, whereas classic DL instance retrieval should not be able to detect missing explicit relations.

3.4.3 Separation of Duty Checks

Separation of duty checks are those checks that aim at enforcing the static separation of duty between roles. A SoD constraint between role r_1 and r_2 can be expressed using a negative role authorization r_{auth} such that *enabledRole*(r_{auth}, r_1) and *grantedTo*(r_{auth}, r_2) and *sign*$(r_{auth}, -)$. Before analyzing a policy to detect SoD conflicts we execute a normalization step, which replaces all the role authorizations between roles with semantically equivalent *roleHierarchy* properties in order to simplify the analysis. A role authorization that assigns a role r_2 to a role r_1 is equivalent, for SoD satisfiability analysis, to a super-role relation between r_1 and r_2 and, thus, to *roleHierarchy*(r_1, r_2).

Separation of duty constraints have to be enforced both at role hierarchy level, in this way we directly prevent that a role r_1 is declared super-role of another role r_2 such that r_1 and r_2 are in a SoD constraint, and at user hierarchy level, to avoid that to a user are assigned, directly or indirectly, two roles r_1 and r_2 that are involved in a SoD constraint. In order to keep track of all SoD conflicts on *Roles*, we have defined a class *SoDConflictOnRole* $\sqsubseteq$ *Role*. Separation of duty constraints on the role hierarchy can be expressed adding to the ontology the following set of axioms:

$$\begin{aligned} &\forall auth \in RoleAuthorization : sign(auth, -), grantedTo(auth, r_1), \\ &\qquad enabledRole(auth, r_2) \\ &\quad SoDConflictOnRole \equiv \exists canBe+.\{r_1\} \sqcap \exists canBe+.\{r_2\} \end{aligned}$$

We can thus enforce the SoD at role hierarchy level simply adding the axiom *SoDConflictOnRole* $\sqsubseteq \bot$ to the ontology.

In a similar way to what we have done for the identification of SoD conflicts at role hierarchy level, we defined a class *SoDConflictOnId* $\sqsubseteq$ *Identity* that keeps track of the conflicts on the user hierarchy. We express SoD constraints using the following axioms:

$$\begin{aligned} &\forall auth \in RoleAuthorization : sign(auth, -), grantedTo(auth, r_1), \\ &\qquad enabledRole(auth, r_2) \\ &\quad SoDConflictOnId \equiv \exists canHaveRole+.\{r_1\} \sqcap \exists canHaveRole+.\{r_2\} \end{aligned}$$

and to enforce the SoD constraints we simply have to add to the ontology the axiom *SoDConflictOnId* $\sqsubseteq \bot$.

Permission-Based and Object-Based SoD Checks A first way to handle *Permission-based* and *Object-based* SoD constraints is by extending the ontology with new axioms. For *Permission-based SoD* we defined a new property $canDo : Principal \rightarrow Action$ that is equivalent to $grantedTo^{-1} \circ toDo$. To keep track of the conflicts, we have defined a new class $PSoDConflict \sqsubseteq Principal$ and in order to identify the actions involved in constraints we defined the property $PermBasedSod : Action \rightarrow Action$. We express SoD constraints using the following axioms:

$$\begin{aligned} &\forall a_1, a_2 \in Action : PermBasedSod(a_1, a_2) \\ &\quad PSoDConflict \equiv \exists canDo.\{a_1\} \sqcap \exists canDo.\{a_2\} \end{aligned}$$

and to enforce the SoD, we simply have to add to the ontology the axiom $PSoDConflict \sqsubseteq \bot$. In the same way we can define the *Object-based SoD*. In DL formulas we write $canActOn : Principal \rightarrow Resource$, $canActOn \equiv grantedTo^{-1} \circ on$, $ObjBasedSod : Resource \rightarrow Resource$,
$RSoDConflict \sqsubseteq Principal$. In order to enforce the *Object-based SoD*, we add to the ontology the following set of axioms:

$$\begin{aligned} &\forall res_1, res_2 \in Resource : ObjBasedSod(res_1, res_2) \\ &\quad RSoDConflict \equiv \exists canActOn.\{res_1\} \sqcap \exists canActOn.\{res_2\} \\ &RSoDConflict \sqsubseteq \bot \end{aligned}$$

However, this approach has two drawbacks:

1. it adds complexity to the ontology;
2. it represents all these SoD constraints in a redundant way, because the only difference between all of them is the hierarchy on which the constraint is applied.

A second approach is to implement *Permission-based* and *Object-based SoD* constraints using an ad-hoc role hierarchy and, thus, representing them as *Role-based SoD*. Thus, we can define, for each pair composed by an action $a \in Actions$ and a resource $res \in Resource$, a new role $r_{a,res}$ that represents a high level role. Each role $r_{a,res} \in Role$ has assigned to it only a $SystemAuthorization$, which allows it to do the action a on the resource res. Then, we define for each action $a \in Actions$ a role r_a, and for each resource $res \in Resources$ a role r_{res}. These roles are low level roles; they do not have any authorization assigned to them, but they are used to express the SoD constraints. In order to create the adequate relationships between high level and low level roles, we add the following axioms to the ontology:

$$\begin{aligned} &\forall res \in Resource, \forall a \in Actions : \\ &\quad roleHierarchy(r_{a,res}, r_{res}) \\ &\quad roleHierarchy(r_{a,res}, r_a) \end{aligned}$$

These axioms create a low level of abstract roles, parameterized only on the action or on the resource, that are sub-roles of high level roles. This second level aggregates the roles on a resource and action basis. Now, we can simply express *Permission-based SoD* and *Object-based SoD* using the axioms presented for *Role-Based SoD*, but considering only low level roles, respectively the ones associated with actions or with resources. For instance, in order to express SoD between two resources res_1 and res_2, we can create a negative role authorization r_{auth} such that $grantedTo(r_{auth}, r_{res_1})$ and $enabledRole(r_{auth}, r_{res_2})$, whereas to express the SoD between two actions a_1 and a_2 we can create a negative role authorization r_{auth} such that $grantedTo(r_{auth}, r_{a_1})$ and $enabledRole(r_{auth}, r_{a_2})$.

3.5 Policy Incompatibility

Given a policy that contains a set $A = \{a_1, \ldots, a_n\}$ of SystemAuthorizations, we define: 1. p_i the set of *Principal*s the authorization a_i is granted to, directly or indirectly, 2. *act* the *Action* associated with the authorization a_i, 3. r_i the set of *Resource*s to which the enabled permission can be applied, directly or indirectly. An inconsistency in the policy may arise when exists, at least, $a_i, a_j \in A$, such that $sign(a_i) \neq sign(a_j) \wedge p_i \cap p_j \neq \emptyset \wedge act_i = act_j \wedge r_i \cap r_j \neq \emptyset$.

This kind of conflicts is usually called *Modality conflict* and arises when principals are authorized to do an action a on a resource r by a positive authorization, and forbidden to do the same action a on the resource r by a negative authorization. In this case the two authorizations are said to be *incompatible*.

The management of *Modality conflicts* first requires to detect them. Then, when a conflict is detected, a choice has to be made about how the conflict can be solved, deciding between the positive and negative authorization which is the one that is going to hold.

We note that it is usually impractical to simply remove the "weaker" authorization and to replace the policy with one that does not present modality conflicts. Even in a scenario like the one considered in this paper where the containment structure of principals, actions, and resources is static, this step may lead to an excessive growth of the policy. The common approach consists in identifying a criterion to use in the resolution of conflicts, and applying it every time a modality conflict is detected.

In the literature and in systems, several criteria have been proposed and implemented to solve this kind of conflict [76]. A simple criterion is the "*Denials take precedence*", which states that, in case of conflicts, the Negative Authorization always wins. A more flexible criterion is the "*Most specific Wins*", which states that, when one authorization dominates the other, the more specific wins. While several works already present techniques to formalize and compose modality conflicts, such as [62], [23] and [76], no approaches are known to us that use ontologies and Semantic Web Tools detect this kind of inconsistencies and to solve them using the "*Most specific Wins*" criterion.

For simplicity we introduce the property *canActAs* to take into account the different ways a principal can activate the profile of another principal. *can-*

ActAs: *Principal* $\rightarrow$ *Principal* is a transitive and reflexive property such that *containedIn* $\sqsubseteq$ *canActAs*, *canBe+* $\sqsubseteq$ *canActAs* and *canHaveRole* $\sqsubseteq$ *canActAs*. *canActAs*(p_1, p_2) means that p_1 is more specific than p_2.

We have also introduced the property *winVs*: *SystemAuthorization* $\rightarrow$ *SystemAuthorization* which can be used to express if the Authorization p_1 wins against p_2 (i.e., if p_1 is a specialization in all the components). In our approach we consider the *Unique Name Assumption* (UNA).

We can detect modality conflicts and compose them following the "*Most specific wins*" criterion, using SWRL rules like the following:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1),
    sign(?a1,?s1), on(?a2,?r2), toDo(?a2,?act),
    grantedTo(?a2,?pr2), sign(?a2,?s2),
    containsResource+(?r2,?r1), canActAs(?pr2,?pr1),
    DifferentFrom(?s1,?s2) -> winsVs(?a2,?a1)
```

When the conflict involves two policies that have the same specificity level, we apply the "*Denials take precedence*" criterion, with the following rule:

```
PositiveAuthorization(?a1), SystemAuthorization(?a1),
    on(?a1,?r), toDo(?a1,?act), grantedTo(?a1,?pr),
    NegativeAuthorization(?a2), SystemAuthorization(?a2),
    on(?a2,?r), toDo(?a2,?act), grantedTo(?a2,?pr) ->
    winsVs(?a2,?a1)
```

However, not all the conflicts can be solved using these criteria, because in some conflicts we can not find a policy that is more specific than the other one (e.g., one has a high level in the resources and the other in the principals).

To detect these cases, which we assume have to be notified to the security administrator responsible for the design of the policy, we defined a new symmetric property *unsolvableConflict*: *SystemAuthorization* $\rightarrow$ *SystemAuthorization*, which contains all the pairs of policies involved in conflicts that are not solvable using the "*Most specific wins*" criterion. Unsolvable conflicts are detected by means of SWRL rules like the following one, an extended presentation of the SWRL rules managing this service appears in Appendix C.1:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1),
    sign(?a1,?s1), on(?a2,?r2), toDo(?a2,?act),
    grantedTo(?a2,?pr2), sign(?a2,?s2),
    containsResource+(?r2,?r1), canActAs(?pr1,?pr2),
    DifferentFrom(?s1,?s2) -> unsolvableConflict(?a1,?a2)
```

The *Policy Incompatibility* service produces three sets of policies: 1. a set of authorizations that does not contain any conflict, 2. a set of conflicting authorizations, in which all the conflicts have been solved, 3. a set of unsolved conflicts, which should be fixed by security administrators.

On the set of solved conflicts, namely the union of the first and second set, we can apply the others services presented in the following sections.

3.6 Policy Minimization

Access Control policies usually contain a large number of authorizations, created by security administrators. One of the problems in policy management is the presence of redundant authorizations. A redundant authorization is an authorization that is dominated by other authorizations and that does not contribute to the policy, i.e., its removal would not modify the behavior of the system.

Given two authorizations a_1 and a_2 with same action and sign, and called p_i (respectively r_i) the principals (respectively resources) associated, directly or indirectly, with a_i. If $p_2 \subseteq p_1$ and $r_2 \subseteq r_1$ and a_2 is not involved in any conflict with other authorizations, then a_2 is redundant with respect to a_1, and can be safely removed from the policy without modifying the behaviour of the system. The *Policy Minimization* function offers to security administrators a way to detect redundant authorizations.

We define the property *dominate*: *SystemAuthorization* → *SystemAuthorization* to represent the case in which all the domains associated with an authorization contain the domains associated with another authorization. To identify these cases, we use the following SWRL rule:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1),
    on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr2),
    canActAs(?pr2,?pr1), containsResource+(?r2,?r1),
    DifferentFrom(?a1,?a2)  -> dominates(?a1,?a2)
```

We can define that $auth_1$ is redundant with respect to $auth_2$, if $auth_2$ dominates $auth_1$, and the signs of $auth_1$ and $auth_2$ are the same, otherwise the pair of authorizations is not removable. We have thus defined the property *implies* : *SystemAuthorization* → *SystemAuthorization* and the symmetric property *unRemovable* : *SystemAuthorization* → *SystemAuthorization*. Then, we define the following SWRL rules to classify the pairs of authorizations involved in the *dominates* property:

```
dominates(?a1,?a2), sign(?a1,?s), sign(?a2,?s) ->
    implies(?a1,?a2)

dominates(?a1,?a2), sign(?a1,?s1), sign(?a2,?s2),
    DifferentFrom(?s1,?s2) -> unRemovable(?a1,?a2)
```

Redundant authorizations are those authorizations that are redundant at least with respect to another authorization, and are not involved in any instance of the *unRemovable* property. In this way only authorizations that are not involved in a conflict are considered redundant. This is a safe approach, because a redundant authorization involved in one or more conflicts can have an impact on the policy, depending on how the conflicts are solved. The safe removable authorizations can be detected using the following SPARQL query:

```
PREFIX rbac:
    <http://www.posecco.eu/ontologies/accesscontrol#>
SELECT ?auth2 WHERE { ?auth1 rbac:implies ?auth2 .
      NOT EXISTS { ?auth2 rbac:unRemovable ?auth3 . } }
```

Since SPARQL queries are evaluated under the closed world assumption, they can query the result of the execution of the SWRL rules, returning to the security administrator in a direct way the list of authorizations that are redundant and can be removed, whereas classic DL instance retrieval should not be able to detect missing explicit relations.

3.7 Separation of Duty

A common class of constraints represented in security policies is *Separation of Duty* (SoD). These constraints follow the common best practice for which sensitive combinations of permissions should not be held by the same individual in order to avoid the violation of business rules. *Role Based Access Control* is particularly well-suited to express this kind of constraints because the role hierarchy allows an easy mapping of real world business rules to the access control model.

In the following, we primarily consider *Role-based Static SoD*. Given two roles r_1 and r_2, the Static SoD between these two roles means that it must not exist a user u who can be assigned both r_1 and r_2.

We focus on the Static SoD, but the model can be immediately extended to support also Dynamic SoD constraints, where it is required to check that a user does not concurrently activate conflicting roles (it is sufficient to assume that the model provides the concept of *active* role). The aim of the *SoD Satisfiability* service is to check whether a set of authorizations satisfies a set of SoD constraints.

In our model the roles are represented as individuals of the class $AuthzRole$, and the role hierarchy is expressed using the property *roleHierarchy* and the user role assignment relation is represented using the RoleAuthorizations introduced in Section 2.2.2, which allow a user to enable a specific role.

A SoD constraint between role r_1 and r_2 can be expressed using a negative role authorization r_{auth} such that $enabledRole(r_{auth}, r_1)$ and $grantedTo(r_{auth}, r_2)$ and $sign(r_{auth}, -)$. Before analyzing a policy to detect SoD conflicts we execute a normalization step, which replaces all the role authorizations between roles with semantically equivalent *roleHierarchy* properties in order to simplify the analysis. A role authorization that assigns a role r_2 to a role r_1 is equivalent, for SoD satisfiability analysis, to a super-role relation between r_1 and r_2 and, thus, to $roleHierarchy(r_1, r_2)$.

Separation of duty constraints have to be enforced both at role hierarchy level, in this way we directly prevent that a role r_1 is declared super-role of another role r_2 such that r_1 and r_2 are in a SoD constraint, and at user hierarchy level, to avoid that to a user are assigned, directly or indirectly, two roles r_1 and r_2 that are involved in a SoD constraint.

In order to keep track of all SoD conflicts on *Roles*, we have defined a class $SoDConflictOnRole \sqsubseteq Role$. Separation of duty constraints on the role hierarchy can be expressed adding to the ontology the following set of axioms:

$$\forall auth \in RoleAuthorization : sign(auth, -), grantedTo(auth, r_1), \\ enabledRole(auth, r_2) \\ SoDConflictOnRole \equiv \exists canBe+.\{r_1\} \sqcap \exists canBe+.\{r_2\}$$

We can thus enforce the SoD at role hierarchy level simply adding the axiom *SoDConflictOnRole* $\sqsubseteq \perp$ to the ontology.

In a similar way to what we have done for the identification of SoD conflicts at role hierarchy level, we defined a class *SoDConflictOnId* $\sqsubseteq$ *Identity* that keeps track of the conflicts on the user hierarchy. We express SoD constraints using the following axioms:

$$\forall auth \in RoleAuthorization : sign(auth, -), grantedTo(auth, r_1), \\ enabledRole(auth, r_2) \\ SoDConflictOnId \equiv \exists canHaveRole+.\{r_1\} \sqcap \exists canHaveRole+.\{r_2\}$$

and to enforce the SoD constraints we simply have to add to the ontology the axiom *SoDConflictOnId* $\sqsubseteq \perp$. Appendix C.4 presents a pruning algorithm that can be used to improve the performance of the *Separation of Duty satisfiability* service.

Our approach can be easily extended to handle other kinds of SoD constraints, such as *Permission-based SoD* (which requires that no user is allowed to do both actions a_1 and a_2) or *Object-based SoD* (which requires that no user can access both resources res_1 and res_2), as shown in Appendix C.3.

Using a similar approach, we can implement the *Policy Incompatibility* service using the *SoD Satisfiability* service. We only need a preprocessing step. The preprocessing algorithm is presented in Algorithm 5 in Appendix C.2. It takes as input the set of roles R, the set of System authorizations SA, the set of Role authorizations RA, and the role hierarchy RH. The algorithm, for each System authorization *sysauth*, creates a pair of roles, related to the action a and resource r associated with *sysauth*. Then, it creates two new system authorizations in order to grant to the just created roles the positive and negative authorizations to do the action a on the resource r. Finally, it removes the authorization *sysauth* and replaces it with adequate role authorizations in order to preserve the semantics of the policy.

3.8 Implementation and Experiment

We implemented a prototype for the evaluation of the behavior of the techniques presented in this paper. The prototype consists of three Java modules that invoke the services of the Semantic Web tools to manage *Policy Incompatibility*, *Policy Minimization*, and *Separation of Duty*. The prototype uses the API provided by the OWL API library [51] to access both the ontology and the

SWRL rules. Since there are no freely available large datasets of real security policies, we chose to test our prototype against policies built according to an interpretation of the data in bibliographic databases. We used randomly selected subsets of *PubMed Central*[2] (PMC), well known in the medical sciences, and DBLP[3], well known in the computer science community.

Each of them provides a rich set of attributes and relationships that represent real and extensive social networks. PMC has rich information about journals, with a description of editorships and the funding of papers. DBLP has a rich description of conferences. The two databases support different experiments.

We created instances of principals in the following way: (a) for each author or editor, we add a *SingleIdentity*, (b) for each group of authors that have written a paper together, we create a *Group* containing the identities, (c) for each group of editors of a conference or a journal, we create a *Group* containing the identities, (d) for each journal issue (and conference for the DBLP case), we create an "*editor*" *Role* and an "*author*" *Role*. We then created the following authorizations with identities as principals: (a) for each paper author we create the authorizations to *read* and *write* the paper and the negative authorization to *review* the paper, (b) for each editor of the issue of the journal containing the paper we add the authorizations to *read* and *review* the paper and the negative authorization to *write* the paper, (c) for each author that receives funding from the same grant that funded the paper, we add a negative authorization to *review* the paper and an authorization to *read* the paper. We then have authorizations with groups as principals: (d) for each group containing all the authors belonging to the institution to which the author is affiliated, we add the authorization to *read* the paper, (e) for each member of the editorial board of the journal that published the paper we add the authorizations to *read* and *review* the paper and the negative authorization to *write* the paper, (f) for the group of authors of the paper, we add the authorizations to *read* and *write* the paper and the negative authorization to *review* the paper (these are redundant authorizations), (g) for the group of editors of the paper we add the authorizations to *read* and *review* the paper and the negative authorization to *write* the paper (these are also redundant authorizations).

With this model we are able to create a rich set of authorizations that has a clear motivation and is associated with the structure of a concrete application. The conflicts detected would correspond to anomalies that relate with possible conflicts of interest.

We have run *Policy Incompatibility* and *Policy Minimization* services on random samples of the PMC dataset, while we have run the *Separation of Duty satisfiability* service on random samples of the DBLP dataset, enforcing SoD constraints on author and editor roles for the same conference/journal issue. To better evaluate the performance of our technique, we used two different reasoners to enforce the *Separation of Duty*: *HermiT* [104] and *Pellet* [107]. HermiT is based on a novel *hypertableaux* calculus, which provides a particularly efficient

[2]http://www.ncbi.nlm.nih.gov/pmc/
[3]http://dblp.uni-trier.de/

reasoning. Pellet is based on the *tableaux* algorithms developed for expressive DL.

3.8.1 Experimental results

Experiments have been run on a PC with two Intel Xeon 2.0GHz/L3-4MB processors, 8GB RAM, four 1-Tbyte disks and Linux operating system. The results of the experiments are reported in the following graphs. Each observation is the average of the execution of ten runs.

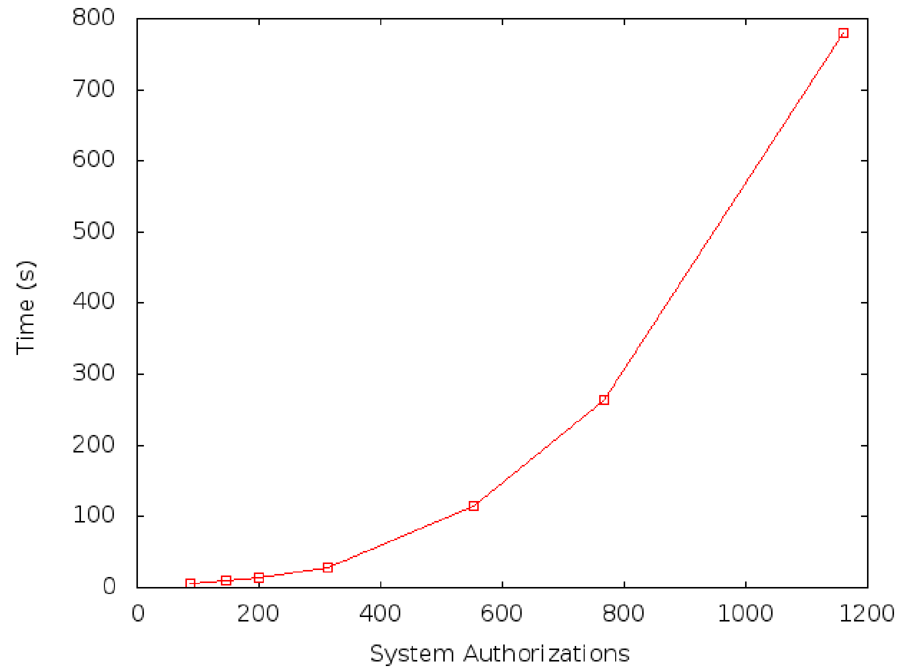
The first set of experiments aimed at evaluating how the performance of the *Policy Incompatibility* service evolves with the increase in the number of authorizations. Figure 3.9(a) reports the observed performance. It is clear that this solution is applicable for policies with a relatively small number of authorizations. For large policies, the simplicity in the definition of the verification rules is associated with a large computational cost.

The second set of experiments aimed at evaluating the response time of the *Policy Minimization* service with the increase of the number of authorizations. Figure 3.9(b) reports the results of these experiments, which exhibit response times for the same policies that are significantly larger than those observed in the previous experiments. The specific testcase is characterized by a large number of redundant authorizations and in general the analysis required for the identification of redundant rules requires to produce a large number of derivations. As presented in [81], policies are often unnecessarily complicated due to redundancies and for the cases typically considered in these analyses the number of authorizations is relatively small and this makes the approach presented in Section 3.6 acceptable. However, for the largest policies considered in our experiments, a significant benefit can be obtained from the use of tools, like the reasoners able to efficiently process DL structures, which require a different and more complicated representation of the problem.

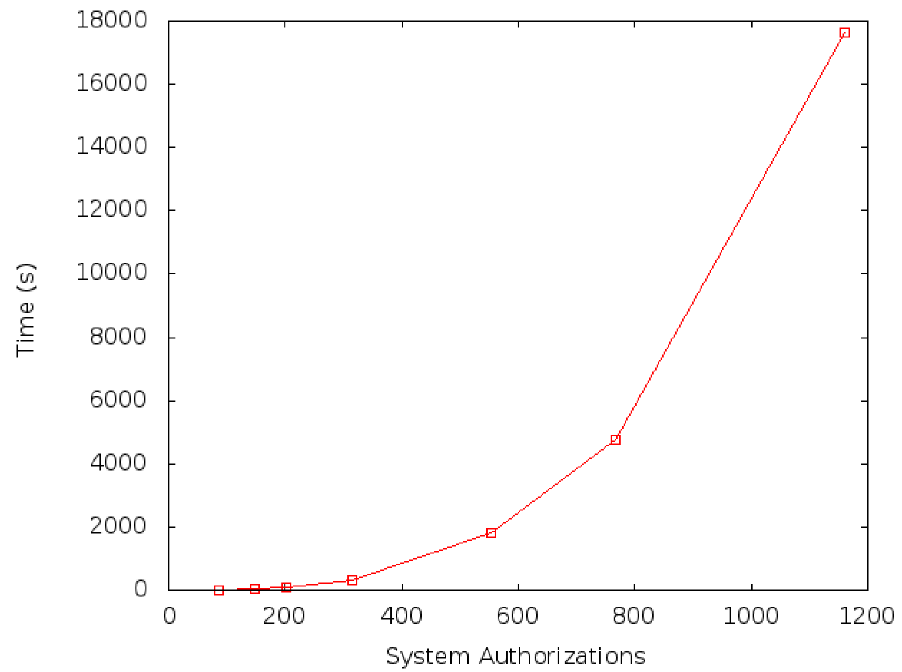
The third experiment aimed at evaluating the performance of the *Separation of Duty* service, considering the use of two reasoners. We were also interested in analyzing the behaviour of the pruning phase, to verify its impact on the performance. As depicted in Figure 3.9(c), both *HermiT* and *Pellet* offer adequate performance for up to 3000 role authorizations. *HermiT* offered better performance and we restricted the analysis of the performance for larger configurations only to *HermiT*.

In Figure 3.9(d) we can see how the performance evolves with the increase of the number of role authorizations. Even for extremely large policies (more than 35,000 role authorizations), the response times remain adequate for the profile of the security design activity, which operates in sessions that have long duration.

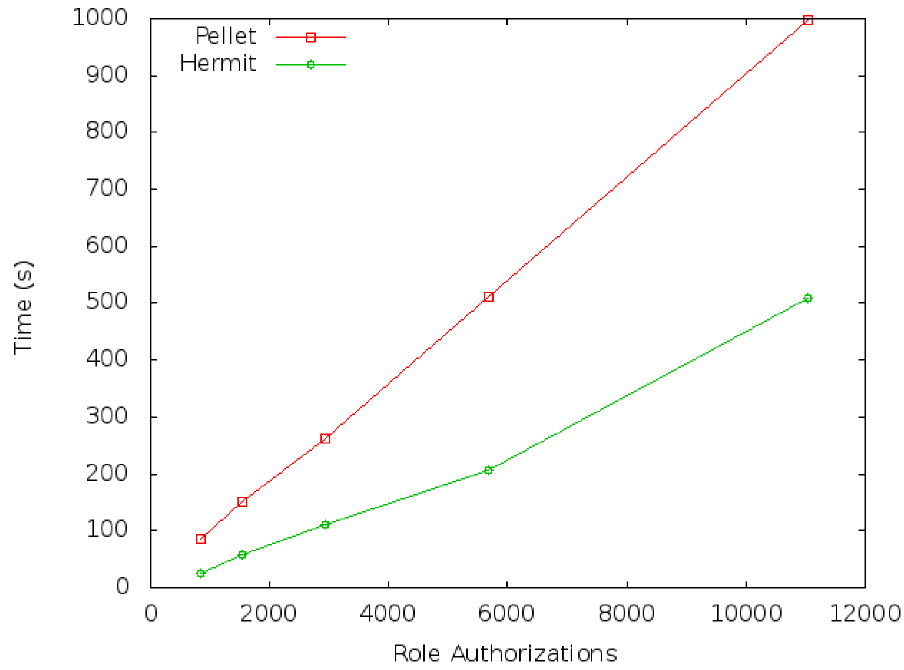
The optimization based on pruning described in Section 3.7 proves to be able to offer a significant contribution. Figure 3.9(e) shows the ratio between the time saved by the optimization and the time spent to execute the pruning phase. We omit to analyze the curve in detail; we synthetically observe that the



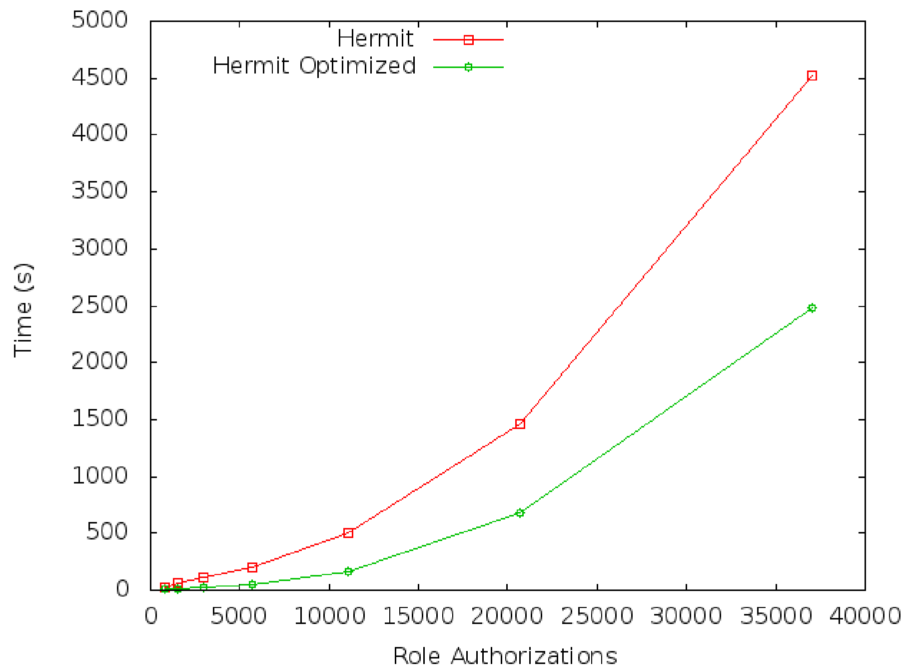
(a) Policy Incompatibility performance.



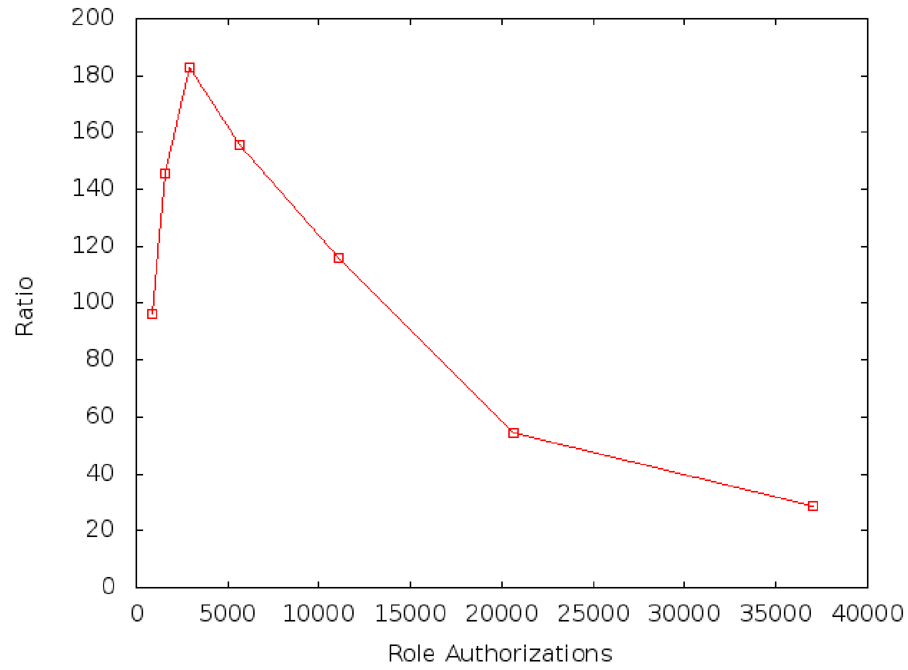
(b) Policy Minimization performance.



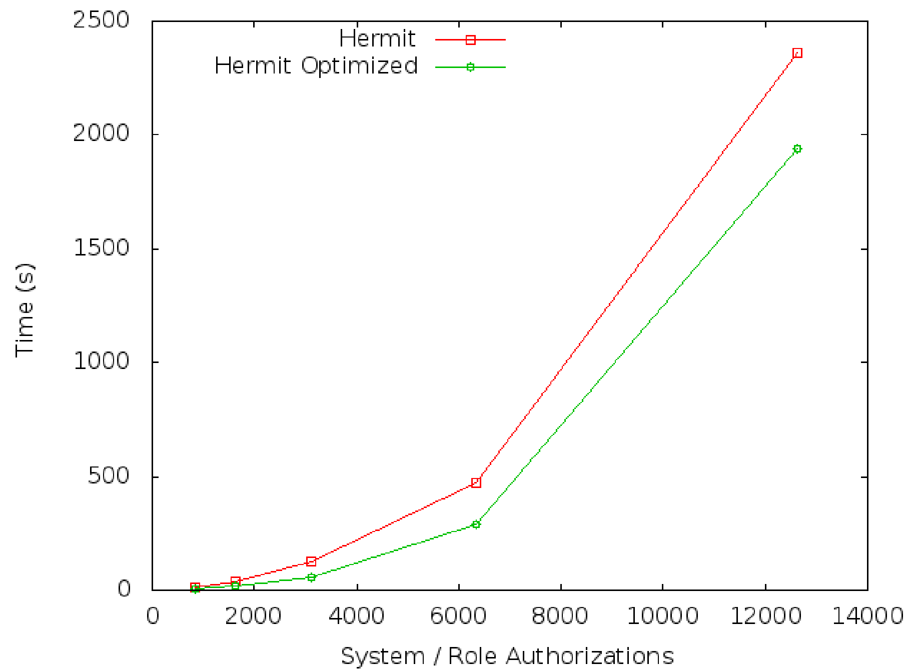
(c) Separation of Duty analysis performance.



(d) Separation of Duty analysis performance.



(e) Ratio between the overall gained time and the time spent to do the pruning phase.



(f) Policy Incompatibility as Separation of Duty.

ratio remains always above 30 and reaches values near to 200, offering robust support to the utility of this optimization.

As discussed in Section 3.7, the approach used for the management of *Separation of Duty* constraints can be adapted to solve the *Policy Incompatibility* problem. Figure 3.9(f) reports the performance observed using this approach, offering a remarkable performance increase compared to the normal approach depicted in Figure 3.9(a).

4

Policy Minimization

Access control policies used in real systems are often unnecessarily large due to redundancy. Since the size of the policy is one of the main factors that determine the cost of managing the security configuration of a system, minimizing the size of a policy can ease the management of the policy itself and can reduce the cost of the management process. Furthermore the size of a policy influences the performance of the access control system, and thus minimizing the size of the policy can improve the access control system performance [74, 119].

Although *redundancy* seems a natural and quite simple concept, providing a formal definition of it in case of access control policies is not trivial for several reasons. One of the reasons is that we may have to deal with conflicts between authorizations that may influence the result of our redundancy detection process. Another reason is that at a certain point during our redundancy detection process we may consider as redundant a subset of the authorizations in the policy, and the choice of the authorizations to effectively tag as redundant may influence the redundancy of the others. In the following, we are going to survey several definitions of redundancy that were presented in the literature. Al-Shaer et al. in [1–3] define a rule r as *redundant* iff there are other rules that produce the same actions as r, such that the removal of r does not affect the security policy. A similar definition can be found in the work of Kolovski et al. [67], which defines a policy element as *redundant* if its removal does not change the final behavior of the policy. Also Liu et al. [74] and Yuan et al. [119] agree with the definition of redundancy given above by saying that a rule is redundant iff its removal does not influence at all the behavior of the security policy. All these authors give a very similar definition of *redundancy* (which we call *basic definition* in the following) and we can consider all the definitions as equivalent to saying that an authorization (or a rule) is *redundant* iff it does not affect

the behavior of the access control mechanisms (i.e., its presence or absence in a policy passes unnoticed). The main problem of this definition is that, although it is very simple and intuitive, it often lacks in precision and flexibility: the definition of behavior of a policy is usually not specified in a formal way and it is only defined in terms of a specific and concrete access control system. Hu et al. [57] try to overcome this problem by providing a more formal definition, saying that a rule r is redundant in case the *authorization space* (which is a collection of access requests to which a policy element is applicable) derived from the policy that contains r is the same as the one derived from the policy without r.

Although the definition given above seems clear, it presents subtle side-effects that we show with a simple example. Let P be a policy composed by the authorizations a_1, a_2, a_3 where a_1 produces exactly the same effect as the combination of authorizations a_2 and a_3.

A first problem is that the *basic definition* of *redundancy* is not invariant w.r.t. the decisions of our redundancy-removal process. This fact means that a rule may be considered redundant at a certain point in time during the redundancy-removal process, and lose this property at a later time. For instance, if we consider the policy P, the redundant authorizations are a_1, a_2, a_3 (i.e., the complete policy), but if we remove authorization a_3, then a_1 is not redundant anymore. This means that we cannot safely remove sets of redundant authorizations and the definition cannot be applied looking only at the starting policy state. The definition of *redundancy* should take into account all the decisions taken during the redundancy-removal process.

Furthermore, by iteratively removing authorizations that satisfy the *basic definition*, we usually do not obtain a unique solution. Rather, we can obtain several equivalent policies with different number of authorizations. For instance, in our example both policies $\{a_1\}$ and $\{a_2, a_3\}$ are equivalent and redundancy-free, but they have different size. The fact that the *basic definition* of *redundancy* does not take into account the number of authorizations in the final policy can be surprising, since the main motivation behind the development of redundancy detection techniques is the improvement of access control mechanisms' performance, which primarily depends on the size of the access control policies. This aspect of the redundancy problem is important, because an effective redundancy-removal process should always aim at computing the minimum redundancy-free version of the given policy, not limiting the goal to the identification of one of the redundancy-free versions.

An underlying assumption in all the previous definitions is that the only way on which we can act on a policy is by removing redundant authorizations. This assumption introduces some limitations in the search for performance improvements. For instance, if P is a policy with 6 authorizations, there may exist another policy P' that is equivalent to P but it contains only 4 authorizations that were not in P. In this case, we should prefer P' over P because it can lead to an improvement in system performance. We consider as another aspect of the *redundancy* problem the computation of the policy that models the behavior of the system with the minimum number of authorizations.

It is then obvious that the *basic definition* of *redundancy* is not enough for handling this issue effectively. There is the need of one or more definitions for several aspects of the redundancy problem that take into account the size of the resulting policy, and the dependency between actions performed at different steps of the redundancy-removal process.

In this Chapter we present a formalization of the redundancy problem in access control policies that considers three different ways in which a security administrator can act on a policy containing redundancy. In the first approach, the administrator can compute an equivalent policy that does not contain redundancy anymore, i.e., she computes an *irreducible policy*. However, given a certain policy there usually are several irreducible versions of the same policy. Given the fact that the number of authorizations of the policy is one of the major factors that influence the management cost of the policy itself, in the second approach the security administrator identifies among the *irreducible policies* obtained by the original policy one with the minimum number of authorizations, which is the *minimum irreducible policy*. With the third approach, the security administrator may be interested in computing the representation of the access control system with the minimum number of authorizations, i.e., the *minimum policy*.

4.1 Model

Due to the complexity of the *minimization problem* in the following we will use a simplified version of the IT Security metamodel presented in Section 2.2.2 containing the following entities:

- **Principals**: represent users and groups of principals. Each *Principal* may contain one or more other *Principals* and this fact is represented by the function *contains*:*Principal*$\rightarrow 2^{Principal}$. The function *contains+*: *Principal*$\rightarrow 2^{Principal}$ is the transitive closure of *contains*.

- **Actions**: represent the actions that users can execute. Each *Action* may be composed by one or more *Actions* and this fact is represented by the function *composed*:*Action*$\rightarrow 2^{Action}$. The function *composed+*: *Action*$\rightarrow 2^{Action}$ is the transitive closure of *composed*.

- **Resources**: represent the resources on which users can act. Each *Resource* may contain one or more *Resources* and this fact is represented by the function *containsResources*:*Resource*$\rightarrow 2^{Resource}$. The function *containsResources+*:*Resource*$\rightarrow 2^{Resource}$ is the transitive closure of *containsResources*.

An instance $\mathcal{M}$ of our model is a list of *Principals*, *Actions* and *Resources* with the associated functions[1]. $\preceq_P$, $\preceq_A$ and $\preceq_R$ are partial orders defined over *Principals*, *Actions* and *Resources* respectively. For instance, given two

[1]The hierarchies defined in $\mathcal{M}$ must be acyclic.

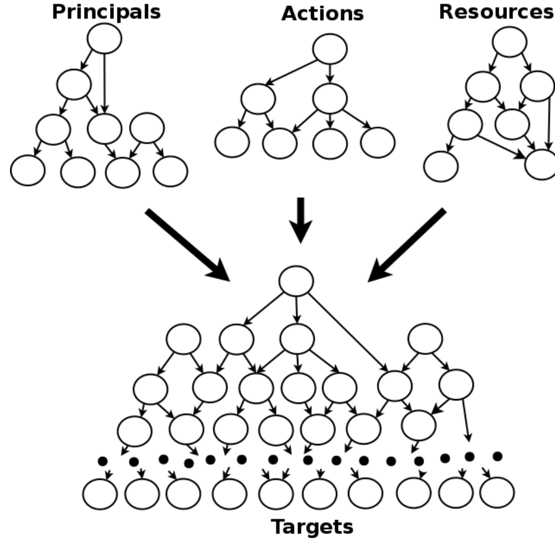


Figure 4.1: Targets Hierarchy.

principals p_1 and p_2 we say that $p_1 \preceq_P p_2$ iff $p_1 \in$*contains+*$(p_2) \cup \{p_2\}$. The definition of $\preceq_A$ and $\preceq_R$ can be obtained in a similar way (the fact that $\preceq_P$, $\preceq_A$ and $\preceq_R$ are partial orders is enforced by the fact that the hierarchies over *Principals*, *Actions* and *Resources* are acyclic). We say that an element of one of the hierarchies in $\mathcal{M}$ is *primitive* iff it is a leaf of the hierarchy (e.g., a principal p is a primitive element iff *contains*$(p) = \emptyset$).

We assume that the assignment of permissions to users can be derived from the system, e.g. in Role-Based Access Control (RBAC) [101] the user-permission assignment matrix can be directly computed from the user-role assignment and the role-permission assignment matrices.

The basic element about which we can express access control decisions is called a *Target*, and it is defined in the following way:

Definition 1. Target: *a target consists of a* Principal p*, an* Action a *and a* Resource r*. We represent a target as a triple* $< p, a, r >$*. We say that a target* $< p, a, r >$ *is* primitive *iff* p*,* a *and* r *are primitive elements.*

Given two targets $t_1 =< p_1, a_1, r_1 >$ and $t_2 =< p_2, a_2, r_2 >$, we say that t_1 implies t_2, denoted by $t_1 \preceq_T t_2$, iff $p_1 \preceq_P p_2 \wedge a_1 \preceq_A a_2 \wedge r_1 \preceq_R r_2$. The partial order $\preceq_T$ defines the *Target Hierarchy* $TH_{\mathcal{M}}$, shown in Figure 4.1. Primitive targets are the leafs of the hierarchy.

Security administrators can define access rights on the targets by means of authorizations, which are defined in the following way:

Definition 2. Authorization*: An authorization consists of a triple composed by a set of* Principals P*, a set of* Actions A *and a set of* Resources R*. Each authorization has a sign* s *that can be* $+$ *or* $-$*. It is used in order to state,*

respectively, whether an authorization is positive *(i.e., it grants the permission to do something) or* negative *(i.e., it denies the permission to do something). We graphically represent an authorization in the following way:* $< s, P, A, R >$.

An authorization $auth =< s, P, A, R >$ defined over the model $\mathcal{M}$ is associated in an unique way to a set of *Targets* $T_{auth} = P \times A \times R$ on which the authorization acts. Without loss of generality, we can express access control decisions only in terms of primitive targets (and thus we consider only primitive targets when we compute the set T_{auth}). Given an authorization, the following functions can be used in order to obtain the elements contained in it:

- *principals*:*Authorization*$\rightarrow 2^{Principal}$ retrieves the set of *Principals* involved in the authorization,
- *actions*:*Authorization*$\rightarrow 2^{Action}$ retrieves the set of *Actions* involved in the authorization,
- *resources*:*Authorization*$\rightarrow 2^{Resource}$ retrieves the set of *Resources* involved in the authorization,
- *sign*:*Authorization*$\rightarrow \{+, -\}$ retrieves the *sign* of the authorization.

Given a set of authorizations Δ and an authorization $a \in \Delta$, the *region* defined by a over Δ is $R_a^\Delta = \{a' \in \Delta | T_a \cap T_{a'} = \emptyset\}$. The region of an authorization a contains all the authorizations that may interact with a. We can define a hierarchy over authorizations in the following way: given two authorizations $auth_1$ and $auth_2$, we say that $auth_1$ is dominated by $auth_2$, i.e., $auth_1 \preceq auth_2$, iff $T_{auth_1} \subseteq T_{auth_2}$.

Example 2. *Figure 4.2 shows a graphical representation of a set of authorizations applied over a set of targets. This example will be used in the next sections as running example. The graph is composed by two different components (a) the authorizations, represented by squares labeled with a sign, and (b) the targets, represented by circles (this set of targets represent a part of the target hierarchy shown in Figure 4.1). Edges between authorizations represent the* $\preceq$ *ordering relation, whereas edges between targets represent the* $\preceq_T$ *ordering relation. In this representation, given an authorization auth the set* T_{auth} *is defined by all the nodes in the target hierarchy reachable from the node representing auth. For instance, if we consider the authorization* $A2$ *then* T_{A2} *is the set* $\{T0, T1, T6\}$. *The region defined by the authorization* $A2$ *is* $\{A1, A2, A3, A4\}$.

Let *auth* be an authorization. *auth* specifies an access control decision on the targets on which it can act (represented by the sign $sign(auth)$), and we represent this fact by saying that *auth* assigns a label $sign(auth)$ to the targets in T_{auth}. We represent the basic access control decision by means of the concept of *Privilege*, which represents a *Target* labeled with a sign s, defined in the following way:

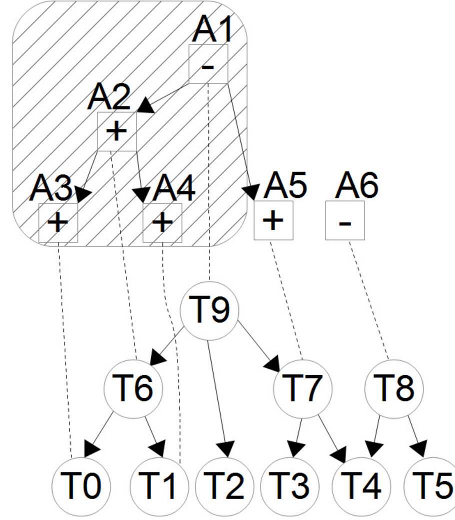


Figure 4.2: Graphical representation of authorizations

Definition 3. Privilege: *A privilege consists of a target* $t = < p, a, r >$ *and a sign* $s \in \{+, -\}$ *(inherited from an authorization). We graphically represent a privilege in the following way:* $< s, p, a, r >$*. If the sign* s *is* $+$ *then the privilege represents the fact that the* Principal p *is allowed to do the* Action a *on the* Resource r*. On the contrary, if the sign* s *is* $-$ *then the privilege represents the fact that the* Principal p *is not allowed to do the* Action a *on the* Resource r*. We say that a privilege* $< s, p, a, r >$ *is* primitive*, i.e., it does not imply other privileges, iff the target* $< p, a, r >$ *is primitive.*

Given a privilege the following functions can be used in order to obtain the elements of the model contained in it:

- *principal*:*Privilege*→*Principal* retrieves the *Principal* involved in the privilege,
- *action*:*Privilege*→*Action* retrieves the *Action* involved in the privilege,
- *resource*:*Privilege*→*Resource* retrieves the *Resource* involved in the privilege,
- *sign*:*Privilege*→ $\{+, -\}$ retrieves the *sign* of the privilege.

Each authorization *auth*, thus, grants a set of privileges (i.e., a set of labeled targets). More formally, we say that each authorization *auth* associates a sign s to the targets in the set T_{auth}. The procedure shown in Algorithm 1 can be used to compute the set of privileges associated with an authorization. We denote the set of privileges associated with the authorization *auth* as *privileges(auth)*.

Due to the fact that authorizations and privileges have a sign, we may have conflicts between different authorizations and privileges in the same policy P.

Algorithm 1: *privileges* procedure for authorizations

Input : Authorization $auth$
Output: $Privileges$
begin
  $Privileges = \emptyset$;
  for $p \in principals(auth),\ a \in actions(auth),\ r \in resources(auth)$ **do**
    for $p' \in contains+(p) \bigcup \{p\},\ a' \in composed+(a) \bigcup \{a\},$
    $r' \in containsResources(r) \bigcup \{r\}$ **do**
      if $contains(p') = \emptyset \wedge composed(a') = \emptyset \wedge containsResources(r') = \emptyset$ **then**
        $Privileges = Privileges \bigcup \{< sign(auth), p', a', r' >\}$;

Several approaches have been proposed in the literature for the solution of conflicts between authorizations [**?**, 16, 76]. Our approach is not tied to any particular conflict resolution strategy, indeed it can be used with any conflict resolution strategy that satisfies certain requirements, explained in detail in Section 4.1.1. Let $\mathcal{M}$ be a model, let $P_{\mathcal{M}}$, $A_{\mathcal{M}}$ and $R_{\mathcal{M}}$ be the set of principals, actions and resources associated with the model, and let Δ be the set of all possible authorizations that can be defined over $\mathcal{M}$. We represent a conflict resolution strategy as a function $\psi : P_{\mathcal{M}} \times A_{\mathcal{M}} \times R_{\mathcal{M}} \times \mathbb{P}(\Delta) \rightarrow \Delta \cup \{\perp\}$ that takes as input a target $< p, a, r >$ and a set of authorizations A and returns the authorization in A that determines the privilege granted by the set A w.r.t. the $< p, a, r >$ (if no authorization in Δ can be applied to $< p, a, r >$ then ψ returns $\perp$).

Let Δ be a set of authorizations, we denote with *privileges*(Δ, ψ) the set of privileges granted by Δ w.r.t. the conflict resolution strategy ψ. In this case *privileges*(Δ, ψ) can be computed by using the procedure shown in Algorithm 2 that returns the set of labeled targets associated with Δ w.r.t. the strategy ψ. Given a set of authorizations Δ, a conflict resolution strategy ψ and an authorization $auth$, we say that Δ dominates $auth$ w.r.t. ψ, denoted by $auth \preceq \Delta$, iff $privileges(auth) \subseteq privileges(\Delta, \psi)$.

Algorithm 2: *privileges* procedure for sets of authorizations

Input : Set of authorizations Δ, Conflict resolution strategy ψ
Output: $Privileges$
begin
  $Privileges = \emptyset$;
  for $auth \in \Delta$ **do**
    for $p \in principals(auth),\ a \in actions(auth),\ r \in resources(auth)$ **do**
      for $p' \in contains+(p) \bigcup \{p\},\ a' \in composed+(a) \bigcup \{a\},$
      $r' \in containsResources(r) \bigcup \{r\}$ **do**
        if $contains(p') = \emptyset \wedge composed(a') = \emptyset \wedge containsResources(r') = \emptyset$
        then
          $a = \psi(p', a', r', \Delta)$;
          $Privileges = Privileges \bigcup \{< sign(a), p', a', r' >\}$;

Definition 4. Policy: *A policy consists of a set of authorizations* Δ *and a conflict resolution strategy* ψ. *We represent a policy as a pair* $< \Delta, \psi >$.

Let $P =< \Delta, \psi >$ be a policy, we denote with $P' \subset P$ a policy $P' =< \Delta', \psi' >$ such that $\Delta' \subset \Delta$ and $\psi' = \psi$.

Each policy is identified by its behaviour, which is defined in the following way:

Definition 5. Behaviour of a Policy: *the behaviour of a policy P is the set of privileges granted, directly or indirectly, by the policy.*

The behaviour of a policy P can be computed using the function *privileges* presented in Algorithm 2, thus we can check whether two different policies are equivalent by checking whether they enable the same set of privileges. The behaviour of the policy models how the real access control system behaves. Given the fact that the same behaviour can be modeled by means of several different policies, the equivalence relation between policies is defined in the following way:

Definition 6. Equivalence of Policies: *Two policies $P =< \Delta, \psi >$ and $P' =< \Delta', \psi' >$ are equivalent, $P \equiv P'$, iff they have the same behaviour, i.e., iff* $\text{privileges}(\Delta, \psi) = \text{privileges}(\Delta', \psi')$.

Let P be a policy defined over the model $\mathcal{M}$. There exists always a policy P' expressed only in terms of primitive elements of $\mathcal{M}$ which is equivalent to P and such that $|P| = |P'|$. In order to obtain P' we can proceed in the following way: we define an authorization $< s, Pr', A', R' > \in P'$ for each authorization $< s, Pr, A, R > \in P$ where Pr', A' and R' contains only the primitive elements that can be derived from the elements in Pr, A, and R respectively. The fact that $P \equiv P'$ follows trivially from the definition of equivalence and from the *privilege* procedure. In the following we can thus consider, without loss of generality, policies expressed only in terms of primitive elements.

4.1.1 Conflict resolution strategies

Let $\mathcal{M}$ be a model, let $P_{\mathcal{M}}$, $A_{\mathcal{M}}$ and $R_{\mathcal{M}}$ be the set of principals, actions and resources associated with the model and let Δ be the set of all possible authorizations that can be defined over $\mathcal{M}$.

A conflict resolution strategy is a function $\psi : P_{\mathcal{M}} \times A_{\mathcal{M}} \times R_{\mathcal{M}} \times \mathbb{P}(\Delta) \rightarrow \Delta \cup \{\perp\}$. ψ takes as input a set of authorizations Δ and a target $< p, a, r >$, and it returns as output the authorization $auth \in \Delta$ that is applied over $< p, a, r >$ according to the strategy (if no authorization in Δ can be applied to $< p, a, r >$ then ψ returns $\perp$).

A conflict resolution strategy ψ may satisfy one or more of the following properties:

- **Polynomiality:** we say that ψ is polynomial (in the size of Δ and of the target hierarchy) iff there is an algorithm that implements ψ which is in **P**,

- **Completeness:** we say that ψ is complete iff for any possible set of authorizations Δ and for any target $t =< p, a, r >$ if $\exists auth \in \Delta :< p, a, r >\in T_{auth}$ then $\psi(p, a, r, \Delta) \neq \perp$,
- **Monotonicity:** we say that ψ is *monotone* iff for any possible set of authorizations Δ and for any target $t =< p, a, r >$ then $\psi(p, a, r, \Delta) = \psi(p, a, r, \Delta \cup \{auth\})$ holds for any authorization $auth$ such that $< p, a, r > \notin T_{auth}$. A monotone conflict resolution strategy is one that produces a result that depends only on the authorizations in Δ that are related with $< p, a, r >$ (i.e., adding unrelevant authorizations do not change the outcome of the strategy).

We say that ψ is a *valid* conflict resolution strategy for our framework, iff ψ satisfies the three properties above.

For instance, we can represent the *Denial takes precedence* strategy in the following way:

$$\psi_{DTP}(p, a, r, \Delta) = \begin{cases} a_i & \textit{if } < +, p, a, r > \in \textit{privileges}(a_i) \wedge \\ & \nexists a_j \in \Delta \setminus \{a_i\} : \\ & < -, p, a, r > \in \textit{privileges}(a_j) \\ a_i & \textit{if } < -, p, a, r > \in \textit{privileges}(a_i) \\ \perp & \textit{otherwise} \end{cases}$$

It is easy to see that the *Denial takes precedence* strategy satisfies all the requirements stated above.

Example 3. *If we consider only the authorizations A5 and A6 of Figure 4.2, we can define the sets (a)* $T_{A5} = \{T3, T4, T7\}$*, and (b)* $T_{A6} = \{T4, T5, T8\}$*. We can notice that there is an intersection between* T_{A5} *and* T_{A6}*, and the two authorizations have a different sign. The application of the strategy Denial takes precedence means that for the target T4 (i.e., the intersection) will be applied the authorization A6, the negative one.*

The *Most Specific Wins* strategy can be represented by the following function ψ_{MSW}:

$$\psi_{MSW}(p, a, r, \Delta) = \begin{cases} a_i & \textit{if } < p, a, r > \in T_{a_i} \wedge \\ & \nexists a_j \in \Delta \setminus \{a_i\} : \\ & (< p, a, r > \in T_{a_j} \wedge a_j \preceq a_i) \\ \perp & \textit{if } \exists a_i, a_j \in \Delta :< p, a, r > \in T_{a_i} \\ & \wedge < p, a, r > \in T_{a_j} \wedge \\ & a_i \not\preceq a_j \wedge a_j \not\preceq a_i \\ \perp & \textit{otherwise} \end{cases}$$

It is easy to see that ψ_{MSW} is not complete because it may return $\perp$ also in case there are authorizations in Δ that can be applied to the target $< p, a, r >$ but these authorizations are not comparable.

Example 4. *If we consider only the authorizations A2 and A3 of Figure 4.2, we can define the sets (a)* $T_{A2} = \{T0, T1, T6\}$*, and (b)* $T_{A3} = \{T0\}$*. We can notice that* $A3 \preceq A2$ *because* $T_{A3} \subseteq T_{A2}$*. For instance, for the target T0 we can apply two authorizations A2 and A3. The Most Specific Wins strategy chooses to apply the authorization A3 because it is more specific than A2.*

Although our framework is independent from a specific conflict resolution strategy, for concreteness in the the running example and in Section 4.4 we consider the conflict resolution strategy ψ that applies the *Most Specific Wins* and *Denial takes precedence* criteria. This strategy can be modeled in the following way:

$$\psi(p, a, r, \Delta) = \begin{cases} \psi_{MSW}(p, a, r, \Delta) & \textit{if } \psi_{MSW}(p, a, r, \Delta) \neq \perp \\ \psi_{DTP}(p, a, r, \Delta) & \textit{otherwise} \end{cases}$$

ψ satisfies all the requirements stated above.

4.2 Redundancy

Sometimes real policies contain redundancy [81], for several reasons caused by the evolution of security policies during time. Although the concept of *redundancy* is easy to understand, defining it formally is not trivial, especially, as in our case, when we consider conflicts.

A simple definition of *redundancy* may be the following: "Given a policy P, we can define an authorization *auth* as *redundant* when it does not add anything to the behaviour of P.". The process of removing redundancy is called *redundancy-removal* process. The problem of this definition is that redundancy is not invariant, i.e., it depends on the sequences of decisions taken during the redundancy-removal process. Indeed, by solving a conflict between authorizations or by removing an authorization from the policy, we may change the set of authorizations that are redundant. As a consequence, the sequence of decisions taken during the redundancy-removal process in order to achieve a redundancy-free equivalent policy, may lead to significantly different results in terms of size of the final policy.

In the following we try to formalize the *redundancy* problem. The proof of all the theorems are given in the appendix.

We start by defining the concept of *redundancy condition*, which refines the definition given above.

Definition 7. Redundancy Condition*: An authorization* $auth \in P$ *satisfies the* redundancy condition *w.r.t. the policy* $P = <\Delta, \psi>$ *iff* $\forall <p, a, r> \in T_{auth}$:
$sign(\psi(p, a, r, R^{\Delta}_{auth} \setminus \{auth\})) = sign(\psi(p, a, r, R^{\Delta}_{auth})) \wedge$
$\psi(p, a, r, R^{\Delta}_{auth} \setminus \{auth\}) \neq \perp$.

In other words, *auth* satisfies the redundancy condition w.r.t. P iff its presence or absence does not influence the access control decision for any possible target $<p, a, r>$ on which it can act.

Example 5. *For instance, authorization $A2$ in Figure 4.2 satisfies the redundancy condition. The set T_{A2} contains the targets $T0$, $T1$ and $T6$ and for each of these targets the presence of $A2$ does not influence the behavior of the policy. Hence, we can safely remove $A2$.*

Given a policy P, we can remove one by one all the authorizations that satisfy the redundancy condition without changing the behaviour of the policy. However, by removing authorizations from P we may change the redundancy condition for other authorizations. We model this phenomenon with the concept of *sequence of reductions*.

Definition 8. Sequence of Reductions (SoR): *let P_0 be a policy, and let $P_1, \ldots, P_n$ be a sequence of policies. The sequence $P_0, P_1, \ldots, P_n$ is a* sequence of reductions *iff it satisfies the following requirements:*

- $P_n \subset P_{n-1} \subset \ldots \subset P_1 \subset P_0$,
- *for each $i \in \{1, \ldots, n\}$, $P_{i+1} = P_i \setminus \{\gamma_i\}$ and γ_i is an authorization that satisfies the* redundancy condition *w.r.t. P_i,*
- *there are no authorizations in P_n that satisfy the redundancy condition w.r.t. P_n.*

We say that P_0 is the begin *of the sequence, and that P_n is the end of the sequence. We say that the reduction from P_i to P_{i+1} is a* step *in the sequence.*

Due to the fact that the authorization removed at each step satisfies the redundancy condition, the behaviour of the policy does not change along the sequence of reduction, as stated in Theorem 1.

Theorem 1. *Let $P_0, \ldots, P_n$ be a sequence of reductions. All the policies in the sequence have the same behaviour, i.e., they are equivalent.*

An interesting property of *SoRs* is that once we reach a point P_i in a sequence $P_0, \ldots, P_n$ then all the authorizations that satisfy the redundancy condition at P_i satisfy the redundancy condition also at the following steps of the sequence, as stated in Theorem 2. This means that, although the choice of the authorization to remove may influence the final outcome, the order in which we remove these authorizations does not influence the result (i.e., what influences the size of the final policy is only $\Gamma = \{\gamma_0, \ldots, \gamma_{n-1}\}$ and not the order in the SoR in which we remove authorizations that satisfy the redundancy condition).

Theorem 2. *Let $P_0, P_1, \ldots, P_n$ be a sequence of reductions. Let $auth \in P_0$ be an authorization such that there exists a value $j \in \{1, \ldots, n\}$ for which $\forall j' < j : auth \in P_{j'}$ and auth satisfies the redundancy condition w.r.t. P_j. In this case auth satisfies the redundancy condition w.r.t. all $P_i \cup \{auth\}$ where $j < i \leq n$.*

Another important property of *SoRs* is stated in Theorem 3. The theorem says that whenever there are two equivalent policies P and P' such that P' is a subset of P, then there is always at least a sequence of reductions that starts

from P and passes through P'. This means that we can reduce the problem of finding a redundancy-free version of a policy P to the one of finding an adequate *SoR* starting from P.

Theorem 3. *Let P be a security policy and let $P' \subset P$ be another policy such that P and P' are equivalent. There is always at least one sequence of reductions of the form $P, \ldots, P', \ldots, P_n$.*

We can now define the concept of redundancy-free policy, called *irreducible policy*, in the following way:

Definition 9. Irreducible Policy (IP)*: A policy P is an irreducible version of the policy P' iff it does not contain any authorization that satisfies the redundancy condition w.r.t. P and $P \subseteq P'$. This is equivalent to say that there is a sequence of reductions from P' to P, i.e., $P', \ldots, P$.*

From Theorem 3 follows that in case we reach a policy P in a SoR such that there are no authorizations satisfying the redundancy condition w.r.t. P, then we can soundly say that an equivalent policy $P' \subset P$ does not exist. Hence given a policy P, we can compute an equivalent irreducible version P' in a simple, although sometimes inefficient, way. Initially let $P' = P$, then we iterate over all the authorizations in P', and we check for each authorization $a \in P'$ whether it satisfies the redundancy condition w.r.t. P' or not, if this is the case then we remove the authorization (i.e., $P' = P' \setminus \{a\}$). We iterate the above procedure until no more authorizations satisfy the redundancy condition. It is easy to see that this algorithm is polynomial w.r.t. the number of authorizations in P.

Checking whether a certain policy P is irreducible or not is the Irreducible Policy Problem (IPP) and it can be solved in **P-TIME**, as demonstrated by Theorem 4.

Definition 10. Irreducible Policy Problem*: Given a policy P, checking whether P is irreducible is called Irreducible Policy Problem (IPP).*

Theorem 4. *The* IPP *is in* **P**.

Example 6. *Figure 4.3(a) shows an irreducible version of the policy in Figure 4.2. The new policy was obtained by removing the authorization $A2$ which satisfies the redundancy condition, as shown in Example 5.*

Given a policy P, several different irreducible versions of it may exist. In order to improve the performance of access control mechanisms, a possible solution is to compute the irreducible version of P with the minimum number of authorizations. We call this policy a *Minimum Irreducible Policy.*

Definition 11. Minimum Irreducible Policy (MIP)*: A policy P is a minimum irreducible version of the policy P' iff a policy P'' does not exist such that P'' is irreducible, P'' is equivalent to P', $|P''| < |P|$, $P \subseteq P'$ and $P'' \subseteq P'$. In an equivalent way, we can say that P is a minimum irreducible version of the policy P' iff P is the end of the longest SoR that can be computed starting from P'.*

The problem of checking whether a certain policy P is a minimum irreducible policy with respect to the original policy P' is called Minimum Irreducible Policy Problem.

Definition 12. Minimum Irreducible Policy Problem: *Given two policies P and P', checking whether P is a minimum irreducible policy with respect to P' is called Minimum Irreducible Policy Problem (MIPP).*

We are more interested in the the problem of finding a minimum irreducible version of a given policy P (which is the search problem associated with the MIPP). Since the associated decision problem is **coNP-complete**, the search problem is **NP-hard**.

Theorem 5. *The* MIPP *is* **coNP-complete**.

Example 7. *Although in Example 5, we have shown a redundancy-free version of the policy in Figure 4.2, the resulting policy (obtained by removing $A2$) was not the minimum irreducible one. Indeed a smaller irreducible policy can be computed by removing $A3$ and $A4$ instead of $A2$. This policy, shown in Figure 4.3(b), is the minimum irreducible one, since no smaller irreducible policies exist.*

Given the fact that the behaviour of an access control system may be modeled by means of different equivalent policies, security administrators may be interested in computing the policy with the minimum number of authorizations that models the system, i.e., the *minimum policy*.

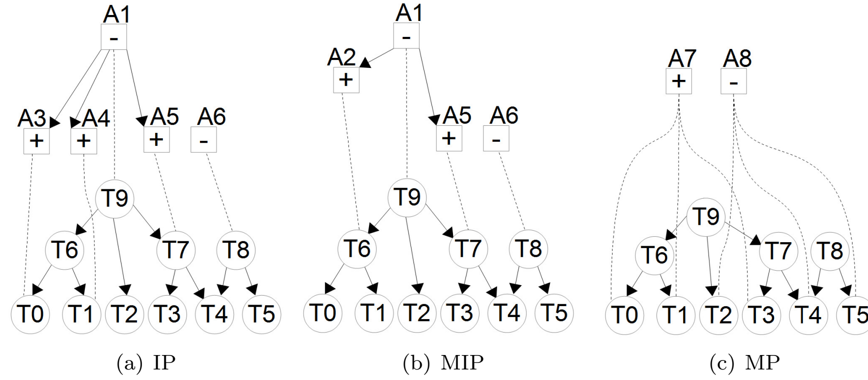
Definition 13. Minimum Policy (MP): *A policy P is said to be minimum iff an equivalent policy P' does not exist such that $|P'| < |P|$.*

Given a policy P we can compute an irreducible policy P' equivalent to P by removing authorizations that satisfy the redundancy condition. However, in order to compute the minimum policy P'' we may have to define new authorizations that do not exist in P or remove authorizations in P with the only constraint that the resulting policy P'' must have the same behaviour as the original one. The problem of checking whether a policy is minimum or not can be defined in the following way:

Definition 14. Minimum Policy Problem: *Given a policy P, checking whether P is minimum is called Minimum Policy Problem (MPP).*

Theorem 6. *The* MPP *is* **coNP-complete**.

We are more interested in the the problem of computing a minimum version P' of a given policy P (which is the search problem associated with the MPP). Since the associated decision problem is **coNP-complete**, the search problem is **NP-hard**. It is worth pointing out that depending on the given policy P, the search problems associated with *MIPP* and *MPP* may do not have a unique solution, i.e., there may be several different *minimum irreducible* and *minimum* versions of the same policy P (all with the same size).



(a) IP (b) MIP (c) MP

Figure 4.3: Redundancy-free policies

Example 8. *Although the policy computed in Example 7, is the minimum irreducible policy, a smaller policy exists and it is shown in Figure 4.3(c)*[2]*. This policy is the minimum policy.*

4.3 Implementation

In this section, we present techniques for solving the *MIPP* and *MPP* problems. We ignore the *IPP* because several algorithms were proposed in the literature to solve this problem [8,57,67,74,119] (and also because any solution for *MIPP* is a solution for *IPP*). In the following we show how *MIPP* and *MPP* can be mapped on the *Weighted SAT* problem. The *Weighted SAT* problem is an extension of the SAT problem, and it is defined as follows:

Definition 15. Weighted SAT*: Given a set of variables U and a collection C of clauses over U, computing, in case it exists, the truth assignment $t : U \rightarrow \{0,1\}$ which satisfies C and minimizes a certain cost function $\sum_{u_i \in U'} k_i * u_i$, where $U' \subseteq U$ and $k_i \in \Re$, is called the* **Weighted Satisfiability Problem***.*

Section 4.3.1 presents how the *MIPP* can be mapped to the *Weighted SAT* problem, whereas in Section 4.3.2 we present a heuristic technique for solving *MIPP*. In Section 4.3.3 we present how the *MPP* can be mapped to the *Weighted SAT* problem. In Section 4.3.4 we present an algorithm that uses a heuristic approach to solve *MPP*. Section 4.4 presents some experimental results.

4.3.1 MIPP to Weighted SAT

We can map the *MIPP* to the *Weighted SAT* problem, and we can use efficient *SAT*-solvers in order to identify the minimum irreducible version of the input policy. Let $P =< \Delta, \psi >$ be the input policy, we want to produce a policy

[2]We assume that the parent node is equal to the union of its children.

$P' =< \Delta', \psi >$ such that $\Delta' \subseteq \Delta$ and P' is a minimum irreducible version of P. We denote with G the set *privileges*(Δ, ψ).

We define a variable a_i for each authorization $a_i \in \Delta$. Due to the fact that the correspondence between variables and authorizations is clear, in the following we switch freely between the two notations (e.g., sometimes a_i may refer to an authorization and sometimes it may refer to the variable associated with that authorization). For simplicity's sake, we do not present formulae in CNF (this is not a problem because it is always possible to translate a formula to an equivalent one in CNF).

The cost function $min \sum_{a_i \in \Delta} a_i$ aims at minimizing the number of authorizations in the resulting policy (and thus it guarantees that the resulting policy is the minimal irreducible one).

We still have to handle conflicts and the conflict resolution strategy in our *Weighted SAT* instance. Our approach considers a general conflict resolution strategy ψ. Let $t =< p, a, r >$ be a target defined in our model, we denote with K_t the set containing all the authorizations that act on the target t (i.e., $K_t = \{a_i \in \Delta | t \in T_{a_i}\}$). Given a target $t =< p, a, r >$ and a privilege $p =< s, p, a, r >$ we define $C(p)$ in following way:

$$C(p) = \bigvee_{X \in \mathbb{P}(K_t) \wedge sign(\psi(p,a,r,X)) = sign(p)} \left(\bigwedge_{a_i \in X} a_i \wedge \bigwedge_{a_i \in K_j \setminus X} \neg a_i \right)$$

For each privilege $p_j \in G$ we add a constraint in the form $C(p_j)$, this constraint enumerates all the possible combinations of authorizations that effectively grant the privilege p_j. By enforcing these constraints we ensure that the behavior of the resulting policy is equivalent to the behaviour of P. An authorization $a_i \in \Delta$ is in Δ' iff the variable associated with a_i is set to one in the result of the *Weighted SAT* problem. It is easy to see that the result of the *Weighted SAT* problem produces a *Minimum Irreducible Policy.*

4.3.2 Heuristic Algorithm for MIPP

As described above the MIPP is a **coNP-complete** problem. Hence, in this Section we propose a heuristic algorithm that allows to find, in an efficient way, an irreducible policy close to one of the exact solutions.

Our approach, shown in Algorithm 3, is based on an iterative process. Let $P =< \Delta, \psi >$ be the initial policy. At each iteration, the algorithm iterates over the authorizations and for each authorization a computes its region R_a^Δ by means of the *computeRegion* procedure. Then the algorithm checks whether the authorization satisfies the redundancy condition w.r.t. P by means of the *isRemovable* procedure, which takes as input (a) the selected authorization, (b) the region, and (c) the resolution strategy. If the authorization a satisfies the redundancy condition, then the algorithm removes it from the policy. The algorithm ends when no more authorizations satisfy the redundancy condition w.r.t. P; this means that the algorithm has reached a fixed point, and from Theorem 3 further reductions are not possible.

Algorithm 3: Heuristic Algorithm for MIPP

```
Input  : Policy P, Conflict Resolution Strategy CRS
Output: MIP P
begin
    bool removed;
    repeat
        removed = false;
        for a ∈ P do
            List region = computeRegion(a, P);
            if isRemovable(a, region, CRS) then
                P.removeAuthorization(a);
                removed = true;
    until removed;
```

In order to improve the performance of the algorithm, we can optimize the *computeRegion* and the *isRemovable* procedures, by taking into account a specific conflict resolution strategy. For instance, if we consider the conflict resolution strategy presented in Section 4.1.1 we can tune both procedures in the following way. Let a be an authorization and P be the policy, the *computeRegion* procedure can produce a restricted region $R'_a = A_a \cup D_a \cup I_a$ where A_a is the set of direct ancestors of a (i.e., $A_a = \{a' \in \Delta | a \preceq a' \wedge \nexists a'' \in \Delta \setminus \{a, a'\} : (a \preceq a'' \wedge a'' \preceq a')\}$), D_a is the set of direct descendants of a (i.e., $D_a = \{a' \in \Delta | a' \preceq a \wedge \nexists a'' \in \Delta \setminus \{a, a'\} : (a'' \preceq a \wedge a' \preceq a'')\}$) and I_a is the set of most specific authorizations that have an intersection (at the target level) with a (i.e., $I_a = \{a' \in \Delta | a' \npreceq a \wedge a \npreceq a' \wedge T_a \cap T_{a'} \neq \emptyset \wedge \nexists a'' \in P \setminus \{a, a'\} : (a'' \preceq a' \wedge T_a \cap T_{a''} \neq \emptyset)\}$). The region R'_a is a subset of the region R_a and usually R'_a is quite smaller than R_a. In the same way we can improve the performance of the *isRemovable* procedure by leveraging the characteristics of the conflict resolution strategy, e.g, first we may check whether the *Most Specific Wins* criterion can be applied; if this is the case, then we need only to find the most specific authorization, otherwise we know that if at least one negative authorization is in the region, then the sign of the resulting authorization will be $-$ otherwise $+$.

4.3.3 MPP to Weighted SAT

In order to obtain an exact solution to the *MPP* problem, we can map it to the *Weighted SAT* problem, and we can use efficient *SAT*-solvers in order to identify the minimum version of the input policy. Let $P = < \Delta, \psi >$ be the input policy, we want to produce a policy $P' = < \Delta', \psi >$ where P' is a minimum version of P. We denote with G the set *privileges*(Δ, ψ). Let k be $|G|$ and n be $|\Delta|$.

We define the following variables:

- $auth_1, \ldots, auth_n$ where each $auth_i$ represents an authorization,
- $p_{i,j}$ for each $i \in [1, n]$, $j \in [1, k]$. Each variable $p_{i,j}$ represents the fact that the privilege $p_j \in G$ is assigned to the authorization a_i,

The cost function $min \ \sum_{j \in [1,n]} auth_j$ aims at minimizing the number of authorizations in the resulting policy. We define the following clauses that aim

Algorithm 4: Heuristic Algorithm for MPP

```
Input  : Policy P =< Δ, ψ >
Output: Policy P' =< Δ', ψ >
begin
    Δ' = ∅;
    Pr = privileges(Δ, ψ);
    List pList = new List(Pr);
    while pList ≠ ∅ do
        p = pList[0];
        pList = pList.remove(p);
        added = false;
        for a ∈ P' do
            if isCompatible(p, a, Pr) then
                added = true;
                T = privileges(a);
                if p ∉ T then
                    principals(a) = principals(a) ∪ principal(p);
                    actions(a) = actions(a) ∪ action(p);
                    resources(a) = resources(a) ∪ resource(p);
                    break;
        if !added then
            a =< {principal(p)}, {action(p)}, {resource(p)} >;
            Δ' = Δ' ∪ {a};
```

at ensuring that the resulting policy is equivalent to the initial policy:

- For each privilege p_j we define a clause in the form $\bigvee_{i\in[1,n]} p_{i,j}$. The clause aims at enforcing the fact that the privilege has to be assigned to at least one authorization.
- For each authorization $auth_i$ we define the clauses in the form $\neg p_{i,j} \vee auth_i$ for each $j \in [1,k]$, which enforce the fact that if one of the privileges is assigned to an authorization, then the variable associated with the authorization is enabled.
- For each pair of privileges p_l and p_m such that $p_l \neq p_m$ and $sign(p_l) \neq sign(p_m)$ and $privileges(\{<\{principal(p_l),\ principal(p_m)\}\ ,\{action(p_l),\ action(p_m)\}\ ,\{resource(p_l),resource(p_m)\}\ >\}) \not\subseteq Z$ we define the clauses in the form $\overline{p_{i,l}} \vee \overline{p_{i,m}}$ for each $i \in [1,n]$ that enforce the fact that two incompatible privileges cannot be assigned to the same authorization.

The resulting policy P' is obtained by creating the authorizations $auth_i$ where $i \in [1,n]$ to which at least one privilege has been assigned to.

4.3.4 Heuristic Algorithm for MPP

We defined a heuristic algorithm, shown in Algorithm 4, which iteratively tries to build the minimum policy by adding a privilege at a time. First the algorithm computes the set of privileges granted by the policy P given as input. The algorithm tries to group together compatible privileges. In order to do this it iterates over the privileges, and at each iteration it tries to add the current privilege to the already existing authorizations. If no compatible authorization

Original Size	Final Size	Delta Time %
43	22	25.58%
70	29	10.19%
88	25	3.78%
151	47	60.74%
184	58	58.86%
219	76	83.08%
253	92	90.64%
295	98	96.88%
360	112	99.19%
451	149	99.49%

Table 4.1: Comparison between the two MIPP methods

exists, it creates a new authorization. The *isCompatible* procedure takes as input a privilege p, an authorization a and the set of privileges Pr granted by the original policy and checks whether a and p can be merged or not, i.e., $\textit{isCompatible}(p,a,Pr) = \textit{privileges}(\{<\textit{principal}(p) \cup \textit{principals}(a), \textit{action}(p) \cup \textit{actions}(a), \textit{resource}(p) \cup \textit{resources}(a)>\}) \subseteq Pr \wedge \textit{sign}(p) = \textit{sign}(a)$.

4.4 Experimental Results

We implemented a prototype for the evaluation of the performance of the techniques presented in this paper. The prototype consists of a Java module that invokes the implementation of the four approaches presented above. The exact solutions use the SAT4J[3] SAT solver. Since there are no freely available large datasets of real security policies, we chose to test our prototype against policies built according to an interpretation of the data in bibliographic databases. We used randomly selected subsets of *PubMed Central*[4] (PMC) which provides a rich set of attributes and relationships that represent a real and extensive social network. It has rich information about journals, with a description of editorships and the funding of papers.

Given a random sample of the *PMC* database, we built an instance of our model in the following way: (a) for each author or editor, we create a *principal*, (b) for each group of authors that have written a paper together, we create a *principal* containing the *principals* associated with the authors, (c) for each group of editors of a conference or a journal, we create a *principal* containing the *principals* associated with the groups (d) we defined three different actions *read*, *write* and *review*. We then created the following authorizations: (a) for each paper author we create the authorizations to *read* and *write* the paper and the negative authorization to *review* the paper, (b) for each editor of the issue of the journal containing the paper we add the authorizations to *read* and *review* the

[3]http://www.sat4j.org - Sat4j library for Java
[4]http://www.ncbi.nlm.nih.gov/pmc/

Original Size	Final Size	Delta Time %
6	4	93.67%
12	4	83.94%
19	4	76.70%
23	8	99.86%
28	9	99.95%
32	10	99.98%
45	11	99.99%
62	13	99.99%

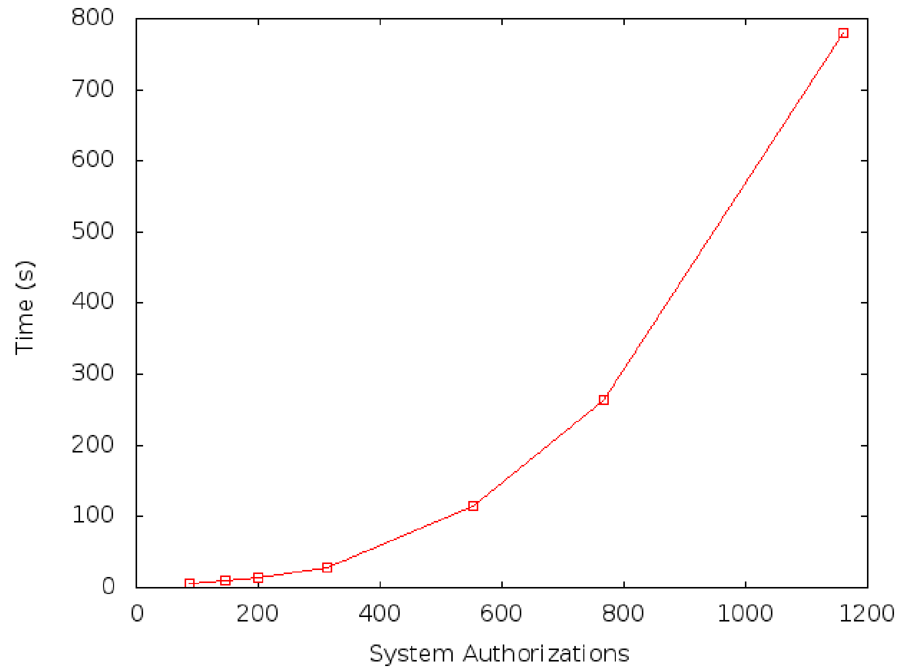
Table 4.2: Comparison between the two MPP methods

paper and the negative authorization to *write* the paper, (c) for each author that receives funding from the same grant that funded the paper, we add a negative authorization to *review* the paper and an authorization to *read* the paper, (d) for each group representing the institution to which the author is affiliated, we add the authorization to *read* the paper, (e) for each group representing the editorial board of the journal that published the paper we add the authorizations to *read* and *review* the paper and the negative authorization to *write* the paper. Experiments have been run on a PC with two Intel Xeon 2.0GHz/L3-4MB processors, 12GB RAM, four 1-Tbyte disks and Linux operating system. Each observation is the average of the execution of ten runs.

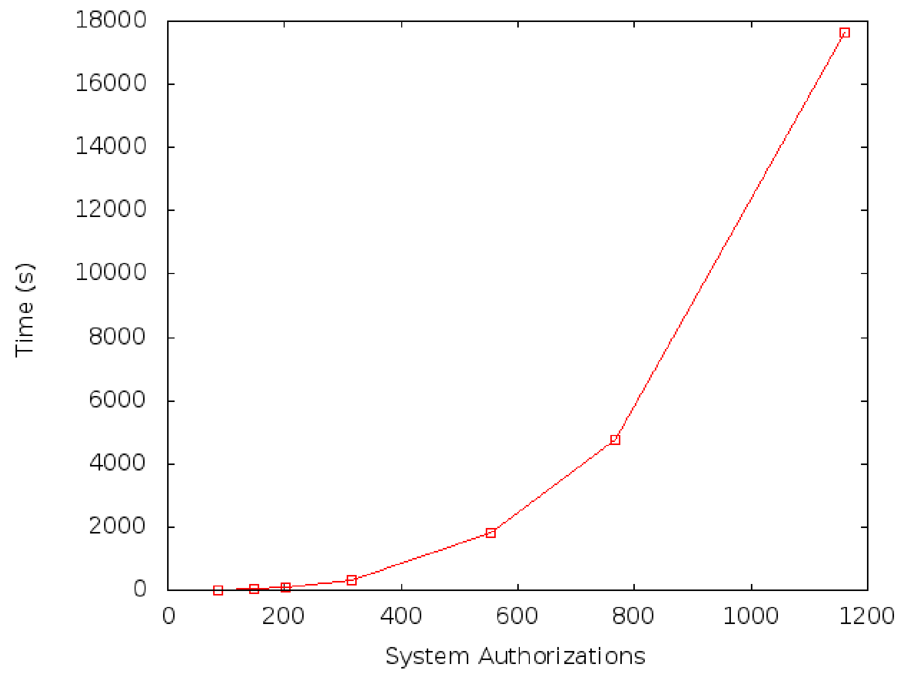
The results of the exact approach for the MIPP problem are shown in Figure 4.4(a), whereas the performance of the heuristic algorithm are shown in Figure 4.4(b). Table 4.1 shows a detailed comparison between the results of the two approaches. The heuristic approach is always able to identify the exact solution and the savings in execution time range from 3.78% to 99.49%.

The results of the exact approach for the MPP problem are shown in Figure 4.4(c), whereas the performance of the heuristic algorithm are shown in Figure 4.4(c). Table 4.2 shows a detailed comparison between the results of the two approaches. Also in this case the heuristic approach is always able to identify the exact solution and the savings in execution time range from 76.70% to 99.99%. However the MIPP exact solution scales better than the MPP one.

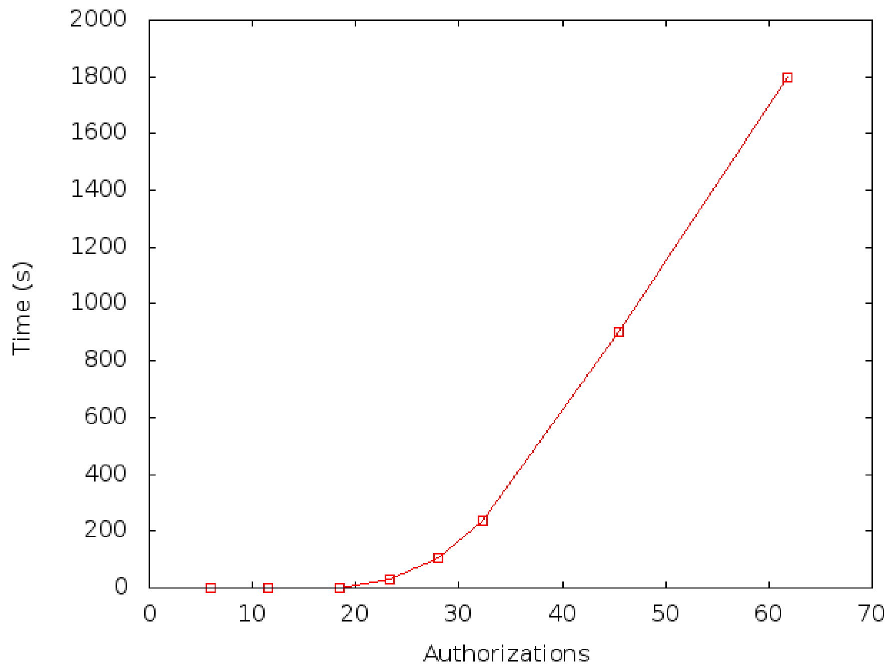
Empirical results show that both heuristic algorithms could be a good approximation of the exact ones. They allow the analysis of real policies with a good precision and with good response time. Figure 4.4(e) and Figure 4.4(f) compare the performance of the two heuristic algorithms, both in terms of execution time and reduction. The execution time of the two algorithms is very similar with policy with a size lower than 5000 authorizations. After that threshold the MIPP algorithm performs better. On the other hand, the MPP algorithm allows to compute smaller policies than the MIPP. The MPP allows to obtain a policy which is usually 20% smaller than the policies resulting from the MIPP, but its execution may take more time.



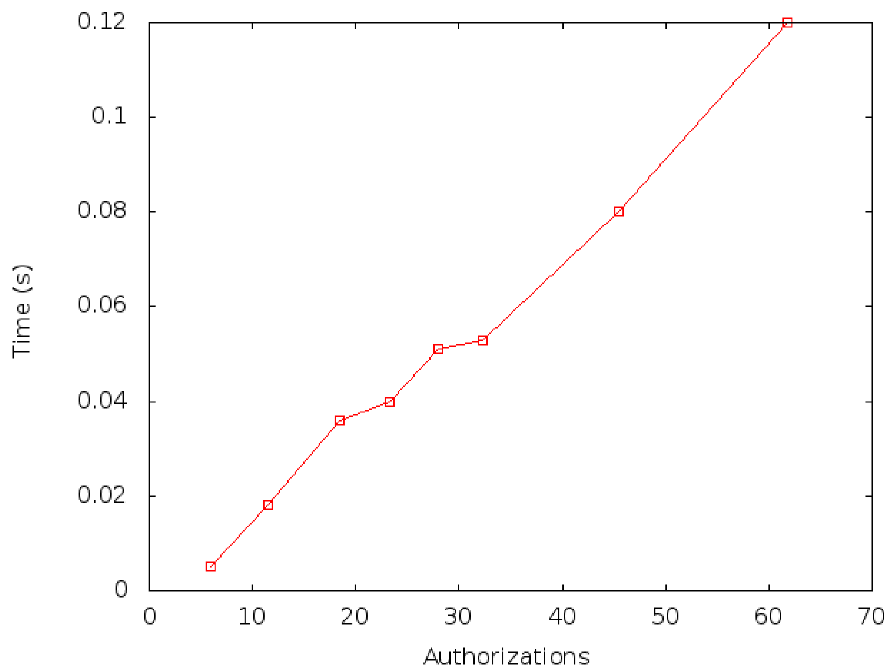
(a) SAT MIPP.



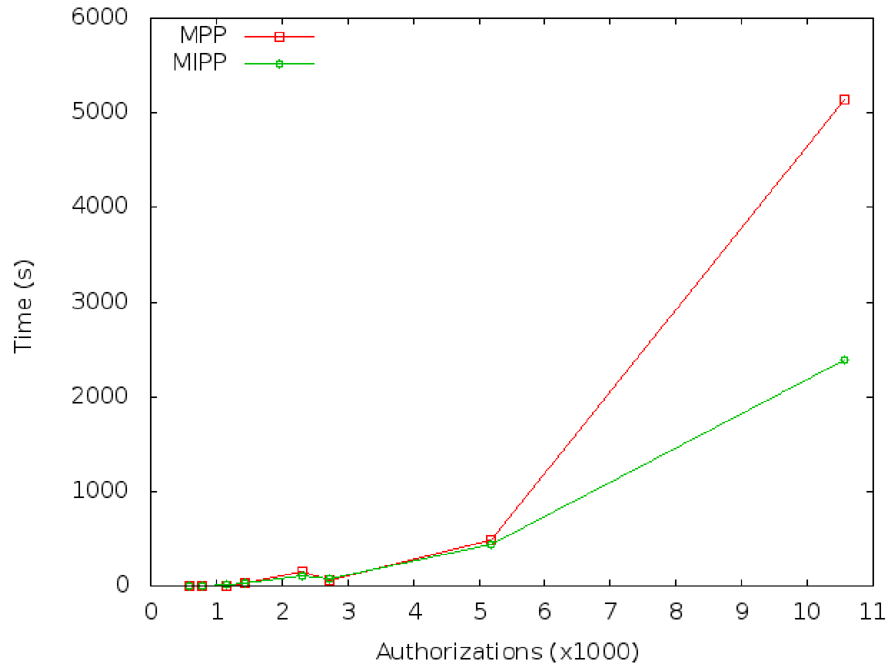
(b) Heuristic MIPP.



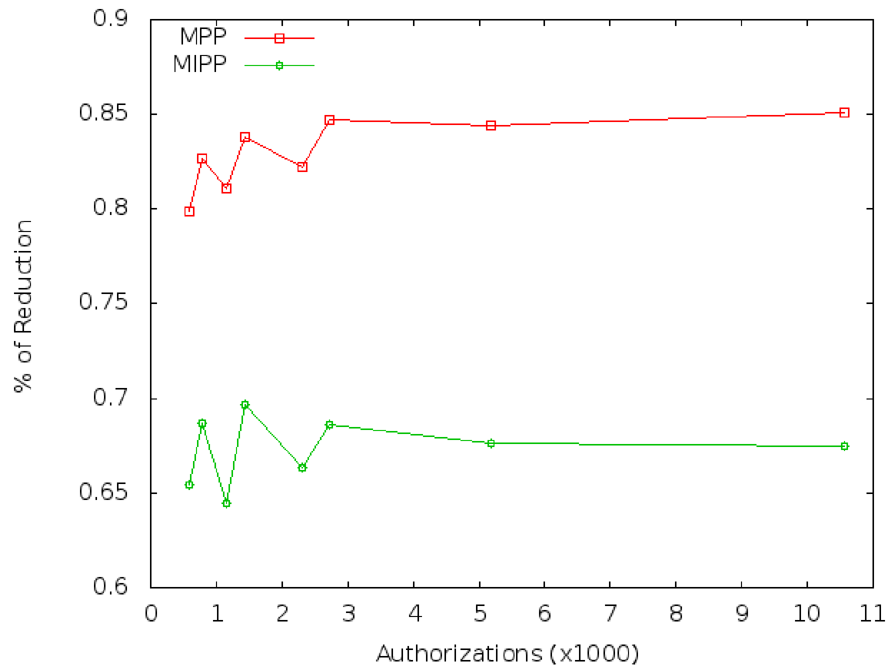
(c) SAT MPP.



(d) Heuristic MPP.



(e) Time Comparison.



(f) Size Comparison.

5

Policy Refinement

The *Policy Refinement* is the process that translates the policy p_u into an equivalent policy p_l represented at a lower abstraction layer. In this context, "p_u equivalent to p_l" means that the enforcement of p_l (the policy at the lower level) guarantees the enforcement of p_u (the policy at the higher level).

Therefore, a set of high-level business requirement can be refined to a set of configurations for the security controls available in the network/distributed system (where there are the resources) to protect, that, once deployed, satisfy the requirements.

5.1 Overall workflow

Figure 5.1 shows the the workflow for the generation from IT policies of abstract authentication and authorization configurations. It is composed by the following steps:

- Retrieve the IT Policy and the landscape
- Determine the Enrichment Type
- Enrich the IT Policy
- Choose and Apply a Refinement Strategy
- Generate the Abstract Configurations

Process the IT Policy and the landscape. The first step is to retrieve the IT Policy and the landscape from the PoSecCo repository. Internal representations of the IT Policy and Landscape descriptions are created. The

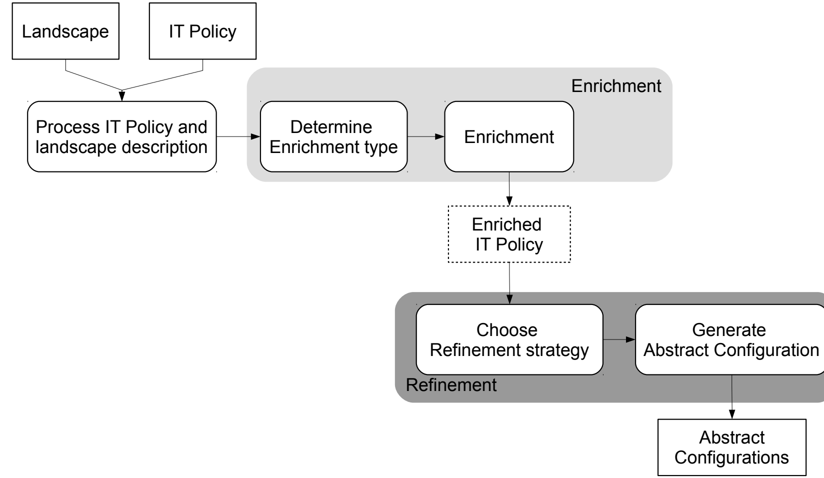


Figure 5.1: Application Configuration Service general workflow.

formats of these representations support the evaluation of queries against these descriptions.

Determine Enrichment Type. The IT Policy refers to entities that may correspond to one or more entities in the landscape as represented at the IT level in the PoSecCo Repository. Each entity may be specialized (i.e., enriched) according to its profile, as described in the landscape or as provided by the security designer in the interaction with the tool supporting this phase. For instance, ITInterfaceModels may correspond to one or more ITInterfaces in the landscape. Each ITInterface in the landscape has a particular type (HTTPMethod for web applications, WebServiceOperation for web services and EJBMethod for Enterprise Java Bean applications). In the same way, an ITResourceModel instance may correspond to several ITResources, and one of them can be an instance of a family of systems (e.g., DBMSs), associated with a specific version (e.g., MySQL 5.4).

Enrichment of Entities. This step covers the actual enrichment of the IT Policy. For the management of the access control policy, we assume that each ITSecurityRule within the IT Policy is decomposed and further enriched.

The authentication and authorization subjects, which are instances of ITPrincipal, are enriched to entities that are in use by authentication information providers. These providers include LDAP Directory Servers, Windows Authentication and Linux Pluggable Authentication Modules. In the enrichment process, human intervention is required to map IT Policy subjects to entities stored in authentication sources and to provide the location of the authentication providers. For the enrichment of interface

objects, we consider those ITInterfaceModels that refer to one or more ITInterfaces in the landscape. For instance, in the generation of access control policies for J2EE application servers the relative path of the IT-Interface corresponding to the ITResource deployed on the J2EE server needs to be computed by traversing the landscape model.

Security rules describe the authentication and authorization configuration of the system. An important component is the definition of ITActions, which the IT Policy refers to. ITActions will be enriched with concrete actions that can be performed on real systems. For example, an ITAction such as ‘access’ may be mapped to HTTP methods such as ‘GET’ and ‘POST’.

Consider Candidate Solutions and Choose. This phase represents the application of the technical refinement strategies, as discussed above. Depending on the type of element, a variety of approaches will be used. We foresee approaches that are automated, or manual or a combination of both. In Section 3 and in [92], we performed an analysis on end-user scenarios. Based on this analysis, we identified that the translation of IT Policies into abstract access control configurations (for operating system, DBMSs, Web servers and Web application servers) does not yield a large number of alternative configurations. Hence, we expect a manual approach for selecting an authorization configuration out of a set of alternatives. This is in contrast to network policies. The processing of the network policy will instead rely on the evaluation of a potentially large number of alternative configurations, with the need to apply a selection criterium to choose the “optimal” configuration among all those that would be able to satisfy the design requirements.

Generate Abstract Configurations. The outcome of the earlier step is used to compose the configuration rules of an abstract authorization configuration and an abstract authentication configuration, respectively. All the configuration rules in an abstract configuration correspond to one application/ITResource on which the abstract configuration can be deployed. Thus, when composing abstract configurations, configuration rules are grouped by the application/ITResource on which the rules are to be deployed.

5.2 Enrichment of IT Policies

The enrichment process is responsible for the introduction of new details in the description of security policies and the related IT resources. Introducing new details means adding elements to the original ontology. It can be implemented in different ways, according to different requirements.

The enrichment capabilities of the PoSecCo system are built upon a set of basic and primary techniques. These techniques cover the main enrichment

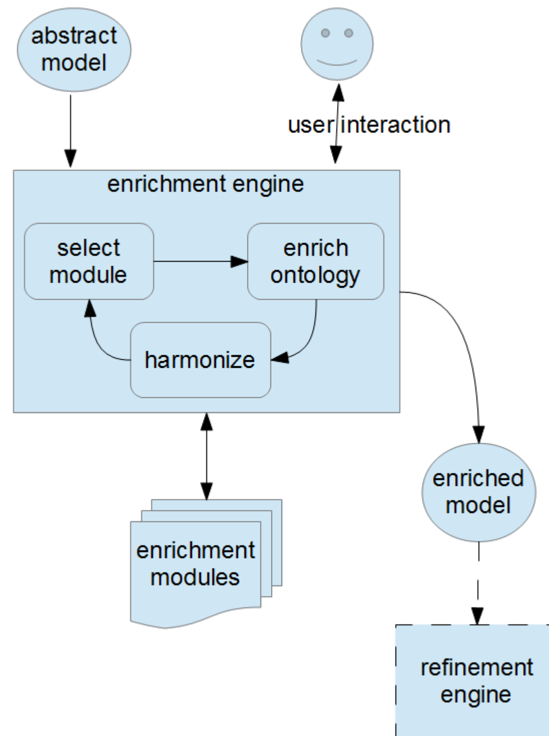


Figure 5.2: Functional schema of the enrichment engine. It selects and executes each module from the knowledge base, obtaining an enriched model ready for refinement

strategies and techniques for a quite general set of resources and the corresponding access control policies. For example we can define specific enrichment for databases or for some web servers.

If an application requires a more detailed description of specific types of resource and security policies, or when new vendor specific information is needed, the PoSecCo administrator must be able to add new knowledge and to define new ways to enrich the PoSecCo model for the new scenario. In order to support this level of flexibility the enrichment process in PoSecCo is designed in a modular way. Enrichment is guided by the *enrichment engine*. The enrichment engine is a sort of virtual machine that observes the current status of the semantic model at the IT level, and evaluates if one of the enrichment strategies is applicable. If one or more strategies are applicable, the enrichment engine executes it in an automatic or in an interactive way. The fundamental element of the knowledge base used by enrichment engine is called the *enrichment module* and is the technical implementation of one *enrichment strategy*.

So an enrichment module is a complete and self-contained description of the

actions to be carried out to execute a specific enrichment and it must provide:

- the description of the situations in which the specific enrichment can be executed;
- a way to identify the candidate targets of the enrichment, taken from the set of individuals in the model;
- the description of the operations to be executed to perform the enrichment;
- the definition of the interaction with the user required during enrichment.

In this section we will present common characteristics of enrichment modules, with a practical overview of the different enrichment approaches we use in PoSecCo. We present possible classifications of enrichment modules according to the effects they have on the IT level ontological model and to the level of interaction they have with the user of the PoSecCo tool. Finally we go in some details about the structure of each module, that is the way in which the module can be enabled, recognized as applicable on a specific set of nodes, executed and evaluated.

5.2.1 Enrichment process and enrichment modules

The enrichment process is built up of several enrichment steps. Each step adds a single element or a set of strongly interconnected elements to the ontology.

The actual execution of the whole enrichment is not rigidly defined as in a classic BPM scenario. There is no unique or predefined sequence of actions to be performed in order to obtain a refinable IT level description. Furthermore, in some situations the user can continue enrichment of some fragment of an IT model even after it becomes complete enough to enable refinement. This is the case, for example, of scenarios in which further non mandatory details about IT level resource structure can enable more efficient refinement strategies, which could not be executed on less refined versions of the model.

For these reasons, refinement usually proceeds in a user-determined order. Each enrichment module implements an enrichment and the actual sequence of actions is determined in an interactive way, by highlighting for the user the set of possible enrichment steps to be performed.

From the functional point of view, the general structure of an enrichment module passes through the following steps:

1. evaluate all the individuals in the IT level, and identifies all the nodes that are ready to be processed;
2. optionally, the user is asked to select the node he wants to enrich;
3. one or more automatic actions are executed to actually perform the desired elaboration;
4. the results of the enrichment phase are reported to the user.

So, the general structure of an enrichment module is composed of at least two parts, corresponding to steps 1 and 3:

1. an individual selection component. This can be in the form of a query like object, used to retrieve individuals. In most cases it can be a SPARQL query or a SPARQL-DL query with only one variable in the *SELECT* clause. Another possibility is to express the characteristics of the desired individuals via an ad hoc concept definition C. This will be used in an instance retrieval reasoning task to retrieve all the individuals suitable for enrichment

2. an action. This is a deduction step that adds new knowledge to the ontology. In most cases a deduction step executes custom Java code or an inference rule (like an *SWRL* rule) on the item or the items selected in the previous step.

5.2.2 Types of enrichment techniques

Depending of the kind of element that the enrichment step adds to the ontology we can have different types of enrichment techniques: *existing individual classification*, *property enrichment* and *individual creation*.

Existing individual classification In the simplest case, the detail in ontological model can be extended by specifying the actual type of some resource. Generally speaking, both the FSM ontology and IT policy ontology include one or more taxonomies, for example, the *resource hierarchy* and the *IT action hierarchy*.

resource hierarchy is used to classify elements like web servers and databases into families, from an abstract level down to a vendor-specific level. As the type of the resource becomes more specific, more characteristics are available for refinement.

action hierarchy starts from the generic actions a principal can execute on a generic resource. These are specialized down to the specific actions for a specific resource. As an example, the *SELECT* action, which can be executed only on database resources, is a specialization of the generic *READ* action.

Figure 5.3 shows a very simple case of enrichment via resource classification. In this scenario the *document_db* node is known to be an instance of *DatabaseServer*. However, like most classes in the IT resource taxonomy, *DatabaseServer* is a generic class, that is, all of its instances may be further refined to a more specific class. In this example the refinement engine can guide the user to select the right vendor specific implementation, in this case *OracleServer*. This choice can be delegated to the user or it can be taken by the system, if, at design time, we are able to specify some sufficient conditions to be classified to the specific class, and, in addition, we have all the required details about the resource to

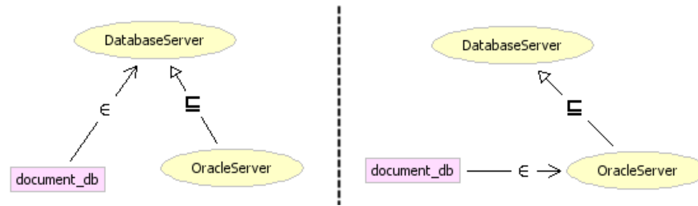


Figure 5.3: A simple example of classifying an existing database server individual

be classified. Enrichment by classification is a fundamental building block of ontological reasoning services and should be considered as an in-line enrichment. This means that, it is automatically triggered when the PoSecCo tool queries the semantic model to retrieve specific instances of the subclass.

Individual creation Creating new individuals is a possible way to enrich an existing ontology. It is necessary, if directly enriching the entity would cause a contradiction. For example, adding a property to an entity might violate an ontology constraint like property functionality. In these cases the creation of a new individual can be the only possible way to express the required enriched properties.

So, it is natural that this solution is quite similar to the approach used for policy refinement. In fact, in policy refinement the structure of the consumed IT policy and the structure of the produced abstract configuration are very different. They represent completely different elements according to the PoSecCo metamodel, and it is trivially verifiable that the identification of the two elements can make the whole model inconsistent.

Figure 5.4 shows a very simple application of the *individual creation* approach to the database scenario introduced above. The new created individual, namely *refined_document_db*, is connected to the original individual through the property *enrichmentOf*. This allows tracing of the enrichment chains, mainly

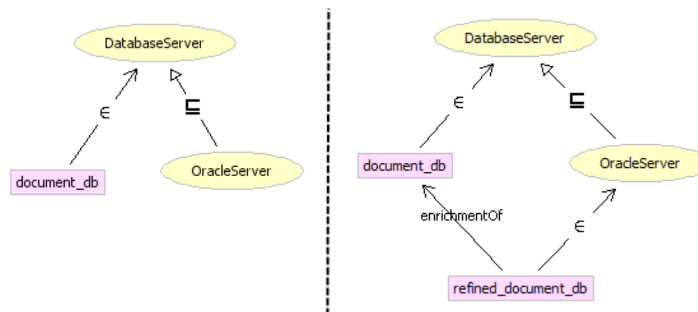


Figure 5.4: Creation of a new individual to represent enriched resource

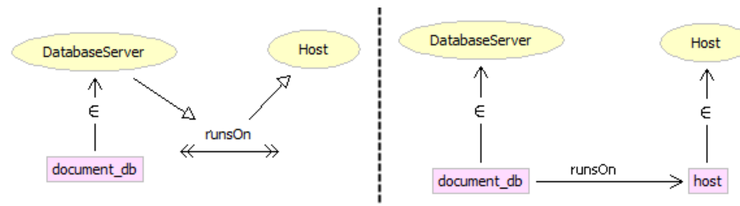


Figure 5.5: Enrichment can consist of drawing new relations between existing nodes

for auditing purposes, and can be used to prevent multiple refinements of the same individual.

Property creation After an element of a PoSecCo model is enriched it is usually in need of further details. Details can concern addresses, configurations or other metadata that may be inapplicable at a higher abstraction level.

The introduction of new metadata is a very important way to introduce new details. It can consist of the introduction of new literals or in the definition of new relations between different nodes. In fact, to remain within the decidable fragments of the *OWL* language, a strict distinction between properties and relations is required:

object properties can be used to connect different individuals and represents binary relations. We will refer to *object properties* as *relations* for the sake of clarity;

datatype properties are intended to connect individuals to literals, that is, to strings or other serializable values, including numbers, dates and other commonly used datatypes. We will refer to *datatype properties* as *properties* when no confusion arises.

Figure 5.5 shows an example of property creation through the *object property* named *runsOn*. From the domain ontology (on the left-hand side of the figure) we know that *Database Servers* run on *Hosts*. Since *document_db* is already classified as a *Database Server* the system can guide the user through the selection of the host it is running on.

If the ontology not only defines the property as being applicable to the specific type of node, but also specifies the property range, the PoSecCo tool is able to assist the user in selecting an appropriate property filler, by restricting the choices to the individuals already classified into the class of the property range.

We can distinguish between two different kinds of property creation enrichment:

strong property enrichment is available when a property is mandatory for the type of nodes we are evaluating. Such a mandatory constraint can be

expressed by defining a minimum cardinality restriction on the property, that is a necessary condition (a.k.a. a superclass) for belonging to the class of the individual.

For example, we can say that every *DatabaseServer* must run on exactly one host:

$$DatabaseServer \sqsubseteq \exists_{=1} runsOn$$

If the property enrichment is strong and the individual does not specify a value for the property, the PoSecCo tool will ask the user to define the value of the property as a mandatory enrichment step [1].

weak property enrichment is possible when no cardinality or existential restriction is available for the property. The user can be allowed to add a new filler for the relation (or equivalently the value for the property), but he is not forced to do so.

In the general case, but especially for weak property enrichments, it is a best practice to always define a maximum cardinality for every property (or relation) involved in enrichment. Maximal cardinality can be used to block the system and to avoid the request for another filler (or value) when the maximum number of fillers has already been reached.

The property creation technique requires the choice of the filler from the list of eligible nodes, or the entry of the specific value for the property. For these reasons it is unlikely to be fully automated, and is usually executed in an interactive way.

Property specialization Similarly to the individual classification case, properties can be enriched by specifying either a more specific sub-property, or a more specific sub-relation which holds between two nodes.

In the most general scenario we have:

- an individual x, which is classified as an instance of concept (class) $C2$;
- concept $C2$ is a specialization of the concept $C1$ (for example, x started as an instance of $C1$, and was classified as an instance of $C2$ in a previous individual classification enrichment step);
- we already know that $R(x, y)$ (that is x is connected to y through relation R);
- we have a set of sub-properties of R, named $R_1 \dots R_n$.

[1] please note that, for the purposes of ontology consistency checking, the lack of knowledge about a mandatory property (or equivalently a mandatory relation) is not a cause of inconsistency due to the Open World Assumption (OWA). Nevertheless the A-box of the PoSecCo models is usually assumed to be complete, so the enrichment step is mandatory

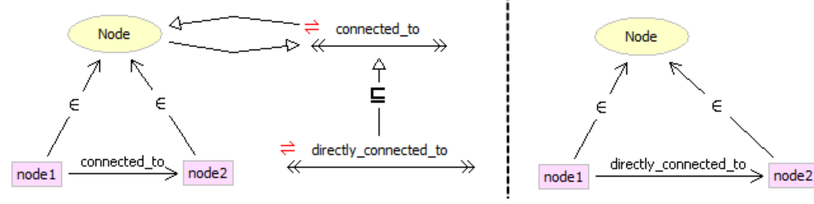


Figure 5.6: Specification of an existing relation between existing nodes

In this case the user can be asked to choose a refinement $S \in \{R_1 \ldots R_n\}$ of R and to add a link $S(x, y)$ instead of $R(x, y)$. All the elaborations which consume an R link will continue to work after this substitution, since the R link is trivially derived from the S one due to the *owl:subpropertyOf* axiom.

Property specialization suffers the same limitations as previous approaches, with the additional limitation that the conditions of mandatory refinement are not easy to express in plain *Description Logics*.

A possible approach to cover some very limited cases is to restrict the cardinalities and use existential qualifications to prove the more specific type of the property:

- $\top \sqsubseteq \exists_{\leq 1} R$ [2] states that R is functional (so all its subproperties will be functional)
- $R_i \sqsubseteq R$ defines a property hierarchy to be navigated
- $C2 \sqsubseteq \exists S$ represents the fact that all the instances of $C2$, which may be instances of $C1$ if we assume that $C2 \sqsubseteq C1$, must have at least one S relation.

With this kind of representation, if we have an individual that already has an R-image, and assuming the closure of the example model, it is quite simple to show that we can prove that $S(x, y)$ holds.

Figure 5.6 shows a simple example of enrichment, in which we can guide the user through the enrichment of the *connectedTo* link between two nodes into a more detailed type of link. According to the syntax used above, Figure 5.6 instantiates existing relation specification where $x = node1$, $y = node2$, $R = connected_to$ and $S = directly_connected_to$.

5.3 Relation with harmonization modules

As already stated in previous sections, enrichment modules can vary substantially in the kind of interaction with the user they require and the level of autonomy they are granted for automatic model enrichment. User interaction

[2] $\top$ concept denotes the full domain of disclosure, so any axiom in the form $\top \sqsubseteq C$ states that C holds for all the individuals in the model

ranges from guided manual enrichment to fully automatic enrichment which is invisible to the user.

In all of the cases, with the exception of full automatic enrichment, the system is asked to validate the resulting model. This is especially true in the case of user proposals. In fact, ontology driven model editing can avoid local and quite trivial semantic errors during the enrichment process. At the base level ontology editing support guider the user in defining correct properties between existing individuals. This can be easily achieved by allowing the user to select only the properties defined for the type of source node (using property domains) and to select the target individual from the classes compatible with the property (using property ranges in the simplest case, or by interpreting OWL restrictions). The user interface can be instructed to present to the user only the possibilities that respect constraints on the types of individuals. However these constraints must be evaluated locally on the specific fragment of the model, like the specific triple the user is adding to the knowledge base.

Nevertheless, even semi-automatic enrichment may be checked. In fact, each enrichment step can be considered as the execution of a complex rule. Each module extracts the suitable individuals, asks the user to select target individuals from the identified set, and adds some new knowledge about the individual, by introducing new nodes (as in the individual creation technique) or adding new properties to existing individuals.

However, each module is necessarily designed on the basis of local structures. Its applicability, and therefore the results of its application, is expressed in terms of a query or of a concept that covers only the structure of the nodes close to the candidate target. After the execution of the enrichment we can evaluate the global effect of the modification on the whole model. So, after both manual and semi-automatic enrichment steps, a global evaluation must be carried out to ensure, at the base level, that no inconsistency was introduced during enrichment.

In the general case only a small fragment of possible conflicts and modeling errors can be discovered by straightforward terminological reasoning. In these cases the inline reasoner is sufficient to highlight the problem. A good support for this scenario is the *explanation* capability [27,61]. Explanation consists of extracting a trace of the proof of a theorem, e.g., a subsumption, or of highlighting the axioms involved in a clash when an inconsistency is detected. Explanation is widely supported in modern *description logics* reasoners. If the execution of an enrichment step generates an inconsistency, the PoSecCo tools are able to retrieve conflicting axioms. If the inconsistency was not present before enrichment at least one of the axioms introduced by the enrichment module will be included in the explanation set. So, the tool can report to the user a quite detailed description of the cause of the inconsistency. The user can then discard the sub-model added in the last enrichment step or he can use free ontological guided editing support to change the other fragments of the model to resolve the inconsistency.

This scenario depicts the enrichment process as an activity that can massively impact different fragments of the ontology. It is our opinion that, in order

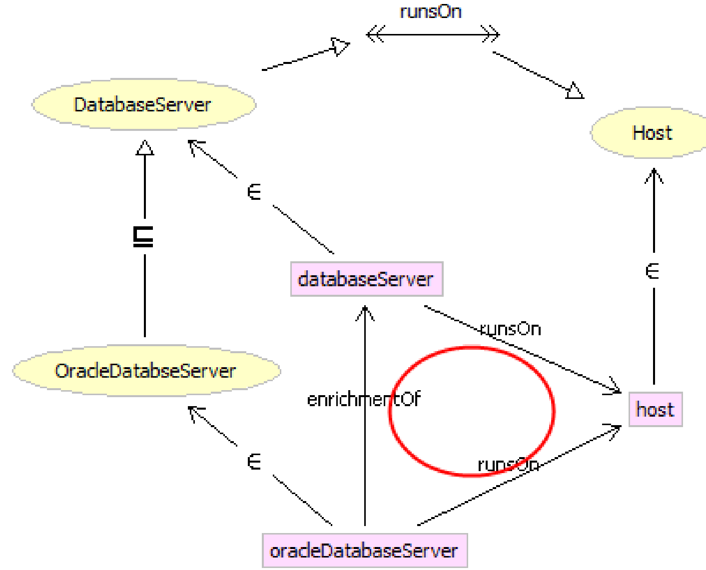


Figure 5.7: Harmonization condition for pre- and post- enrichment consistency

to make the process governable, and to keep it under the control of the user, re-tractability of the last enrichment step is the best approach. It guarantees that the enrichment process proceeds in a tree-shaped manner, exploring different possibilities but avoiding situations where a choice taken w.r.t. a specific fragment of the model can interfere with other enrichments guided by different criteria.

Unfortunately, only a small part of possible errors in the model can be evaluated by general reasoning tasks. Description Logics reasoner can only reveal terminological errors, that is, they are able to guarantee that the model satisfies general structural characteristics of the domain, precisely those characteristics we are able to describe in *T-box* axioms. However, many modeling errors involve complex structures in the model. In PoSecCo the detection, and, possibly, the resolution, of complex errors beyond purely terminological errors, is achieved through *harmonization modules*. Harmonization modules come in different shapes, according to different harmonization techniques and, more specifically, according to the sequence of manipulations to be done in order to check the specific harmonization property.

Some harmonizations make sense only after a refinement step. This is especially true when we use the individual creation approach to enrichment. The introduction of a new individual to represent the more detailed version of the original one provides the basis for enrichment tracing. On one hand, The user can navigate through chains of *enrichmentOf* properties to see all the intermediate steps that connect the higher level description to the most detailed one,

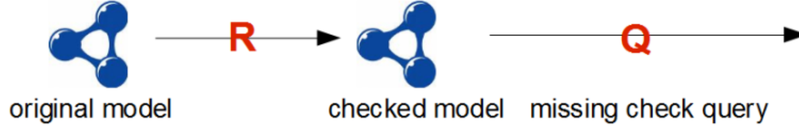


Figure 5.8: Property verification rule pattern

which is ready to be refined into abstract configurations. On the other hand, consistency between the description of views of the same resource at different levels of detail poses some issues. Figure 5.7 shows a simple case related to the enrichment of a database server. It is obvious that the enriched version of the server must run on the same host as its original description.

In this case harmonization consists of checking that the chain $runsOn^{-} \circ enrichmentOf \circ runsOn$ closes in a circle. This is a special case of the *property verification rule*, which is one of the classic cases of types of harmonizationas as classified in D 2.4 [**?**].

A property verification rule is composed of two steps, as shown in Figure 5.8:

1. A rule "*R*" (usually a SWRL rule) is used to add verification flags to all the nodes connected in the circle. After the execution of this rule, if the subgraph satisfies the desired structural constraints, one or more elements will be flagged as "correct". In our example *oracleDatabaseServer* should be the target to be checked and the rule may be in the form:

   ```
   type(?x, Server), enrichmentOf(?x,?x1),
   runsOn(?x1,?h), runsOn(x,h)
   ->
   type(?x, Correct)
   ```
   Listing 5.1: Enrichment harmonization rule for closed circles

   and

   ```
   type(?x, Server), not enrichmentOf(?x,[])
   ->
   type(?x, Correct)
   ```
   Listing 5.2: Enrichment harmonization rule for simple cases

   The two rules, if executed together, mark as correct the servers that either are hosted on the same host as their abstract versions, or that are not the refinement of other nodes. This creates the so called "*checked model*.

2. A query "*Q*" is executed on the *checked model* to retrieve all the nodes that should be marked as "correct", but are not. In our case the query, in the form of a SPARQL SELECT, may be as follows:

```
SELECT ?x
WHERE {
?x a :Server
{OPTIONAL {?x a ?y}
FILTER ?y=:Correct
}
FILTER ( !BOUND(?y) )
}
```

Listing 5.3: Enrichment harmonization rule

IT level harmonization can be necessary for both resource enrichment and policy enrichment. In fact, the addition of more details to abstract policies can result in modality conflicts between different policies, which were not revealed at a higher level of abstraction. In these cases, conflict resolution strategies must be taken into account in the rule part of the harmonization module.

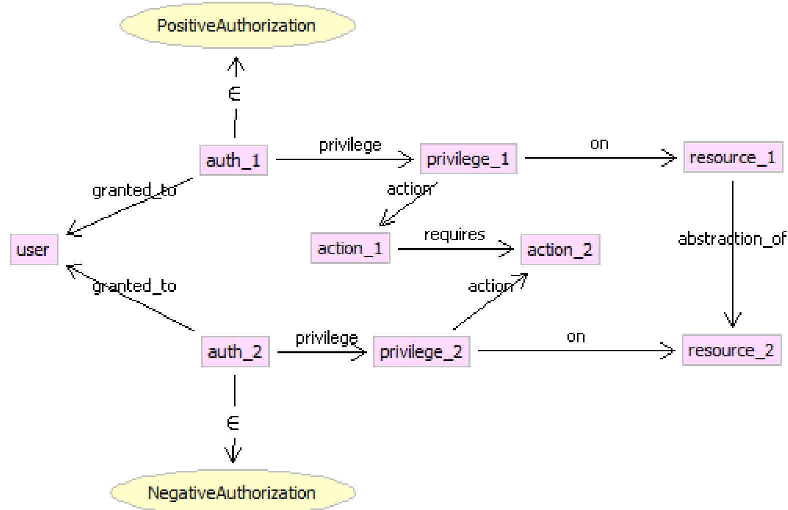


Figure 5.9: Simple modality conflict scenario

Figure 5.9 represents a classic case of modality conflict, which can be highlighted and possibly resolved using another harmonization pattern: *Violation detection rules* [92]. The two privileges have different modalities (signs), and, due to the dependency between the two kinds of actions, they describe a contradictory policy. The important point in the context of enrichment is that some details, e.g., the link between *privilege2* and *action2* in Figure 5.9 can be introduced by an enrichment step, so this kind of harmonization scenario can be integrated into enrichment just like the other cases introduced above in this section.

Another important case is the refinement of action types. Assume that a user group or a role is allowed to execute a generic *read* action on a server. In the enrichment phase the server is enriched to be a database server. In this case an enrichment module can suggest that the user should specify the nature of *read* in terms of the database specific ontology fragment. In this scenario the user is probably guided through the enrichment by allowing the selection of known specializations of the *read* action that are related to the *database* resource type model. However, the PoSecCo user is allowed to override system suggestions, and select another kind of action, like an *UPDATE*, which is consistent w.r.t. the resource type, but not w.r.t. action specialization, or *READ_FILE*, which is consistent w.r.t. action specialization but not w.r.t resource type. In this case, and in similar cases, a specialized harmonization module can interpret the model defined by the user and highlight the inconsistency. However, this harmonization module makes sense only as an evaluation of the execution of the specific enrichment module.

In order to solve this problem, some specialized harmonization modules must be logically associated with the enrichment modules they depend upon.

5.4 Relation with refinement modules

As said before, user interactive enrichment modules are similar to refinement modules from the structural point of view. Like the refinement scenario, the enrichment strategy can follow the *Select, Create and Connect (SCC)* approach. An enrichment strategy is an n-tuple $< Q, R >$ where:

$Q = \{N_1, N_2, ..., N_k\}$ represents the set of candidate nodes for the enrichment process;

$R = \{R_1, R_2, ..., R_m\}$ represents the set of SWRL rules that allow the introduction of new properties and, optionally, new enriched nodes.

Compared to refinement strategies (see below), enrichment is simpler due to the use of the same terminology (or ontology) to represent both the original high level abstract description and the enriched one. Despite this difference, we can apply the same techniques we introduced for policy refinement, using the property *enrichmentOf* instead of the property *implements*.

Finally, we have to consider the interaction between enrichment and refinement phases.

Figure 5.10 shows an abstract view of the whole workflow. Enrichment works inside the IT level, and the enriched objects can first be harmonized, and later refined. Harmonization occurs inside the IT level, either at the same level of abstraction, or between different levels of abstraction. Refinement, on the other hand, bridges the gap between the business level, the IT level and the landscape level, the later of which contains the abstract configurations.

Enrichment is a sort of refinement inside the IT level. In order to keep a strict separation between the phases depicted in figure 5.10, only the most

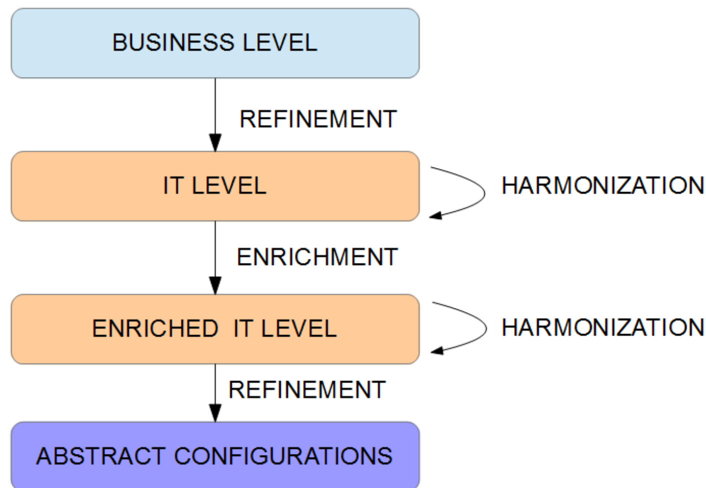


Figure 5.10: Abstract representation of the refinement process.

enriched elements can be used as a source for the refinement phase. Even if it is technically possible to further enrich an already refined entry at the IT level, this practice can lead to confusing and difficult to maintain models.

This semantics is enforced by further constraining the execution of refinement rules. In fact, in order to guarantee the desired behavior, the refinement engine should be aware of the status of the enrichment process, and should be able to ignore abstract nodes, and operate only on the fully enriched ones. Thus,the IT-level nodes suitable for refinement to abstract configurations (see Section 2.2.2) must fulfill these conditions:

1. the individual has all the properties and all the characteristics required for refinement;

2. furthermore, no individual is defined that represents a more concrete and enriched version of the same entity.

The second condition can be enforced by adding the rule that no incoming property of type *refinementOf* involves the candidate node.

5.4.1 Enrichment of IT Policies

IT policies that describe authorization and authentication rules have to be enriched, using the techniques described above, before it is possible to refine them to technology-specific abstract configurations for Database Management Systems, File Systems and Web Applications. The enrichment of IT policies that contain authorization or authentication rules happens in two phases. The first

phase, also called the first level of enrichment, enriches the IT Policy with further technology independent IT-level concepts. More specifically, it allows us to represent, in an abstract way, respectively authorization actions, authentication methods and authentication conditions.

In the second level of the enrichment process, an IT policy is augmented with technology specific information. This phase is realized by importing *enrichment ontologies* that are specific to the target system's technology and type of policy (e.g. authentication or authorization).

5.5 Refinement of IT Policies

Before starting with the description of the refinement process, we have to introduce the object property *implements*. This property connects an authorization/authentication at the IT level to its representation at the Abstract Configuration level. This object property will allow to maintain a link among elements belonging to different levels in the overall architecture.

We define two different approaches for the refinement of the IT policies. The approaches are general and can be used both for authentication and authorization, for OS, DBMS, and Web Application Server. Each approach is based on the use of ontologies and related technologies (e.g., SWRL, SPARQL). The first approach is called *Select, Create and Connect (SCC)*, the second approach is called *Select and Construct (SC)*.

5.5.1 Select, Create and Connect (SCC)

A refinement strategy is a triple $< Q, N, R >$ where:

$Q = \{N_1, N_2, ..., N_k\}$ represents the set of candidates nodes for the refinement process;

$N = \{C_1, C_2, ..., C_n\}$ represents the set of concepts that belong to the Abstract Configuration;

$R = \{R_1, R_2, ..., R_m\}$ represents the set of SWRL rules that allow to infer the links among nodes.

The IT Policy refinement starts by selecting candidate nodes which are elements of the IT Policy. Candidate nodes can be selected manually by the user through a GUI or by querying the IT Policy ontology. An example of a query to retrieve the candidate nodes would be a selection of all elements that are not linked to an Abstract Configuration through an 'implements'-relationship. The concepts of the Abstract Configuration are the entities of the configuration metamodel defined in Section 2.2.3. The SWRL rules are ad-hoc and created by the designer of the PoSecCo tool.

We will describe this method with the help of a concrete example. For the example, we consider the ITPolicy presented in WD1.8 [36]: *only system*

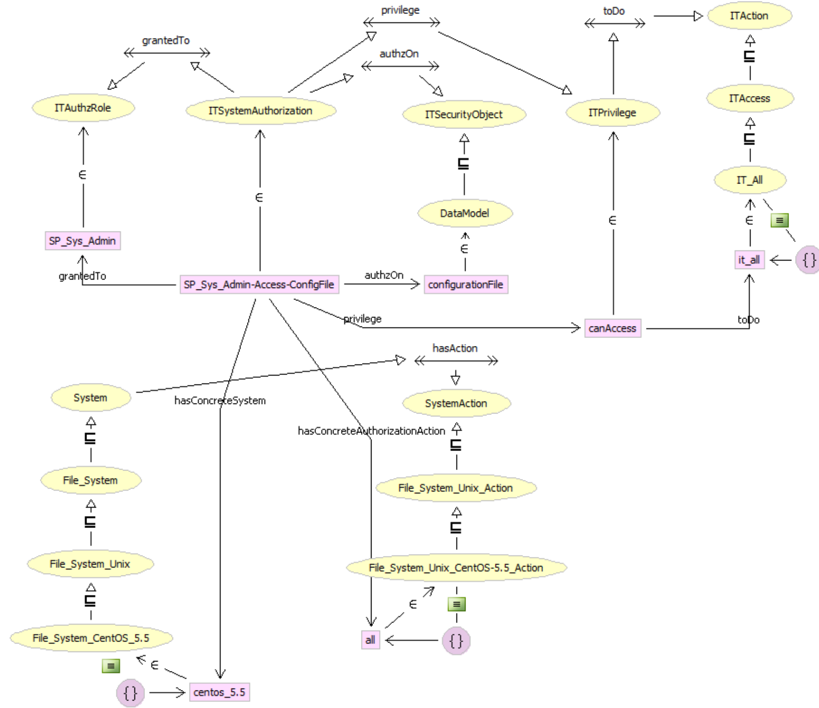


Figure 5.11: Ontological representation of the authorization.

administrator shall access to OS-level configuration file on Unix server (ITP60). Figure 5.11 shows the ontological representation of the authorization, already enriched (the example only considers the positive authorization that gives to administrators the privilege to access the configuration file; the "only" part, managed by a negative authorization, is not considered).

The first step of the SCC method is to *Select* the candidate nodes for the refinement process. To do this, we use the object property *implements*. If an authorization/authentication element has one (or more than one) *implements* property, it can be considered refined, otherwise it is a candidate node for the refinement process. In order to select all these nodes, the process uses a SPARQL-DL query that provides as result a list of elements already enriched and ready for the refinement process. Listing 5.4 shows an example of the query.

```
SELECT ?p
WHERE{
?p a:ITSecurityRule
OPTIONAL {?x :implements ?p}
FILTER (!bound(?x))
}
```

Listing 5.4: SPARQL query that allows to retrieve all nodes ready for the refinement process.

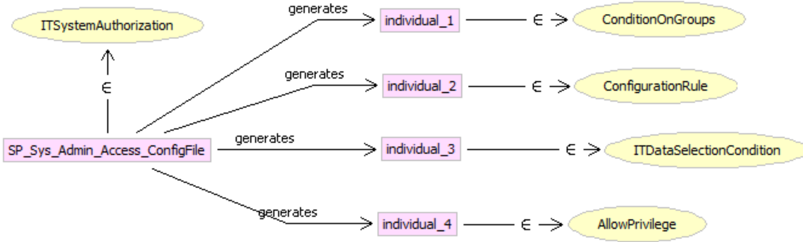


Figure 5.12: Output generated by the *Create phase*

This query returns all the *ITRoleAuthorization*, *ITSystemAuthorization* and *ITAuthenticationRule* individuals that are not refined. In the example, we assume that the query returns the individual *SP_Sys_Admin-Access-ConfigFile*.

The second step allows to *Create* technical nodes. This step is mandatory if we want to use SWRL, because, in order to maintain the safety, SWRL rules cannot create new nodes. To overcome this limitation, we have to use custom Java modules that insert into the ontology a set of new nodes, and add the *generates* object property. Figure 5.12 shows the result of this step.

The third step *Connect* is used to link the nodes created in the previous step. This connection is done with the use of the following SWRL rule.

```
ITSystemAuthorization(?auth),
DataModel(?cF),
authzOn(?auth, ?cF),
generates(?auth, ?res),
rdf:type(?res, ITResourceSelectionCondition),
generates(?auth, ?cR),
rdf:type(?cR, ConfigurationRule),
generates(?auth, ?cOnG),
rdf:type(?cR, ConfigurationOnGroups),
generates(?auth, ?aP),
rdf:type(?aP, AllowPrivilge),
=>
implies(?cR, ?auth),
matches(?cR, ?cOnG),
matches(?cR, ?res),
enforces(?cR, ?aP)
```

Listing 5.5: SWRL rule that allows to infer links among nodes.

Figure 5.13 shows the result of the execution of the SWRL rule.

To summarize, the approach integrates Java modules for the creation of nodes that will be the basis for the individuals in the abstract configuration, overcoming in this way the limitations of SPARQL and SWRL, but at the same time using these languages for the benefits they offer in the identification of the nodes that have to be processed and for the management of the connections among nodes that are consistent with the semantics of the refinement.

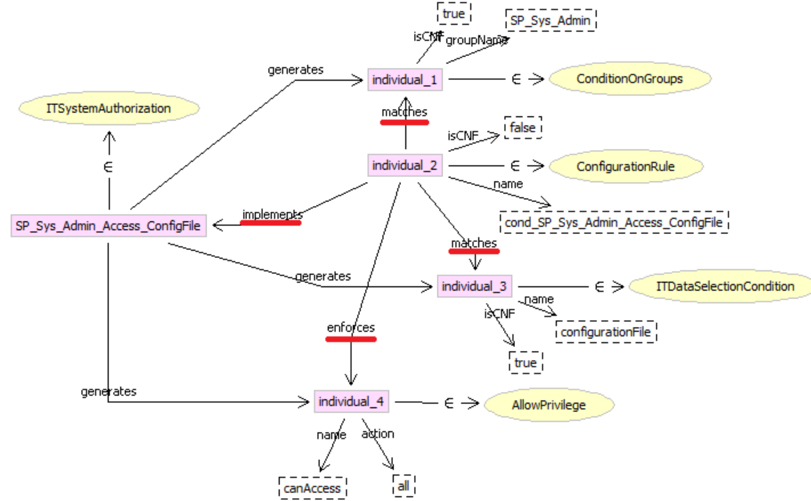


Figure 5.13: Output generated by the *Connect phase*

5.5.2 Select and Construct (SC)

This second approach avoids the use of Java modules in the *Create* phase and only relies on the use of SPARQL-DL and SPARQL. A refinement strategy is a couple $< Q, CQ >$ where:

$Q = \{N_1, N_2, ..., N_k\}$ represents the set of candidates nodes for the refinement process;

$N = \{CQ_1, CQ_2, ..., CQ_n\}$ represents a set of parametric *construct* SPARQL queries.

Like in the description of the previous approach, we will explain this method with the use of the following ITPolicy described in WD1.8 [36]: *Only Administrators shall be able to logon to Unix servers* (ITP50). Figure 5.14 shows the ontological representation of this ITPolicy.

The first step of this approach is the same as the previous one. With the use of a SPARQL-DL query, we *Select* all the nodes ready for the refinement process. Listing 5.6 shows the query.

```
SELECT ?p
WHERE{
?p a:ITSecurityRule
OPTIONAL{ ?x :implements ?p}
FILTER(!bound(?x))
}
```

Listing 5.6: SPARQL query that allows to retrieve all the nodes ready for the refinement process.

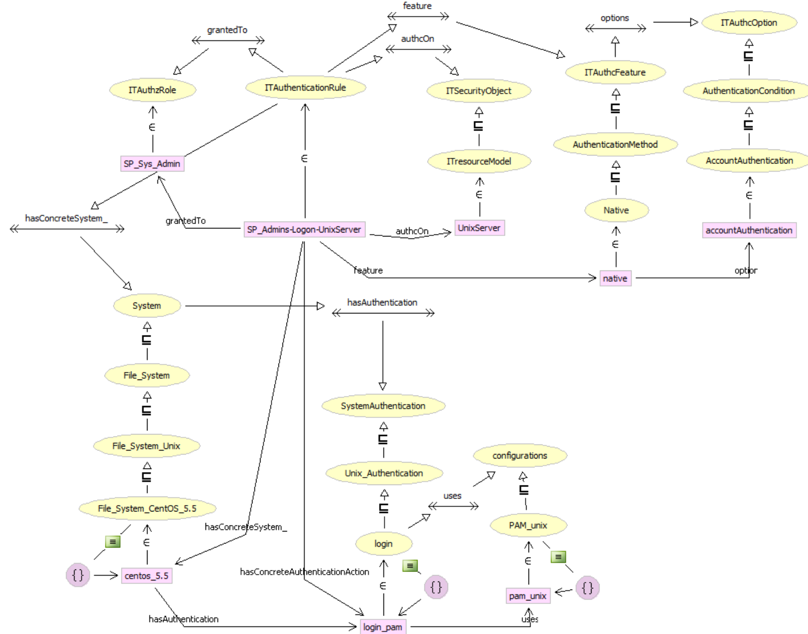


Figure 5.14: Ontological representation of the ITPolicy ITP50.

We assume that the result of this query in this example is the individual *Admin_Logon_UnixServer*, which does not have an *implements* property with other elements.

The second step of this method is to *Construct* the new nodes and link them. This can be done with the use of two SPARQL queries. Listing 5.7 and Listing 5.8 show the query that allow to create new nodes.

```
CONSTRUCT{
_:C rdf:type :ConfigurationRule;
:implements ?x -> :SP_App_Admin-Logon-UnixServer;
:matches _:RES;
:matches _:GR;
_:RES rdf:type ITResourceSelectionCondition;
_:GR rdf:type ConditionOnGroups;
:name ?r.
}
WHERE {
:SP_App_Admin-Logon-UnixServer a :ITAuthenticationRule;
:grantedTo ?r.
?r a ITAuthRole.
}
```

Listing 5.7: SPARQL query that permits to *Construct* new nodes

```
CONSTRUCT{
?c :enforces ?priv
?priv a AllowPrivilege;
:action ?action.
}
WHERE {
?c :implements :SP_App_Admin-Logon-UnixServer;
:enforces ?priv
:SP_App_Admin-Logon-UnixServer privilege ?p.
?p :actions ?action.
}
```

Listing 5.8: SPARQL query that infers the links among the new nodes.

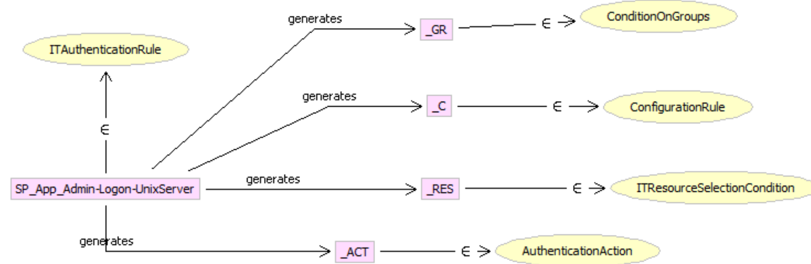


Figure 5.15: New nodes generated by the *Construct phase*

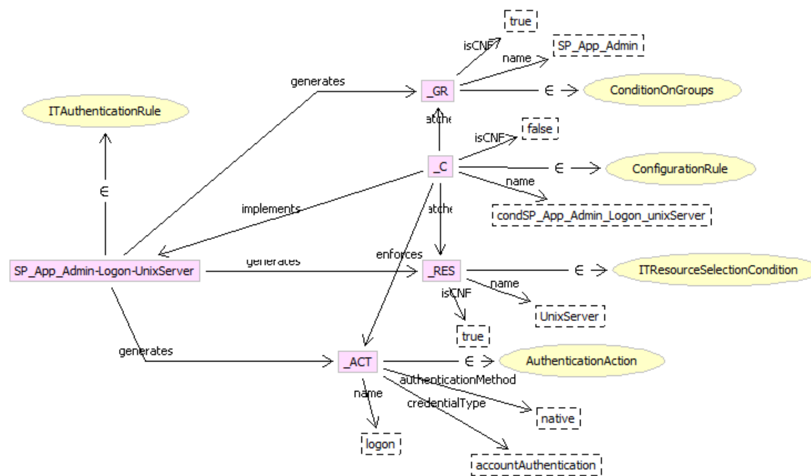


Figure 5.16: Output generated by the *Construct phase*

Both the methods described above can be used for the refinement of authorization and authentication rules. The second approach permits to refine an ITPolicy with the use of only Semantic Web tools. Instead, the first approach

has to use custom Java modules in order to introduce new nodes and this introduces a component that is not as transparent and easy to implement as pure Semantic Web tools.

6

Software for the IT Policy language

The application of the model-driven approach to the security requirements requires the implementation of a rich collection of tools supporting, for each of the many components in the system, the refinement from the high-level representation of security requirements to the concrete security configuration.

To support the user (i.e., Security Administrator) in this scenario we have developed the *PoSecCo ITPolicy Tool* [86] that can be used to define access control policies, to check the absence of structural errors and to reason about the model in order to identify conflicts and inconsistencies. Furthermore, the tool can derive and deploy, in a semi-automated way, the concrete security configuration of the system (e.g., OS, DBMS).

However, to drive the tool in the derivation process (i.e., refinement) there is the need to increase knowledge on the security configuration in the model relying on Semantic Web technology, and specifically on an enrichment of the model with the use of OWL (Ontology Web Language).

The use of OWL within the definition of the model will bring significant benefits, especially in terms of partial automatization of the process of model verification and refinement. In fact, thanks to the use of OWL, model implementation can be done in part automatically. We use the Web Ontology Language (OWL), which is a family of knowledge representation languages based on Description Logic (DL) with a representation in RDF, to define our model and to describe access control policies

We then use Semantic Web technology, namely *OWL-DL* [79], *SWRL* [54], *SPARQL* [97] and other technology, commonly used in the model driven approach, to provide security administrators with the following functionalities:

1. **Editing**;

2. **Harmonization**;

3. **Refinement** and **Enrichment**.

This Chapter presents the main design principles that have been used in the construction of the Eclipse plug-in *ITPolicy Tool* that implements the aspects described in Sections 2, 3 and 5.

6.1 Requirements

The consideration of the role of the IT Policy Tool in PoSecCo leads to the identification of a number of requirements, presented below and briefly commented on how they are met by the solution presented in the next sections.

ITP-R01 The IT Policy Tool is coordinated with the other tools, preceding and following it in the workflow. The IT Policy Tool receives input from the Business Policies and the IT Functional level. The relationships with these tools have been clearly specified. The structure of the IT Policy Tool has also been designed taking into account the design of the Infrastructure meta-model.

ITP-R02 The IT Policy supports the use cases related with the IT level. The analysis above shows the relationship between the IT Policy Tool and the PoSecCo use cases.

ITP-R03 The IT Policy Tool provides a graphical representation of the IT policy. The elements in the IT Policy can be represented in a *graphical user interface* (GUI) to facilitate both the user in the design phase of the IT Policy and the auditors in the design of the controls.

ITP-R04 The IT Policy Tool helps the user in the design of the IT Policy. The IT Policy Tool offers a rich editing and exploration support for the design of the IT Policy.

ITP-R05 The IT Policy Tool produces an IT Policy that can be integrated with other environments. A translation to XACML has been designed for the authorizations in the PoSecCo IT Policy, in order to facilitate the integration with other environments and their policy analysis tools.

ITP-R06 The IT Policy Tool delivers the possibility to build an ontological representation of the IT Policy. The IT Policy Tool is enriched by the support for ontologies. A significant advantage offered by ontologies is the possibility of checking the consistency of the model instances, going well beyond what can be offered by classical modeling tools.

ITP-R07 The IT Policy Tool provides the functionalities to introduce preliminary audit. The IT Policy can offer the functionalities, through the use of ontologies and also the reasoning, to implement some preliminary audit.

ITP-R08 The IT Policy Tool provides the interface to communicate with the PoSecCo repository. The IT Policy Tool offers the functionalities to retrieve and store both the IT Policy and other artifacts (for instance the ontology) in the PoSecCo repository.

ITP-R09 The IT Policy Tool offers to the user the capability to introduce and mantain a direct link between different elements within different levels of the PoSecCo architecture. The IT Policy Tool offers the functionalities to create a direct link both between the IT Policy and the Business requirements and between the IT Policy and the Landscape. This is a critical requirement for the PoSecCo refinement chain.

ITP-R10 The IT Policy Tool provides the functionalities to enrich the ontology with specific information about the environment where the policy is applied. The IT Policy Tool allows the user to use external tools, like Protégé, to add specific information about the environment where the IT Policy will be applied.

6.2 Architecture

We introduce here the main design principles that followed in the construction of the PoSecCo IT Policy Tool, an Eclipse plugin that supports the definition of IT Policies. The pattern used in the development of the tool is the *Iterative and incremental development*. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of earlier parts or versions of the system. Learning comes from both the development and use of the system, where possible key steps in the process start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added.

The ITPolicy Tool aims at allowing security administrators to define, manage and analyze security policies [45, 86] through several refinement steps. We have chosen to implement it on the basis of the Eclipse framework for four main reasons:

1. Eclipse is now one of the de-facto standards in terms of IDEs and has several plugins related to model driven engineering. The Eclipse framework is flexible enough to support the ITPolicy Tool requirements,

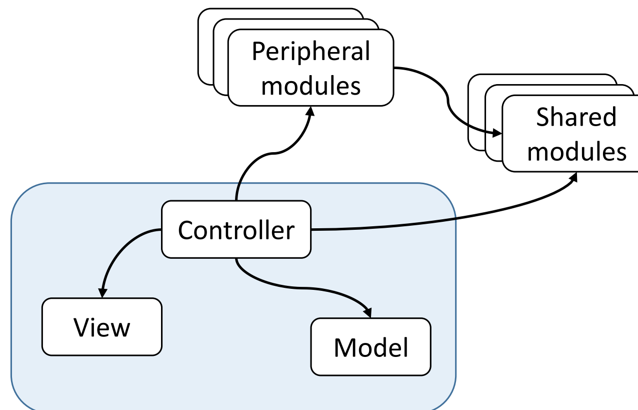


Figure 6.1: Architecture of the Tool

2. it provides several useful characteristics that ease the definition of the GUI of the tool,

3. it can be easily integrated with Semantic Web tools by using the *OWLAPI*, *Jena* libraries (for an exhaustive explanation of the technology used, see Appendix D),

4. by defining a new *extension point*, it lets us define an extensible and flexible way to handle the integration and customization of new services in the architecture.

Figure 6.1 represents the abstract architecture of the tool. The pattern used to develop this level is the *Model-View-Controller* (MVC).

- **Controller** module. The Controller module represents the core of the entire plugin. It instantiates each module at start time, receives the command from the user (through the *View* module) and executes the related action on the data.

- **Model** module. The Model module permits to maintain a dynamic representation of the IT Policy. The model is a collection of Java classes that keep updated the information about the IT Policy.

- **View** module. The View module provides the functionalities to show the information on the screen.

Furthermore, there are two other modules:

- **Peripheral** module. It is also known as *Functional module* because it implements a specific functionality (e.g., Harmonization, Enrichment).

- **Shared** module. It provides functionalities that are common to several modules.

6.2.1 Source Code

The source code of the ITPolicy Tool is available on the ITPolicy Tool web site [1], In order to give an overview on the size and complexity of the implementative effort for the ITPolicy Tool, Table 6.1 provides a "qualitative" description through the use of several metrics. The metrics selected are the following:

- NOP: Number Of Packages
- NOC: Number Of Classes
- TLOC: Total Lines of Code
- NOM: Number of Methods
- VG: McCabe Cyclomatic Complexity

[1] http://cs.unibg.it/posecco/source.html

Project Name	**NOP**	**NOC**	**TLOC**	**NOM**	**VG**
eu.posecco.businessrequirement	5	6	538	33	1.175
eu.posecco.enrichment.core	12	27	3184	225	1.391
eu.posecco.enrichment.landscape	8	10	629	44	1.481
eu.posecco.importfiles	1	3	230	3	1.608
eu.posecco.importontology	2	6	322	16	1.85
eu.posecco.itfunctionalmetamodel	6	7	424	24	1.516
eu.posecco.itharmonization.core	8	48	7904	580	1.568
eu.posecco.itharmonization.interactive.authenticationcheck	1	1	55	1	1.12
eu.posecco.itharmonization.interactive.cyclecheck	1	4	188	4	1.375
eu.posecco.itharmonization.modality.conflict	4	12	2232	14	1.643
eu.posecco.itharmonization.redundancy	1	2	782	11	1.6
eu.posecco.itharmonization.repair	2	5	349	12	2.132
eu.posecco.itharmonization.sod	1	1	122	4	1.25
eu.posecco.itontology	4	5	1844	85	1.764
eu.posecco.itpolicytool	31	100	16767	835	2.164
eu.posecco.itpolicytool.consumer	2	3	64	7	0.923
eu.posecco.itpolicytool.publisher	1	2	49	6	1.067
eu.posecco.itsearch	3	5	582	20	1.576
eu.posecco.itsecuritymetamodel	5	16	5510	271	2.2
eu.posecco.move	5	128	41751	2084	3.009
eu.posecco.neontoolkit.manager	3	4	387	14	1.786
eu.posecco.pellet	3	4	1071	12	1.453
eu.posecco.refinement.core	15	29	4326	242	1.629
Total	124	428	89310	4547	-

Table 6.1: Metrics

6.2.2 Input/Output ITPolicy Tool

Due to the fact that all the artifacts produced by the *PoSecCo tools* (e.g., CoSerMas, SDSS) have to be stored in MoVE, the ITPolicy Tool provides a complete set of functionalities to retrieve and store artifacts in MoVE. More specifically, the ITPolicy Tool can receives as input (a) XMI files, (b) OWL files and (c) XML files, all of them in agreement with the IT layer Security metamodel. Furthermore, the ITPolicy tool can produce as output artifacts in (a) XMI format and (b) OWL format.

6.3 Editor

In order to fulfill the requirements ITP-R01, ITP-R02 related to the editing phase described in Section 6.1, the ITPolicy Tool delivers several functionalities provided by different components.

- a main window to display and edit ITPolicy description files. The tool assigns a different tab panel in the window to every concept from the metamodel. This set of tabs provides the user with a high level guide through the model, always offering direct access to the main components of the model. The selection of a tab opens a form that permits to enter values for each of the properties of the corresponding entity. Instances of each class can be described in a homogeneous way by the same form used for entering its properties. The plug-in has many forms (see Figure 6.2 for an example), in particular there is a form for each class of the model described in [93] and in Section 2.2.2. These forms allow to maintain a high usability of the tool.

- a navigation tool, that organizes resources in a finer taxonomy according to the specific top level concept selected by the user in the main window. The classification proposed to the user is driven by the ontology itself. This makes the tool quite flexible and automatically adaptable to changes in the ontology. The tool can be applied to any fragment of the metamodel.

- a search service, that offers the possibility to the user to search a specific element in the ITPolicy and the related element (e.g., business security requirements).

- a set of task buttons, published in the main toolbar, that offer direct access to functions like the creation of the OWL ontology starting from an instance of the security policy or the activation of the reasoning-based checking (i.e., harmonization).

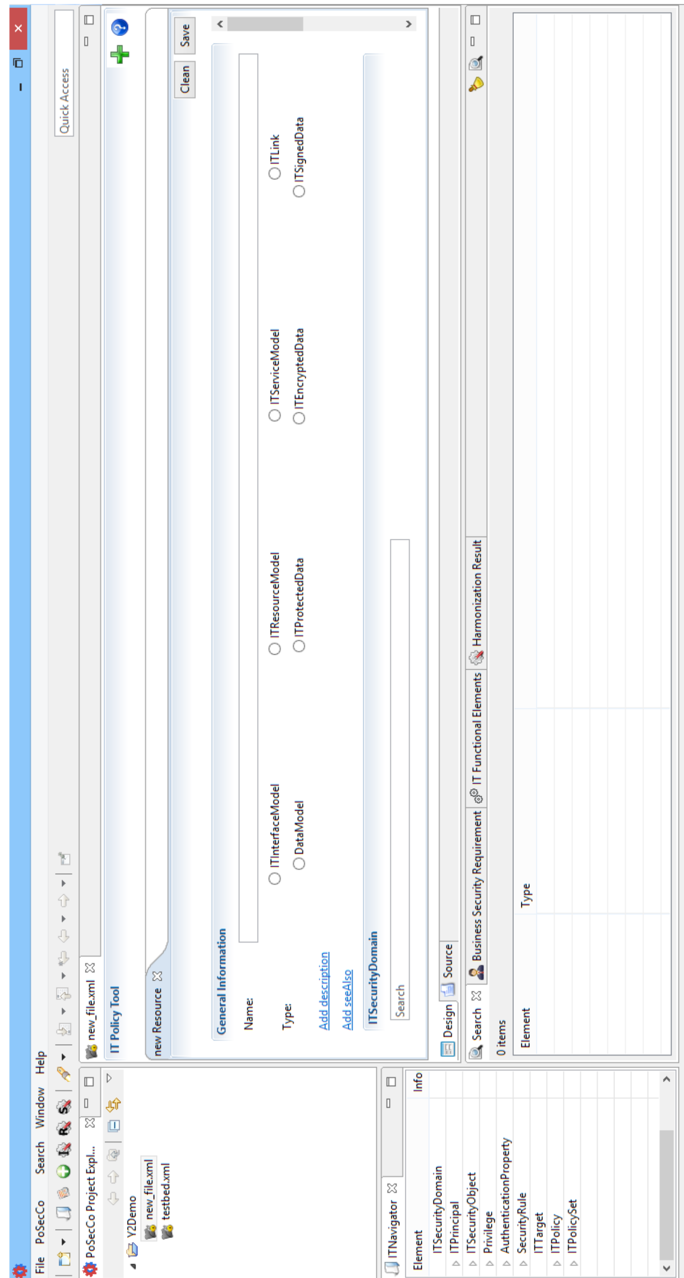


Figure 6.2: Components of the Tool

Another functionality provided by the ITPolicy Tool is the possibility to link the ITPolicy elements to (a) the Business Security requirements, (b) the Functional System model and (c) the Abstract Configuration, in order to create the Policy chain and then store all of this data in *MoVE* [105]. MoVE is the central repository for models and for the ontologies used in PoSecCo. Due to the fact that the link between the ITPolicy and the Abstract Configuration is built automatically through the *refinement phase*, the user must build, manually, the links among the ITPolicy and other layers (i.e., Business Security requirements, Functional System model). In order to aid the user in the building of links among meta-models, the ITPolicy Tool provides two views allowing the user to easily choose the elements, respectively business security requirements and IT functional elements, to link with a specific ITPolicy.

6.4 Harmonization

One of the main functionalities provided by the ITPolicy Tool is the *Harmonization phase* described in Section 3. This phase allows the tool to prevent, detect and correct possible inconsistencies in the policies. Inconsistencies can represent misconfigurations, conflicts between different policies or simply suboptimal or redundant descriptions of the intended constraints [88]. Harmonization is focused on the following aspects:

- verification of the correctness of each policy with respect to the PoSecCo policy ontology;
- detection of possible inconsistencies between contradictory policies at a given level of abstraction;
- detection of possible inconsistencies between an abstract policy and a policy that refines it at a lower level of details;
- identification of redundant policies in the same policy set.

According to the architecture presented in Section 6.2, the peripheral modules in charge to provide these functionalities are the following:

Interactive modules: these modules implement several interactive reasoning services which are executed during the editing phase of the *ITPolicy*. They are used to detect missing individuals in the ontology or to identify axioms that introduce structural violations in the ontology. These modules are implemented by using Semantic Web technologies, primarily by using *SPARQL-DL* due to its graph matching capabilities.

eu.posecco.itharmonization.interactive.checkMissingITElement implements a reasoning service that checks whether a certain *ITAuthenticationRule* exists or not. This functionality aids the user during the editing phase of the ITPolicy. We know that authorization and authentication are strictly related. For instance, it makes no sense define authorizations for

a user on a system without an authentication that allows the user to logon on the system, and respectively define authentications for a user on a system without adding authorizations. Thus, in order to avoid this issue, the ITPolicy Tool provides the functionality to check the presence of an authentication when the user inserts an authorization.

eu.posecco.itharmonization.interactive.checkStructuralConstraints implements several interactive reasoning services that prevent the creation of structural inconsistencies (e.g., cycles in the group hierarchy, in the role hierarchy and in the security object hierarchy).

Standard reasoning modules: these modules can be used to detect anomalies and inconsistencies in the IT policy under development. In this way the *ITPolicy Tool* can report them to the user, and let him modify the model according to the analysis' results, in order to remove inconsistencies. These modules are implemented using Semantic Web technologies, i.e., the IT policy is represented as an *OWL ontology* and the reasoning services are expressed by means of Semantic Web tools, e.g., *OWL-DL*, *SWRL* and *SPARQL-DL*.

eu.posecco.itharmonization.modalityconflict is the component that implements the *Policy Incompatibility* reasoning service, which can be used to detect authorizations that are incompatible; e.g., it can detect if in the *IT-Model* there is an authorization that gives to a user the permission to execute a certain action and another authorization that removes from the same user the permission to execute the same action. It defines the *PolicyIncompatibility* extension for the *ITReasoningService* extension point. Three versions of the service exist: the first two versions implement the service by means of OWL-DL reasoning (the first one uses *Hermit* as reasoner whereas the second one uses *Pellet*) and the last one implements the service by means of *SWRL* rules.

eu.posecco.itharmonization.redundancy is the component that implements the *Redundancy Detection* reasoning service which can be used to detect redundancies in the *IT-Model*, i.e., the model may contain authorizations that are implied by other authorizations and thus can be removed. It defines the *RedundancyDetection* extension for the *ReasoningService* extension point. The reasoning service is implemented by means of SWRL rules and SPARQL-DL queries.

eu.posecco.itharmonization.sod is the component that implements the *SoD Conflict Detection* reasoning service which can detect authorizations that break Separation of Duty constraints expressed in the IT policy. It defines the *SoDConflictDetection* extension for the *ReasoningService* extension point. The reasoning service is implemented by using OWL-DL reasoning. Two versions of the service exist: the first one uses *Hermit* as reasoner, whereas the second one uses *Pellet*.

Repair module: This module implements the repair services that can provide the user with semi-automated repair capabilities. A repair service takes

as input a list of *IFix* objects, which is the result of the execution of a reasoning service, and it computes a list of repair options that can be applied to the IT policy in order to remove the detected problems.

eu.posecco.itharmonization.repair implements two repair services, an automated one and a semi-automated one. The repair strategies implemented in both services are simple. In case the inconsistency is due to a *Redundancy* issue, the repair service removes the redundant authorizations. In case the inconsistency is due to a *Separation of Duty* issue, the repair service removes one of the role authorizations or role hierarchy axioms that cause the SoD. In case the inconsistency is due to a *Modality Conflict*, the repair service removes one of the system authorizations causing the conflict.

Furthermore, a shared module among all the components described above is the *Harmonization core*. It is the component that contains all the functionalities shared by the reasoning services. It contains the classes that manage the ontology by using the OWL-API Java library (enriched by the use of SWRL and SPARQL-DL). It allows the definition of an ontological representation of the IT Policy. *OWL-API* through *OWLDataFactory* objects permits to instantiate all classes, properties and axioms of the ontology. This component is a dependency of the reasoning service components presented above, because all these components use core functionalities. The dedicated functionalities of a specific reasoning service, e.g., a particular reasoner or a set of SWRL rules, are included only in the specific component.

6.5 Refinement

As described in Section 2 the IT layer Security meta-model supports the representation of authentication and access control policies. It introduces several concepts that offer a high degree of flexibility and modularity. Furthermore, an important characteristic of the IT layer Security metamodel is the integration with ontologies, which increase the expressive power of the model and permit to support both the evolution of the scenario and the realization of sophisticated checks (e.g., consistency). Furthermore, ontologies enrich concepts and provide a more detailed explanation of the concepts themselves. The use of ontologies is the foundation of the refinement (and enrichment) process.

As for the harmonization process, the policy refinement process is executed by orchestrating a set of refinement modules. Each refinement module can identify the set of model elements readily available for refinement and produce a new description fragment that describes abstract configurations. In order to manage, in a flexible way, the refinement process we have decided to adopt the same approach described in the harmonization process, thus we have implemented the refinement process with the use of *extension points* (see Appendix E for an exhaustive explanation).

The peripheral module in charge to provide these functionalities is the following:

Refinement module: This module implements the refinement service. The refinement service takes as input the enriched ontology, which is the result of the enrichment process, and it performs the transformation from the enriched IT Level to Abstract Configuration level.

eu.posecco.refinement.core implements the refinement service. It is composed by several modules, each dedicated to a specific target system (e.g., DBMS, OS). It defines the following extensions (both for authentication and authorization) (a) *DBMSRefinementModule* and (b) *OSRefinementModule* for the *RefinementModule* extension point (see Appendix E for an exhaustive explanation). The service is implemented by the use of *Jena* for the ontology management.

6.5.1 Enrichment

The enrichment of IT policies that contain authorization or authentication rules is realized by importing enrichment ontologies that are specific to the target system's technology and type of policy (e.g., authentication or authorization) [102]. The ITPolicy Tool implements several enrichment modules that are executed during the enrichment process. They are used to introduce additional information. As for harmonization and refinement, the import of enrichment ontology is based on the use of *extension points*. The tool defines an extension point, called *Enrichment*, that allows other plugins to contribute with new enrichment modules.

The peripheral module in charge to provide these functionalities is the following:

Enrichment modules: This module implements the enrichment service. The enrichment service takes as input the ontological representation of the IT Policy and, through the import of additional ontologies, generates an enriched ontology containing information about authorization and authentication and the target system.

[**eu.posecco.enrichment.core**] implements the enrichment service. As for the refinement service, it is composed by several modules, each dedicated to a specific target system (e.g., DBMS, OS). For instance, it defines the following extensions (both for authentication and authorization): (a) *DBEnrichmentModule* (for MySQL 5.*) and (b) *OSEnrichmentModule* (for CentOS 5.*) for the *EnrichmentModule* extension point (see Appendix F for an exhaustive explanation). In order to aid the user in the import of the enrichment ontology, the module provides the functionality to use a custom wizard for each enrichment ontology.

6.6 Additional functionalities

In addition to the main functionalities described above, the IT Policy Tool provides additional functionalities, such as the possibility to convert automatically the artifacts from the XMI format to OWL format and viceversa. Furthermore, an interesting functionality provided by the ITPolicy Tool is also the possibility to perform *various kinds of analyses*, like cost-driven analysis, on the basis of the IT Policy defined by the user. The theoretical foundation of these analyses are based on the work described in [64].

6.6.1 Quality of access control policies

The user through the IT layer Security meta-model can define authentication and authorization. An additional feature provided by the meta-model is the possibility to define role authorizations, thus assigning users to roles and then defining permissions on roles. As we know there are two general approaches to define roles: top-down and bottom-up. In the construction of an RBAC system both approaches are used. The IT Policy Tool provides the possibility to define roles through both approaches, in particular the user can define manually the roles (top-down approach) and then define the respective role-to-permission and user-to-role assignment. Otherwise, the IT Policy Tool provides the functionality to use the bottom-up approach. Starting from an IT Policy defined by the user, the tool can build the user-permission assignement (through a “factorization” of all policies) and then build a solution, with the use of heuristic algorithms able to solve the *Role Mining Problem* (RMP).

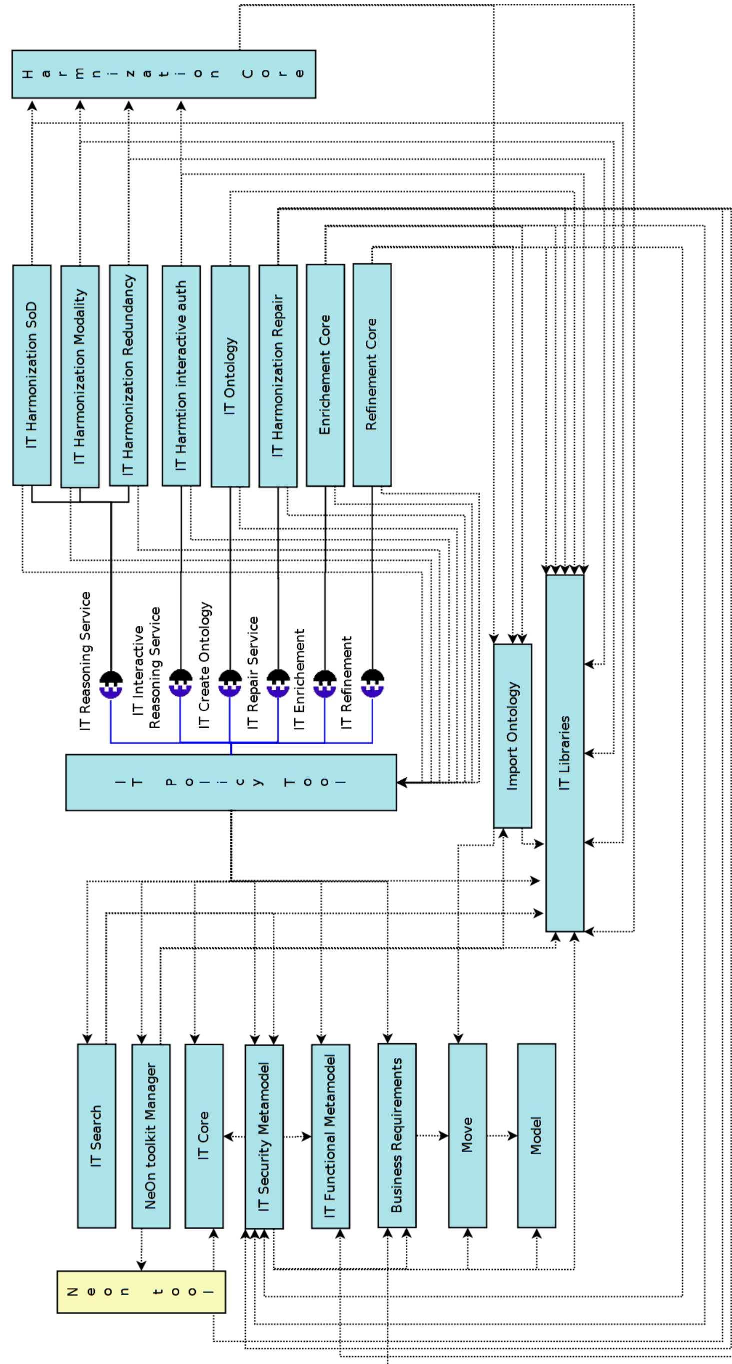


Figure 6.3: Components of the ITPolicy Tool

Part II

Security Management in Mobile Systems

1
Introduction

Mobile operating systems play a central role in the evolution of Information and Communication Technologies. One of the clearest trends of the past few years has been the adoption by users of mobile portable devices, replacing personal computers as the reference platform for the delivery of many ICT resources and services. The rapid success and wide deployment of mobile operating systems has also introduced a number of challenging security requirements, making explicit the need for an improvement of security technology.

The mobile scenario is indeed characterized by two mutually reinforcing aspects. On one hand, mobile devices are high-value targets, since they offer a direct financial incentive in the use of the credit that can be associated with the device or in the abuse of the available payment services (e.g., Google Wallet, telephone credit and mobile banking) [85, 118]. In addition, mobile devices permit the recovery of large collections of personal information and are the target of choice if an adversary wants to monitor the location and behavior of an individual. On the other hand, the system presents a high exposure, with users of modern mobile devices continuously adding new *apps* to their devices, to support a large variety of functions (we follow the common convention and use the term *app* to denote applications for a mobile operating system).

The risks are then greater and different from those of classical operating systems. The frequent installation of external code creates an important threat. The design of security solutions for mobile operating systems has to consider a careful balance between, on one side, the need for users to easily extend with unpredictable apps the set of functions of the system and, on the other side, the need for the system to be protected from potentially malicious apps.

It is to note that the greatest threats derive from apps that are offered through delivery channels that are alternative to the "official" *app markets* (e.g.,

[121]), whose number of app installations is increasing rapidly, pointing out the need of wider security layers. Apps in official markets, instead, are verified by the market owner and the ones detected as misbehaving are promptly removed from the market. The correct management of the app market is crucial, nevertheless it is not able by itself to fully mitigate the security concerns. The mobile operating systems have to provide a line of defense internal to the device against apps that, due to malicious intent or the presence of flaws in system components or other apps, may let an adversary abuse the system.

1.1 Rationale of the approach

The approach that we propose follows the principles of the Android security model, which aims at isolating from each other the apps that are executed by the system. Each app is confined within an assigned domain and interaction between the elements of the system is managed by a privileged component, which enforces the restrictions specified by a policy. The approach presented in the paper aims at strengthening this barrier, introducing an additional mechanism to guarantee that apps are isolated and cannot manipulate the behavior of other apps. The additional mechanism is obtained with an adaptation of the services of a Mandatory Access Control (MAC) model, which enriches the Discretionary Access Control (DAC) services native to the Linux kernel.

MAC models are commonly perceived as offering a significant contribution to the security of systems. However, one drawback of MAC models is represented by *policy management* which is a especially critical in complex systems such as Android, where each OEM tries to customize the MAC policy for its own devices. Samsung KNOX is the most well-known example. Policy customization provides benefits in terms of security, but it inevitably leads to *policy fragmentation*. Our work tries to do a step ahead in the policy standardization defining a set of *entry points* which can be used by both OEMs and developers in order to extend, under specific constraints, the MAC policy to fulfill their own security requirements and subsequently try to mitigate the policy fragmentation problem.

Apps can only become known to the system when the owner asks for their installation. The MAC policy has then to be dynamic, with the ability to react to the installation and deletion of apps, which requires modularity and the capability to incrementally update the security policy, with a policy module associated with an app. We use the term appPolicyModule to characterize it (when space is limited, like in table headers, we may use the acronym APM). The support for appPolicyModules allows app developers to benefit from the presence of a MAC model, letting them define security policies that increase the protection the app can get against attacks coming from other apps, which may try to manipulate the app and exploit its vulnerabilities.

2
Android Security Architecture

The Android security model shows a direct correspondence with the overall Android architecture, which is organized in three layers (from bottom to top): (a) an underlying Linux kernel, (b) a middleware framework, and (c) an upper application layer. The Linux kernel in the lowest layer provides low-level services and device drivers to other layers and it differs from a traditional Linux kernel, because it aims at running in an embedded environment and does not have all the features of a traditional Linux distribution. The second layer, the middleware framework, is composed of native Android libraries, runtime modules (e.g., the Dalvik Virtual Machine and the alternative Android Runtime ART) and an application support framework. The third layer is composed by apps. Apps are divided into two categories: (i) pre-installed apps (e.g., Web browser, phone dialer) and (ii) *third-party* apps installed by the user. In the paper we focus on the consideration of third-party apps, since the pre-installed ones are already covered by the system policy. Each app can be structured as distinct *app components*. There are four different types of app component: (a) *Activities*, typically each screen shown to a user is represented by a single *Activity component*; (b) *Services*, provide the background functionalities; (c) *Content Providers* used to share data among applications; and (d) *Broadcast Receivers* receive event notifications both from the system and from other applications.

Android provides distinct security mechanisms at the distinct layers. The Linux security model based on user identifiers (*uid*) and group identifiers (*gid*) operates at the lowest layer, with each app receiving a dedicated *uid* and *gid*. The granularity of this access control model is at the level of files and processes, reusing all the features of the classical DAC model of Unix/Linux, with a compact *acl* that describes for each resource the operations permitted respectively to the owner, members of the resource group, and every user of the system.

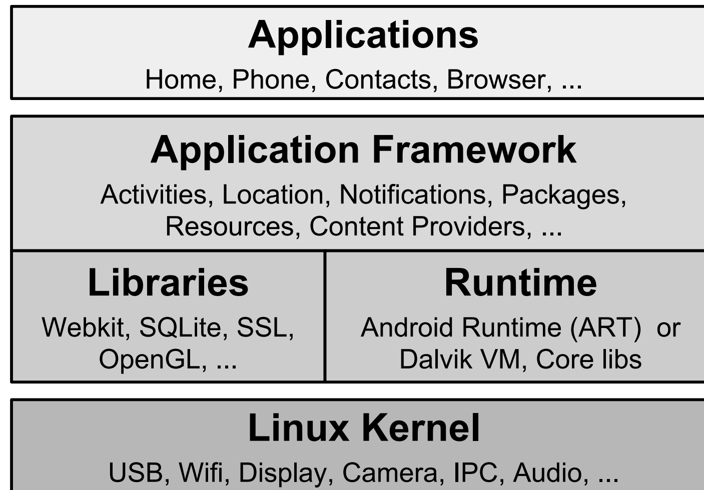


Figure 2.1: Overall representation of Android architecture.

At the application layer, Android uses fine-grained permissions to allow apps or components to interact with other apps/components or critical resources. The *Android Permission Framework* contains a rich and structured collection of privileges, in the 4.4.4 version more than 200, focused on the management of the large variety of resources that are offered by the operating system to apps. The access control model assumes that apps specify in their manifest the set of privileges that will be required for their execution. At installation time, users of Android devices have to explicitly accept the request for privileges by the app; in case the user does not accept the app request, the app is not installed.

2.1 SEAndroid

Recently, the SEAndroid initiative [109] has led to a significant extension of the security services, with the integration of Security Enhanced Linux (SELinux) [78] into the Android operating system. The goal of SEAndroid is to build a *mandatory access control* (MAC) model in Android using SELinux to enforce kernel-level MAC, introducing a set of middleware MAC extensions to the Android Permission Framework. SELinux originally started as the Flux Advanced Security Kernel (FLASK) [71] development by the Utah University Flux team and the US Department of Defense. The development was enhanced by the NSA and released as open source software. SELinux policies are expressed at the level of *security context* (also known as *security label* or just *label*). SELinux requires a security context to be associated with every process (or subject) and object, which is used to decide whether access is allowed or not as defined by the policy. Every request a process generates to access a resource will be accepted only if it is authorized by both the classical DAC access control service and by

the SELinux policy. The advantage of SELinux compared to the DAC model are its flexibility (the design of Linux assumes a *root* user that has full access to DAC-protected resources) and the fact that process and resource labels can be assigned and updated in a way that is specified at system level by the SELinux policy (in the DAC model owners are able to fully control the resources).

The middleware MAC extension chosen to bridge the gap between SELinux and the Android permission framework is called *install-time MAC* [109] (several middleware MACs have been developed, but only the install-time MAC has been integrated into the AOSP). This mechanism allows to check an app against a MAC policy (i.e., *mac_permissions.xml*). The integration of this middleware MAC ensures that the policy checks are unbypassable and always applied when apps are installed and when they are loaded during system startup.

The current design of SEAndroid aims at protecting core system resources from possible flaws in the implementation of security in the Android Permission Framework or at the DAC level. The exploitation of vulnerabilities becomes harder due to the constraints on privilege escalation that are introduced by SELinux. Unfortunately, the current use of SELinux in Android aims at protecting the system components and trusted apps from abuses by third-party apps. All the third-party apps fall within a single *untrusted_app* domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on Android permissions and the Linux DAC support. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy.

2.2 Threat Model

In Android each app receives a dedicated *uid* and *gid* at install-time. These identifiers are used to set the user and the group owner of the resources installed by the app in the default data directory, which is */data/data/"package_name"*. By default, the apps databases, settings, and all other data go there. Since user data for an application also resides in */data/data/"package_name"*, it is important that only that application has access to that particular folder. This confinement of the data folders permits to enforce a strict isolation from other applications. In Android this isolation is only enforced at DAC level, but this is not enough to protect the app and its own resources by other apps with *root* privileges. Android, by default, comes with a restricted set of permissions for its user and the installed applications (i.e., no root privileges). Despite this, apps can gain root privileges in two ways and use it to provide desirable additional features for users, but a malicious app may also abuse it to bypass Android's security measures.

On one hand many benign apps require root privileges to accomplish their job. For example, *Titanium Backup* [112] is one of Google Play's best-selling apps and it needs root privileges to backup system and user applications along with their data. In this scenario, the user typically flashes a recovery console on the device which has the permission to write on the system partition and from

there she installs an app such as *SuperSU* or *Superuser* in order to gain and manage root privileges. After that, the user can give root privileges to other applications. According to Google, users install non-malicious rooting apps by a ratio of 671 per million in 2014 (increased by 38% compared to the 491 per million in 2013 [75]). Moreover, there are successful community-ROMs, such as CyanogenMod with over 10 million installations, that provide root access to the user by default. Here, the user is aware of the fact that some apps act as root in the system and have access to everything, however she does not know how these privileges are used and she has to trust the app.

On the other hand, a malware could exploit a bug in a system component and gain root privileges to freely access the whole system in order to steal personal information or perform fraudulent actions. In this scenario the user is unaware of the fact that an app acts as root. Over the years Android has been attacked by threatening malware apps such as *DroidDreamLight*, which affected 30,000-120,000 users in May 2011 [7]. Recently the app *towelroot* has been released which, exploiting the *CVE-2014-3153* bug of Linux kernel, permits to “root the device” without the need to flash a recovery console, and gives root privileges potentially to all apps. This bug affects all Android versions up to 4.4.4 and thus represents a significant threat in the current DAC-only protection of private app resources.

Our proposal provides a solution to both scenarios through the use of appPolicyModules defined by the app and attached to the SELinux system policy.

2.2.1 Example

Hereinafter, we define a running example that we use as a proof of concept of our solution. *Dolphin Browser* [32] is considered one of the most successful mobile browsers for Android[1] with over 100 million downloads. It uses the *Webkit* engine and provides several features such as *gesture browsing* and *browsing boost*. We use it to show how the threat model defined in the previous section affects the current DAC-only security isolation of a real app and its private data. In Section 3.2 we use this example to identify the requirements and then we will illustrate how the use of a dedicated appPolicyModule can provide better security for its private data. The Android permissions requested by the app are:

```
android.permission-group.NETWORK
android.permission-group.ACCOUNTS
android.permission-group.LOCATION
android.permission-group.MICROPHONE
android.permission-group.CAMERA
```

Listing 2.1: SWRL rule that allows to infer links among nodes.

[1]At the time of writing *Chrome* is the most used mobile browser for Android devices; however, due to the fact that *Chrome* is included in the *Gapps*, it does not belong to the *untrusted_app* domain but to the *isolated_app* domain, thus it is not considered as a third-party app. The same discussion and threat model presented for Dolphin Browser is also valid for *Google Chrome for Android*. Moreover, all the passwords that the user saved in the Desktop version of Google Chrome using the same login details are available in the Android database.

Many browsers include a password manager component that stores confidential information such as usernames and passwords. The common strategy used by almost all the mobile browsers we have analyzed is to keep the credentials in a SQLite database. Following Google's best practices for developing secure apps, the password database is saved in the app data folder, which should be accessible only to the app itself. Another best practice (not used by Dolphin browser) to provide additional protection for sensitive data, is to encrypt local files using a key that is not directly accessible to the application. For example, a key can be placed in a *KeyStore* and protected with a user password that is not stored on the device. While this does not protect data from a root compromise that can monitor the user inputting the password, it can provide protection for a lost device without file system encryption.

Some of the browsers we have analyzed (e.g., *Google Chrome*) store the passwords in plaintext in the database, while others use some form of encryption (e.g., *Dolphin Browser*, *Firefox*). The decision to keep the passwords in plaintext can appear as a weakness, but even when the information is stored in an encrypted form, if the data needs to be recovered automatically by the app without the need of additional information not stored on the device (e.g., a master password known only by the user), a malware could use the same resources used by the legitimate application to retrieve the information.

There are a number of ways one can obtain the Java code back from the APK in order to study the app behavior, to replicate it and to extract the encrypted information. To encrypt the passwords, *Dolphin Browser* used to adopt a static key, which was obtainable by simply looking at the decompiled bytecode. Newer versions of the browser derive the key from the *android_id* of the device, generated during the first boot, whose use is encouraged by the Android Developers community to generate device-specific passwords.

We were able to obtain the decrypted passwords from the *password.db* decompiling the app and studying its behavior. In the same way, a malware that managed to obtain root privileges can access the database and decrypt all the user credentials.

3

SELinux Policy Model

SELinux uses a *closed* policy model, denying every access request that is not explicitly permitted. The SELinux policy is defined using rules, which produce a set of authorizations. The SELinux model is quite rich and offers a number of features that increase its expressive power and flexibility. For instance, SELinux is able to manage a Multi-Level Security model, with the representation of sensitivity labels and categories. These features are used in some systems that rely on SELinux (e.g., Samsung Knox), but they are not currently used in AOSP, which is our reference platform. We then propose a simpler model that allows us to better characterize our approach.

3.1 Model specification

The model uses names with an “av” prefix, like avType instead of “type”, to provide a more precise definition. In the remainder of the paper, we will sometimes used the simpler terms (i.e., type instead of avType) when we see no ambiguity in their use. The “av” prefix stands for “access vector” and is used in SELinux to characterize the rules defining the policy, called AV Rules. The basic elements of this model are:

avType: represents an identifier that can be used to describe both the subject and the target of an authorization; an avType denotes a security domain or the profile of a process or resource in the system; the avType is used to build labels for processes and resources.

avClass: represents the kind of resource (e.g., file, process) that will be the target of an authorization; an implementation of SELinux in a system will

have to provide in its setup a set of avClasses consistent with the variety of resources that the system is able to manage.

avPermission: represents the possible actions a source can apply on a target of a specific avClass, specified in the setup of SELinux; every avClass *cl* has its own set of avPermissions, represented by *cl.permissions* (e.g., *file.permissions* = $\{read, write, execute, \ldots\}$).

In order to give a formal representation of the SELinux policy, we introduce the concept of avAuthorization.

Definition 16. *Given a set T of avTypes, a set C of avClasses, and a set P of avPermissions, an **avAuthorization** a is a quadruple* ⟨ source, target, class, action ⟩, *where:*

source $\in T$ *represents the process (the security principal of the authorization);*

target $\in T$ *is associated with the object that is accessed by the* source*;*

class $\in C$ *denotes the type of resource that is accessed in the operation;*

action $\in \{P \cap \text{class.permissions}\}$ *is the specific avPermission, which has to be compatible with the avClass.*

Each avAuthorization describes a specific request that is permitted in the system.

Example 9. *Consider an app whose process is associated with the avType myapp that wants to read a file both in the internal and external sdcard. The required avAuthorizations are as follows:* ⟨ myapp, sdcard_internal, file, read ⟩, ⟨ myapp, sdcard_external, file, read ⟩.

Definition 17. *An* **avAuthzPolicy** *is a set of avAuthorizations.*

The avAuthzPolicy is derived from the specification of a collection of *avRules*. avRules can be positive or negative, support the use of patterns for the specification of sources and targets, and may use avAttributes.

Definition 18. *An* **avAttribute** *is an identifier that can be used in the construction of avRules. It can be used to support the definition of collections of avAuthorizations. The collection of avAttribute identifiers in a system must be separate from the domain of avTypes.*

Definition 19. *Given a set T of avTypes, a set C of avClasses, a set A of avAttributes, and a set P of avPermissions, an* **avRule** *is a quintuple* ⟨ ruleType, ruleSource, ruleTarget, ruleClass, ruleAction ⟩, *where:*

ruleType *is either* allow *or* neverallow*;*[1]

[1]SELinux also supports the *auditallow* and *dontaudit* rules, which describe the configuration of the auditing services. The model we describe can be easily extended to manage these services.

ruleSource *is a* pattern*, structured in two parts: (a) a set of positive elements* $p_i \in T \cup A$*, and (b) an optional set of negative elements* $n_i \in T \cup A$*;*

ruleTarget *is a* pattern*, with the same structure as the* ruleSource*;*

ruleClass *is a set of avClasses, i.e., each* $c_i \in C$*, denoting the types of resource that are considered by the avRule;*

ruleAction *is a set of avPermissions, where we assume that each* $a_j \in \cap_i c_i$.permissions*, i.e., all the elements have to be compatible with all the avClasses specified in the avRule.*

Each avRule can be represented in a textual form, listing the five components following the order above. The textual notation for patterns keeps all the elements within curly braces, preceding the set of negative elements with a "-" character; a colon separates the ruleTarget from the ruleClass.

Example 10. *In order to group the common avAuthorizations granted to myapp it is possible to create the avAttribute sdcard and assign it to the sdcard_internal and sdcard_external (through the use of* typeattribute*, defined below). Then, the avAuthorizations defined in Example 9 can be derived by the following avRule:* ⟨ allow, myapp, sdcard, file, read ⟩.

The avRules provide a higher-level representation of avAuthorizations. Every *allow* avRule is managed with an expansion of the sets associated with the source, target, class, and action. In general, a cartesian product is computed of all the elements in the positive part. The negative portion of each pattern is used to specify exceptions in the consideration of the positive portion of the pattern.

Example 11. *In order to provide myapp the avAuthorizations needed to create and write files and directories labeled with an avType that has the avAttribute sdcard, defined in Example 10, with the exception of the avType sdcard_internal, the required avRule is as follows:* ⟨ myapp,
{ sdcard -sdcard_internal }, { file dir }, { create write } ⟩.

An element that has a strong impact on the derivation of the low-level avAuthzPolicy is the definition of the association between avTypes and avAttributes.

Definition 20. *The* typeattribute *statement associates an avType with one or more avAttributes. The syntax of* typeattribute *appears in Table 3.1. The interpretation is that the avType will be associated with all the privileges that have been granted to the avAttribute.*

Definition 21. *An* **avRulePolicy** *is a set of avRules and* typeattribute *statements.*

The avAuthzPolicy is obtained by a compilation of the avRulePolicy. The compilation is executed by the *checkpolicy* tool, with a sequence of three steps:

(a) the typeattribute statements are processed, creating new avRules for every avRule where the avAttribute appears, replacing the avAttribute with the avTypes; (b) all the *allow* rules are expanded, producing a set of avAuthorizations; (c) all the *neverallow* rules are expanded and the policy is checked for the presence of conflicts: if even one avAuthorization produced by the expansion of *neverallow* rules matches an avAuthorization produced by *allow* rules, the compilation stops and an empty policy is produced.

3.2 Requirements

Analyzing the introduction of appPolicyModules in the management of per-app security, we need to consider the different cases that emerge from the combination of the *system policy* and an appPolicyModule. From the model presented above, we note that every avAuthorization defined in an SELinux policy has a *source* avType and a *target* avType. These types may be defined in either the *system policy* or the appPolicyModule. We then have four types of avAuthorization, depending on the origin of the source and target domains. Each configuration is associated with a specific requirement that must be satisfied by appPolicyModules.

Each requirement will be described and formalized using a simple formalization that expresses each requirement as a constraint on the relationship between the system avAuthzPolicy AV, derived from the system avRulePolicy S, and the avAuthzPolicy AV', obtained after the integration of an appPolicyModule M with S. We will show in Section 3.3 that our proposed language and restrictions for the appPolicyModules satisfy all the requirements. We assume that there is an avType that describes the domain of safe-to-use resources and actions, called *untrusted_app*, which protects system resources from the abuse of third-party apps (this is the name actually used in the current SEAndroid policy).

An example referring to the *Dolphin Browser* app will also be presented for every requirement, to clarify the impact in the design of the policy. To denote the type of avAuthorization, we use a compact notation where S and A represent respectively the system and appPolicyModule origin of the avType, with this structure: *source* $\rightarrow$ *target*.

Req1 ($S \rightarrow S$), No impact on the system policy: the app must not change the system policy and can only impact on processes and resources associated with the app itself.

An appPolicyModule is intended to extend the system policy and to be managed by the same software modules that manage the system policy. Since third-party apps can not be trusted a priori, it is imperative that the provided appPolicyModule must not be able to have an impact on privileges where source and target are system types.

More formally, AV must be contained into AV' and all the avAuthorizations appearing in $AV' - AV$ have to present as source or target avTypes defined in M (a set represented by notation $M.newAvTypes$).

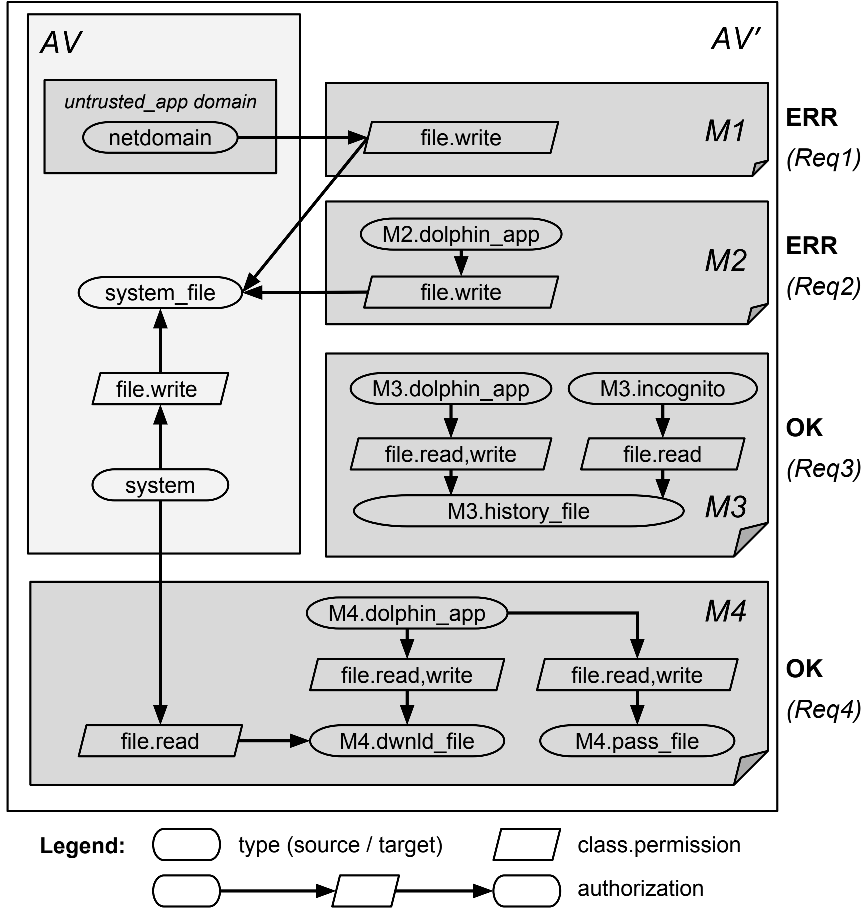


Figure 3.1: Examples of both compliant and non-compliant modules to illustrate requirements.

I.e., $AV \subseteq AV' \ \wedge \ \forall a \in (AV' \setminus AV) \ \rightarrow$
$\quad a.source \in M.newAvTypes \vee a.target \in M.newAvTypes$

Example 12 (Figure 3.1, M1)**.** *The APM associated with* Dolphin Browser *can specify access privileges only on its own resources, such as its own password database, but must not be able to specify authorizations on system resources. Without this restriction the appPolicyModule could provide* unstrusted_app *write access to the type* platform_app_data_file *and corrupt the system resources, or enhance the privileges of system resources that the app can access, creating unpredictable vulnerabilities. Consider the appPolicyModule* M1 *in Figure 3.1. The module defines an authorization that modifies the behavior of the system policy because* netdomain *should not have write access to files labeled as* system_file*. Thus* M1 *will not be installed due to the violation of* Req1*.*

Req2 ($A \rightarrow S$), No escalation: the app cannot specify a policy that provides to its types more privileges than those available to *untrusted_app*.

New domains declared in an appPolicyModule must always operate within the boundaries defined by the system policy as acceptable for the execution of apps. When a new application is installed, its domain has to be created "under" the *untrusted_app* domain, so the system policy can flexibly define the maximum allowed privileges for third-party apps.

More formally, all the avAuthorizations introduced by the appPolicyModule M that have an avType t belonging to the avTypes defined by M as a source will be contained in the set of avAuthorizations that have the system-defined *untrusted_app* avType as source.

I.e., $\forall\ a' \in (AV' \setminus AV)\ |$
$a'.source \in M.newAvTypes \wedge a'.target \notin M.newAvTypes \rightarrow$
$\exists a \in AV\ |$
$(a.source = untrusted_app \wedge a'.target = a.target \wedge$
$a'.class = a.class \wedge a'.action = a.action)$

The constraint forces the avAuthzPolicy to assign to all the types introduced by the appPolicyModule a set of authorizations that corresponds to privileges available to the *untrusted_app* avType. Then, each privilege must have the same class and action of a privilege already assigned to *untrusted_app*.

Example 13 (Figure 3.1, M2)**.** *As highlighted by appPolicyModule* M2 *in Figure 3.1, appPolicyModules can only request a subset of the privileges granted to the* untrusted_app *domain. The APM* M2 *tries to give* M2.dolphin_app *the privilege of writing files labeled as* system_file*, that is not granted to the* untrusted_app *domain. Thus* M2 *will not be installed due to the violation of* Req2.

Req3 ($A \rightarrow A$), Flexible internal structure: apps may provide many functionalities and use different services (e.g., geolocalization, social networks). The appPolicyModule has to provide the flexibility of defining multiple domains with different privileges so that the app, according to the functionality in use, may switch to the one that represents the "least privilege" domain needed to accomplish the job, in order to limit potential vulnerabilities deriving from internal flaws.

Greater flexibility derives from the possibility to freely manage privileges for internal types over internal resources, building a MAC model that remains completely under the control of the app.

More formally, there can exist a pair of new avTypes t and t' introduced by M such that in $AV' \setminus AV$ t receives a privilege that t' does not have.

I.e., if $\exists t \in M.newAvTypes \wedge t' \in M.newAvTypes \wedge$
$a \in AV' \wedge a.source = t \not\rightarrow$
$\exists a' \in AV'\ |\ a'.source = t' \wedge a.target = a'.target \wedge$
$a.class = a'.class \wedge a.action = a'.action$

Example 14 (Figure 3.1, M3)**.** Dolphin Browser *provides the* anonymous surfing *(*incognito*) mode, which allows the user to surf the web without storing permanently the history and the cookies, and in general aiming at leaving no trace in persistent memory of the navigation session. In order to enhance its security and protect the user even from possible app flaws, the appPolicyModule could specify a switch of context (i.e., it may change the SELinux domain associated with its process) when the user enters the incognito mode. In Figure 3.1 the APM* M3 *specifies that the domain* M3.dolphin_app *can read and write files labeled as* M3.history_file*, while the domain* M3.incognito*, used during anonymous surfing, drops the privilege of writing the files, preventing the leakage of resources that may leave a trace of the navigation session.*

Req4 ($S \to A$), Protection from external threats: users of mobile devices may unconsciously install malware apps from untrusted sources that, exploiting some security vulnerabilities, could compromise the entire system or other apps (e.g., steal user information). To mitigate the risk, an appPolicyModule should provide a common way to isolate the app's critical resources. The use of MAC support offers protection even against threats coming from the system itself, like a malicious app that abuses root privileges.

The app can protect its resources from other apps, specifying its own types and defining in a flexible way which system components may or may not access the domains introduced by the APM. This requirement depends on the ability of the MAC model to let app types be protected against system-level elements, an aspect that SELinux supports and not available in classical multi-level systems, which assume a rigid hierarchical structure. Indeed, in the SELinux policy model every privilege has to be explicitly authorized and new avTypes are not accessible by system avTypes unless a dedicated rule is introduced in the appPolicyModule.

More formally, the appPolicyModule M can introduce an avAuthorization that gives to an avType introduced by M a privilege that is not necessarily available to a type in the system policy.

I.e., if $\exists\, t \in M.newAvTypes \wedge a \in AV' \wedge\, a.source{=}\, t \not\to$
$\exists a' \in AV\ |a'.source \notin M.newAvTypes \wedge a.target = a'.target \wedge$
$a.class{=}\, a'.class \wedge a.action{=}\, a'.action.$

Example 15 (Figure 3.1, M4)**.** *The* Dolphin browser *can grant to the* system *type the privilege to read the* dwnld_file *files, used to label the downloaded files, while it prevents the access to the* pass_file *files used to label the password file.*

There are other environments where SELinux is used, like the Redhat Fedora distribution of Linux, that already supports SELinux modules, but the requirements presented above do not apply to them. The reason is that the trust assumptions are different. The modules used in Redhat Fedora permit to structure the security policy, they are trusted and free to revise in arbitrary ways the system policy. Modules in Android are not trusted and it is mandatory that they cannot be used to introduce vulnerabilities in the system.

Additional requirements, not associated with a formal treatment, have also to be considered.

- Not all the developers have the knowledge or are interested to secure their apps with SELinux, so in order not to impede the development they have to experience the same development and installation process, with no impact on their activities. This requirement will be considered in Chapter 4.
- In order to facilitate the deployment, the solution has to be compatible with the implementation of SELinux offered by SEAndroid. This is considered in Chapter 5.

3.3 Policy Module Language

We now present the concrete structure of appPolicyModules. We introduce the subset of the SELinux statements used in their definition and describe the additional statements that will be automatically added to the appPolicyModule by a pre-processor. A critical design requirement is the compatibility with the SELinux implementation available today, which facilitates the adoption of the proposed approach.

Each module presents a head and a body (see right side Figure 4.1). The head describes all the identifiers that the appPolicyModule reuses from the system policy. This is represented by the *require* statement. In case a name appears that is not known to the system, the compilation fails.

The body of the appPolicyModule can make use of the following SELinux statements: *typebounds*, *type*, *attribute*, *typeattribute*, *allow*, *neverallow*, and *typetransition*. These statements are the only ones that can be used in the definition of the appPolicyModule. The syntax for these statements is succinctly presented in Table 3.1.

The *typebounds* statement permits to specify that the collection of privileges of the bounded avType has to fall within the boundaries of another avType. The *typebounds* statement will raise an exception when an *allow* rule introduces a privilege for a bounded type in the source that does not match an existing rule for the bounding type.

```
type dolphin_app;
type dolphin_app_incognito;
typebounds untrusted_app dolphin_app, dolphin_app_incognito;
allow dolphin_app app_data_file:file {read write};
allow dolphin_app_incognito app_data_file:file {read};
```

Listing 3.1: Example of use of *typebounds*. In the system policy there is a rule `allow untrusted_app app_data_file:file {read write};`

The evaluation of compatibility takes into account the presence of other *typebounds* statements in the target, considering as correct the use in the target of an avType that is bounded by the type appearing in the higher-level rule. In

the example in Listing 3.1, the verification by typebounds is satisfied, because both the *allow* rules use in the target an avType that is considered compatible with the *untrusted_app* type. It is useful to emphasize that the *typebounds* statement does not assign the authorizations to the bounded domain, it only sets its upper bound. This is a core principle in our scenario, where policy writers are outside of the trust domain of the core system resources.

The *type* statement permits to introduce new avTypes. To avoid name conflicts between types defined in different modules, the pre-processing adds a prefix that derives from the app name to every identifier (we omit in the examples the representation of this step). If it does not already appear in the module, the pre-processing step will add a *typebounds* statement for every introduced type that will constrain the authorizations referring to types in the system policy to lie within the *untrusted_app* type. The *attribute* statement declares an identifier that can be used to define rules. SELinux policies make extensive use of avAttributes to provide a structure to policies. No constraint needs to be introduced on the definition of new attributes. Attributes produce an effect on the policy when they are used in the *typeattribute* statement, which has been presented above. The pre-processing checks that every *type_id* used in a *typeattribute* statement must be defined inside the appPolicyModule. Without this constraint, a module could violate *Req1* and *Req2*, compromising the system policy and possibly performing an escalation of privileges, by assigning attributes to the *untrusted_app* type. The *allow* and *neverallow* statements permit to create avRules. The pre-processing checks that all the avRules present as a source or target one of the avTypes and avAttributes introduced by the module.

Finally, the *typetransition* statement permits to describe the admissible transitions between avTypes at runtime. We introduce the constraint, checked by the pre-processor, that the avType defined as first parameter has to be an avType defined in the module. The *type_transition* statement is used to perform *object* and *domain* transitions.

- An *object transition* occurs when an object needs to be relabeled (i.e., a file label is changed).
- A *domain transition* occurs when a process with one avType (we call it *transition-startpoint*) switches to another avType (we call it *transition-endpoint*), enacting different avAuthorizations from the original ones. An app could define different domains with limited avAuthorizations and use them when performing specific actions. We note that for a domain transition to succeed, we must grant three different avAuthorizations to the transition-startpoint type.

With respect to object transitions, there is no need to further constrain them, because the process domain must have the corresponding avAuthorizations to be able to create objects with the new label. With respect to domain transitions, when a type transition is authorized and the transition-startpoint type is given the three additional authorizations, the transition-startpoint type is actually able to benefit from all the authorizations that have the transition-endpoint as

Table 3.1: Simplified SELinux syntax used in APMs.

Statement	**Syntax**
attribute	attribute *attribute_id* ';'
type	type *type_id* (',' *attribute_id*)* ';'
typeattribute	typeattribute *type_id attribute_id* (',' *attribute_id*)* ';'
typebounds	typebounds *bounding bounded* (',' *bounded*)* ';'
typetransition	type_transition *type_id type_id*':' '{'*class_id*+'}' *type_id* ';'
allow	allow '{'*pattern*+ ('-'*pattern*)* '}' '{'*pattern*+ ('-'*pattern*)* '}' '{'*class_id*+'}' '{'*perm_id*+'}' ';'
neverallow	neverallow '{'*pattern*+ ('-'*pattern*)* '}' '{'*pattern*+ ('-'*pattern*)* '}' '{'*class_id*+'}' '{'*perm_id*+'}' ';'

The element *pattern*=(*type_id*—*attribute_id*) is not included in the SELinux statements; we use it here to provide a more readable description of the syntax.

the source. This is a potential risk in the definition of the policy, because the *typebounds* statement does not extend its evaluation to the consideration of the types that are reachable through type transitions. The current AOSP system policy does not give to *untrusted_app* any type transition privilege, and at the moment there is no danger, but to avoid any risk we enforce the constraint to accept in the appPolicyModule only type transitions that have a transition-endpoint bounded within *untrusted_app*.

3.3.1 Correctness

We want to show that the appPolicyModules will satisfy the four requirements described in Section 3.2.

With respect to *Req1 (S→S), No impact on the system policy*, we note that the appPolicyModule statements do not have an impact on the system policy, because all the *allow* and *neverallow* statements have to specify as source or target a new avType, guaranteed to be outside of the system policy.

The correctness with respect to *Req2 (A→S), No escalation* is guaranteed through the use of the *typebounds* statements associated with all the avTypes that appear as source in the *allow* statements. It is to note that the *neverallow* rules do not have to be considered here, because they may only cause the rejection of the appPolicyModule by the compiler, but they cannot lead to the escalation of privileges for the new avTypes. The consideration by the compiler of the *typebounds* statements indeed verifies that each *allow* rule r' in the appPolicyModule that refers to system types has a corresponding *allow* rule r

associated with *untrusted_app*.

The respect of *Req3* and *Req4* can be demonstrated with a simple example of an appPolicyModule that shows the desired behavior. Requirement *Req3 (A→A), Flexible internal structure* is satisfied by the example in Listing 3.1, which shows two new avTypes *dolphin_app* and *dolphin_app_incognito*, associated with distinct privileges.

Requirement *Req4 (S→A), Protection from external threats* is supported by the same example: without an explicit rule giving permission, a process associated with *untrusted_app* is not authorized to access files associated with *dolphin_app*.

4 Mapping Android Permissions

The introduction of *appPolicyModules* improves the definition and enforcement of the security requirements associated with each app. However, in the approach presented in the previous section, we assumed that the extension to the MAC policy has to be defined by the developer, who knows the service provided by the app and its source code. Due to the size of the community, we can expect that many app developers will either be unfamiliar with the SELinux syntax and semantics, or know SELinux but not want to use it, avoiding the introduction of strict security boundaries to the app beyond those associated with *untrusted_app*. There is also the risk generated by the presence in devices of a variety of versions of the system policy and the need to guarantee that the appPolicyModule is compatible with it.

However, we observe that the app developers that can be expected to be most interested in using the services of the MAC model are expert developers responsible for the construction of critical apps (e.g., apps for secure encrypted communication, or for key management, or for the access to financial and banking services). This community is possibly small, but their role is extremely critical. They can be expected to overcome the obstacles to the use of appPolicyModules. In addition, the deployment of the policy modularity services opens the door to a number of other services. We consider here how it is possible to use them to enforce a stricter model on the management of Android permissions, relying on the automatic generation of appModulePolicies, solving all the issues identified above.

Looking at the workflow to build an app, developers are already familiar with the definition of security requirements in the *AndroidManifest.xml*, through the use of the tag *uses-permission*. In fact, in order to access system resources (e.g., access to the user's current location) the app has to explicitly request the associ-

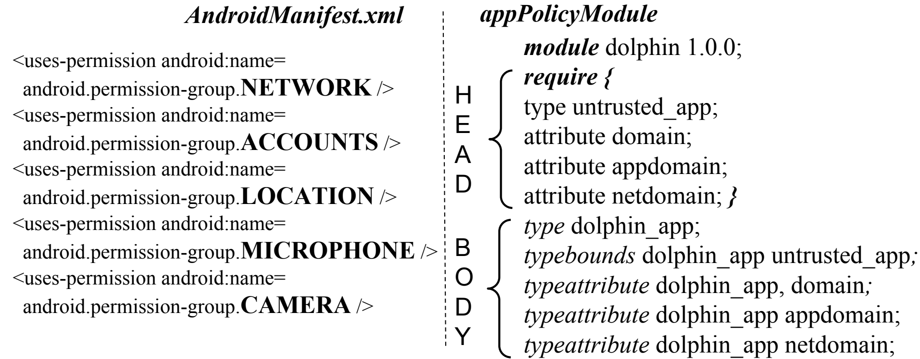


Figure 4.1: Generation of the Dolphin browser appPolicyModule starting from the permissions in the app manifest.

ated Android permissions (e.g., *android.permission-group.LOCATION*), which correspond
both to a set of concrete actions at the OS level and to a set of *avPermissions* granted at the SELinux level (e.g., *open*, *read* on files and directories). The system already offers both a high-level representation and a low-level representation of the privileges needed to access a resource, but they are not integrated and what happens, in the absence of policy modularity, is that the app is associated with the *untrusted_app* domain, which is allowed to use all the actions that correspond to the access to all the resources that are invokable by apps, essentially using for protection only Android permissions. The integration of security policies at the Android permission and MAC levels offers a more robust enforcement of the app policy.

This can be realized introducing a mechanism that bridges the gap between different levels, through the analysis of the high-level policy (i.e., the permissions asked by the app within the Android Permission Framework) and the automatic generation of an appPolicyModule that maps those Android permissions to a corresponding collection of SELinux statements. The generator starts from the representation of the app security requirements expressed in the *AndroidManifest.xml*, builds a logical model of the structure of the *appPolicyModule*, and it finally produces the concrete implementation of the *appPolicyModule* and verifies that all the security restrictions are satisfied.

A necessary step in the construction of the mechanism is the identification of a mapping between policies at the distinct levels. The Android Permission Framework contains more than 200 permissions and most of them present a mapping between the Android permission and a dedicated SELinux domain, already specified in the system policy. The current system policy does not cover all the permissions; e.g., the *downloads*, *calendar*, and *media content* resources are associated with the single *platform_app_data_file* type. We expect this aspect to be manageable with a revision of the policy. However, there is a number of Android permissions that can only be partially supported by this mecha-

nism due to current limitations in the security mechanisms provided by internal components (e.g., SQLite).

To summarize, it is already possible to capture most Android permissions in a precise way and some of them with some leeway, leading in all cases to a significant reduction in the size of the MAC domain compared to what would otherwise be associated with an app. We provide in Figure 4.1 an example of the appPolicyModule that would be generated for the Dolphin browser described in Section 2.2.1. We note that every app will have to be associated with the *domain* and *appdomain* attributes, which provide all the basic privileges required to let an app execute in the system. The head of the module will then have to introduce the *require* declarations that specify this attribute, together with the *untrusted_app* type and the *netdomain* attribute. The body of the module introduces the *dolphin_app* type, the *typebounds* and all the MAC privileges required to access the network and other resources/services. Due to the current SELinux policy structure the access to the *ACCOUNTS*, *LOCATION MICROPHONE* and *CAMERA* services are mitigated by the *system_server*. The *system_server* is the core of the Android system which manages most of the framework services. The access to the requested services is granted by the *system_server* to the *appdomain* type and through the use of the rule *typeattribute dolphin_app appdomain* the *dolphin_app* type inherits these privileges.

In general, with the availability of appPolicyModules, the system could evolve from a scenario where each app is given at installation time access at the SELinux layer to the whole *untrusted_app* domain, to a scenario where each app is associated with the portion of *untrusted_app* domain that is really needed for its execution, with a better support of the classical “least-privilege” security principle.

5

Implementation

The work done by Smalley et al. in [109] represents the basis for our work. We have introduced a set of extensions in order to enrich the current implementation and manage appPolicyModules. We now provide a description of the challenges to enable the concrete use of appPolicyModules in Android. The system has been implemented with an open source license, extending the current version 4.4.4 of the AOSP (link omitted for the anonymity constraints); adaptation to Android L is planned as soon as it will be released.

The current SELinux implementation for Android spans different levels of the Android stack. At the *Application Framework* level, the *SELinux* class provides access to the centralized Java Native Interface (JNI) bindings for SELinux interaction. The *android_os_SELinux.cpp* file represents the JNI bridge. At the *Libraries* level, the SELinux implementation consists of the *libsepol* and *libselinux* libraries. The former provides an API for the manipulation of SELinux binary policies. The latter provides the APIs to get and set process and file security contexts and to obtain security policy decisions.

The first challenges to the integration of appPolicyModules in Android appeared in the adaptation of the current SELinux libraries and in the addition of the libraries needed to build, link and check the appPolicyModules. These libraries are part of the full SELinux environment and are included in the major Linux distributions, but the activation of policy modules in SELinux for Android required more than a simple cross-compilation.

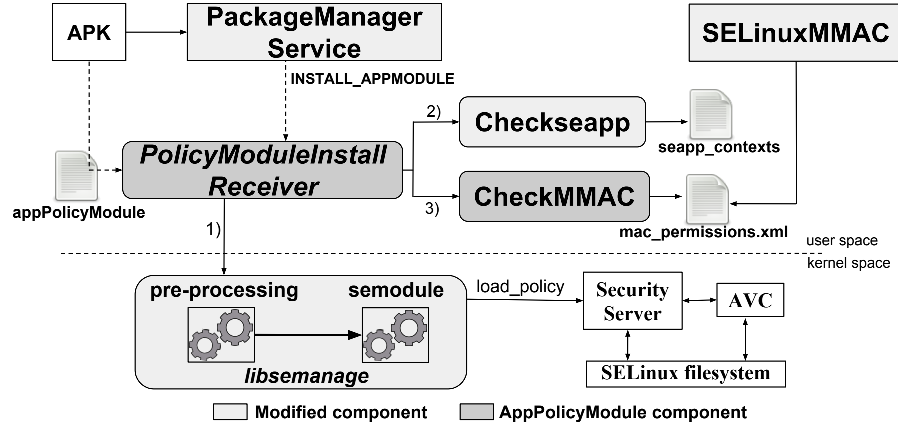


Figure 5.1: appPolicyModule Architecture.

5.1 Changes to SELinux

The work we did at the SELinux level can be structured into four major activities.

First, the *libselinux* library was modified with the introduction of additional features needed by the *libsemanage* library, such as the *selinux-config.c* module. We modified the *checkpolicy* tool in order to build automatically the binary policy at version 26 (standard SELinux implementations support version 29 of the binary policy). In the current SEAndroid implementation the binary policy version range is between 15 and 26. This constraint is enforced by the *load_policy* method when a policy reload is triggered.

Second, the *libsemanage* library, which provides APIs for the manipulation of SELinux binary policies and binary policy modules, was adapted to fulfill the new requirements. Due to the differences in scenario, architecture and requirements, some functions, such as the *genhomedircon* service, were disabled. The *genhomedircon* service is used to generate file context configuration entries for user home directories based on their default roles and is run when building the policy. However, in Android, though there is the possibility to create several users for a single device, they do not have a *home* directory.

Third, the source code of the *semodule* executable was extended in order to correctly interact with the modified version of the *libsemanage* library. The *semodule* tool is used to manage SELinux policy modules, including installation, upgrade, listing and removal of modules. The *semodule* tool may also be used to force a rebuild of the policy from the module repository and/or to force a reload of the policy without performing any other transaction.

Figure 5.1 shows an abstract representation of the complete architecture introduced in order to manage the appPolicyModules. Fourth, in addition to the modifications on the set of SELinux libraries, to meet the requirements introduced in Section 2 a pre-processing phase was introduced. This phase

supports the creation of constraints introduced in Section 3. Thanks to the modularity provided by SELinux, we were able to implement the pre-processing phase reusing several SELinux components.

5.2 Changes to Android

The second set of challenges concerns the app installation process, which starts from the APK file that contains the app. The *PackageManagerService* class provides the APIs that actually manage app installation, uninstallation, and update. The *PackageManagerService* component provides the functions to parse the *APK* file and to assign the SELinux label to the app. The label is retrieved by the *SELinuxMMAC* class from the *mac_permissions.xml* file. The file maps the app certificate to a SELinux label. In the current AOSP version, all third-party apps are assigned to the default stanza of the *mac_permissions.xml* file, regardless of their certificate. To address this limitation and assign to the app the right SELinux type, we introduced a new install service, named *PolicyModuleInstallReceiver*. This service is triggered by an Intent and manages the installation workflow of an appPolicyModule. The workflow is structured as follows:

1. trigger the installation of the policy module, update the SELinux policy and check its correctness;
2. update the *seapp_contexts* file used to label app processes and app package directories;
3. update the *mac_permissions.xml* file used by *SELinuxMMAC* to retrieve the type to assign to the app. This file is used in conjunction with *seapp_contexts*.

5.2.1 Update SELinux policy

The use of the *PolicyModuleInstallReceiver* service requires to broadcast an *intent.action.INSTALL_APPMODULE* intent. When the service receives the intent it performs a JNI call to the *libsemanage* library, which validates (i.e., pre-processing phase), links, expands the module (i.e., *semodule* tool) and triggers the reload of the binary policy.

5.2.2 Update seapp_contexts

In order to meet the requirements Req3 and Req4 described in Section 3.2 the app processes and the app package directories have to be labeled accordingly to the appPolicyModule. In the current AOSP implementation the security context assigned to app processes, respectively app package directories, is retrieved from the seapp_context file by the *selinux_android_setcontext*, respectively *selinux_android_setfilecon2*.

In order to manage the addition of new entries in the *seapp_contexts* file the *checkseapp* utility was extended. Currently, *checkseapp* is used during the AOSP build process to validate the *seapp_contexts* file against policy. The extension permits to dynamically manage the addition/removal of an entry in the *seapp_contexts* according to the domains defined in the appPolicyModule.

To allow AOSP components such as *Zygote* to spawn applications in the correct domain, an update of the *mac_permissions.xml* is needed.

5.2.3 Update mac_permissions.xml

This file is used to configure the install MMAC policy. More specifically this file is used in conjunction with the *seapp_contexts* file in order to determine the *seinfo* label to assign to the app. The *seinfo* value is subsequently used to determine the SELinux security context for the app process and its */data/data* directory based on the *seapp_contexts* configuration.

The information needed to build a stanza for the *mac_permissions.xml* file are (i) the app's X.509 certificate and (ii) the *seinfo* label. The stanza is built on the fly by the *checkMMAC* Java class retrieving the information directly from the parsed *apk* and the entry added to the *seapp_contexts* file. After stanza creation, an update is triggered to insert and refresh the whole *mac_permissions.xml* file. This is needed in order to let *SELinuxMMAC* retrieve the right *seinfo* label for the new app.

5.3 Performance

As it was clearly expressed in the design of SEAndroid [109], it is necessary to have a minimal overhead in terms of performance, both at app installation time and during regular system runtime. We executed a series of experiments for the evaluation of the performance impact of the techniques presented in this paper. Experiments have been run on a Nexus 7 2013 aosp_flo_userdebug 4.4.2_r2. The Android runtime used was ART.

5.3.1 Installation time

We evaluated the performance overhead of our approach at app installation time, due to the fact that the process to install an app was extended in order to manage appPolicyModules. Two different approaches were developed; the first one is consistent with the approach used for SELinux in Fedora. When a new module is ready to be installed, the *libsemanage* tool creates a new version of the policy and re-installs all the old modules plus the new one. If the new policy passes the checking step, then the new policy is stored into the system, otherwise a rollback occurs. This approach introduces a non negligible overhead, because it requires to reconsider the whole policy every time a new module is added to the system. This option is natural for SELinux in Fedora, because the modules are created by trusted entities and they are free to modify the system policy,

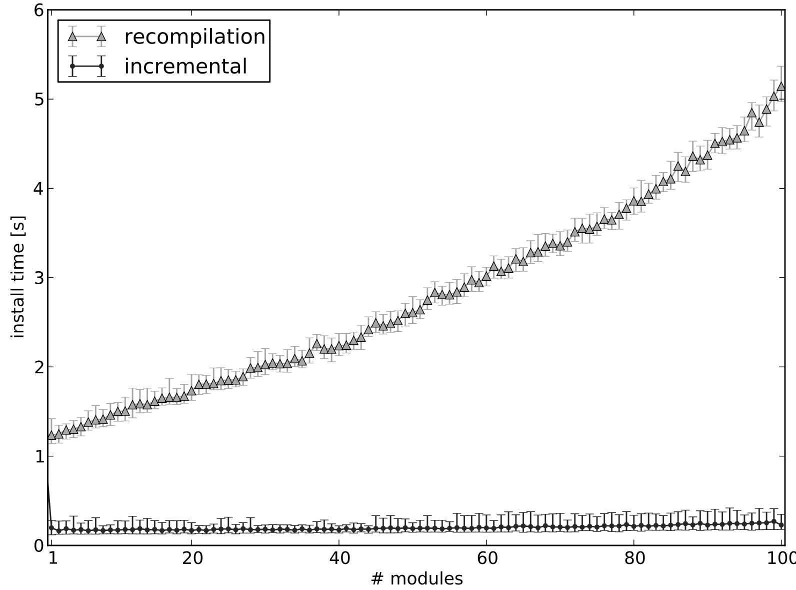


Figure 5.2: Installation time: comparison between the full recompilation and the incremental approach.

with a number of types in the existing binary representation that may have to be updated. As we have already discussed, the requirements associated with the use of appPolicyModules in Android are different, because in our scenario an appPolicyModule cannot modify the system policy (*Req1* in Section 3.2). In addition to the security benefit, this requirement also leads to a simplification of the management of policy modules, because the re-installation of the system policy and all the other modules is not needed. This permits to provide an "incremental" solution, with a significant reduction in the appPolicyModule installation time.

The graph in Figure 5.2 describes the time observed in a scenario where the current system policy has been extended with 100 modules, adding the modules one by one. The tests were run 100 times. Each element in the graph describes the range of measured values and the average. The observations in the upper part of the graph show that as the number of modules increases, the recompilation approach shows a significant increase in the compilation time, due to the fact that at each step the policy becomes larger and its full recompilation more expensive. The incremental approach instead shows a constant response time, with no observable impact deriving from the increase in the size of the overall policy. The average installation time for the incremental approach, which is the one to use, is near to 0.2 s, compatible with the requirements of real systems.

5.3.2 Runtime

We evaluated the performance overhead of our approach at runtime, considering two scenarios with different binary policy sizes.

Table 5.1: Binary policies used in the tests.

policy	**#rules**	**size**
sepolicy	1319	73KB
sepolicy +1000 APMs	35319	631KB

For runtime analysis we used two well known benchmark apps: (i) AnTuTu [4] by AnTuTu Labs and (ii) Benchmark by Softweg [110]. Under both benchmarks, we ensured that the same number of apps/services were loaded and running.

Table 5.2: AntuTu Benchmark (100 iterations), higher values are better

	sepolicy	**sepolicy +1000 APMs**
svm	1130.467	1132.867
smt	3334.533	3341.400
database	630.600	631.333
sram	1534.840	1538.533
float	1938.600	1939.200
snand	1159.320	1159.400
memory	1121.280	1120.800
integer	2285.933	2284.133

AnTuTu Benchmark is a popular Android utility for benchmarking devices. As it was explained in [109], the overhead introduced by SELinux is very limited and it only affects *sdwrite*, *sdread* and *database I/O* tests. The tests performed by Smalley et al. take into account a “static” policy. In our scenario the policy size is not static, but it changes at each installation and can potentially become quite large. However, experimental results highlight how the policy size does not affect the system performance. Table 5.2 shows the results of the benchmark. The impact of the larger policy is not detectable by the experiments.

Table 5.3 shows the results provided by *Benchmark* app developed by Softweg. Similarly to the results obtained by AnTuTu, SELinux does not affect CPU and graphics scores. For the filesystem and sdcard tests, the overhead introduced by the increased size of the policy is negligible. As highlighted by the *create and delete* tests, the time taken to create or delete 1000 empty files increased by less than 1 percent. As explained by Smalley et al. [109] the *create*

Table 5.3: Softweg Benchmark (100 iterations): for the Total File System score, higher values are better

	sepolicy	**sepolicy +1000 APMs**
Create 250 empty files	1.222 s	1.230 s
Create 1000 empty files	0.302 s	0.303 s
Delete 250 empty files	0.351 s	0.351 s
Delete 1000 empty files	0.130 s	0.130 s
Total file system score	342.835	341.158

and delete tests can be viewed as a worst case, since the overhead of managing the *security context* is not amortized over any real usage of the file.

6
Related Work

In the past few years a strong interest has been dedicated to the investigation of Android security. Several solutions have been proposed to increase the security of the system and to protect the apps and system components from a variety of threats. The central role of the proposal by Smalley et al. [109] has already been discussed. We summarize here other important contributions in the area.

Apex [87] allows the user to apply fine-grained control on the permissions requested by applications at installation time. *QUIRE* [29] provides a lightweight provenance system that prevents a special kind of privilege escalation attacks (i.e., the *confused deputy* attack). *TaintDroid*, proposed in [37] by Enck et al., provides functions to detect the unauthorized leakage of sensitive data. *TaintDroid* uses dynamic taint analysis (i.e., taint tracking) to monitor the exchange of sensitive information among third-party apps. While this solution try to identify the information leakage, our proposal goes one step further impeding the leakage at the SELinux level.

Kirin [38] is an extension to Android's application installer. The main security goal of Kirin is to mitigate malware contained within a single application. Kirin checks the permissions requested by applications at install-time. It denies the installation of an application if the permissions requested by the application encompass a set of permissions that violates a given system centric policy. However, due to its static nature, Kirin has to consider all potential communication links over the unprotected interfaces. This will stop any application from being installed, since applications can potentially establish arbitrary communication links over the unprotected interfaces. *Paranoid Android* (PA) [96] implements a virus scanner and a dynamic taint analysis system. It permits to detect viruses and runtime attacks exploiting memory errors such as buffer overflows. PA has the capability to check control flows thereby being able to

address privilege escalation attacks. However, PA requires the execution trace to be stored in secure storage (in order to prevent malicious modification) and impacts the performance of the device, since the execution trace has to be continuously transmitted to a remote analysis server that replays and analyses the execution trace. All of these proposals represent important security enhancements, but they all address access control at only one level of the Android stack. Other solutions, such as *FlaskDroid* [20], *TrustDroid* [19] and *XManDroid* [18] show greater similarity to our work.

FlaskDroid [20] is a security architecture for the Android OS that instantiates different security solutions. It is inspired by the concepts of the *Flask* architecture and is based on SEAndroid. *FlaskDroid* provides mandatory access control on both Android's middleware and kernel layers. This represents an enhancement in terms of the isolation that is provided between separate components, but the two MAC levels are not coordinated and largely use *booleans* in the SELinux policy. In the current SEAndroid implementation, the use of booleans inside the policy is strongly discouraged, for two main reasons: (i) it could introduce compatibility problems, and (ii) it could undermine the default security goals being enforced via SELinux in AOSP itself. Compared to our proposal, the focus of *Flaskdroid* is the security of system modules and the security of third-party apps is not supported. *Flaskdroid* does not permit to dynamically add policy modules without a recompilation of the entire policy.

TrustDroid [19] and *XManDroid* [18] provide mandatory access control at both the middleware layer and at the kernel layer. At the kernel layer, they rely upon *TOMOYO* Linux [49], a path-based MAC framework. *TOMOYO* supports policy updates at runtime, but the security model of SELinux is more flexible and supports richer policies.

RootGuard [103] is an enhanced root-management system which monitors system calls, to detect the abnormal behavior of apps (i.e., malware) with root privileges. It is composed by three components (i) *SuperuserEx*, (ii) *Policy storage database*, and (iii) *Kernel module*, which span the different levels of the Android architecture. The *SuperuserEx* is built on top of the open source Superuser app, the *Policy storage database* is used to store the *RootGuard* policy and the *Kernel module* introduces a set of *hooks* in order to intercept system calls. This implementation is similar to the one used by SELinux, but all the SELinux code is already in the mainline Linux kernel and provides a more robust solution.

7 Conclusions

Security is correctly perceived, both by technical experts and customers, as a crucial property of mobile operating systems. The integration of SELinux into Android is a significant step toward the realization of more robust and more flexible security services. The attention that has been dedicated in the SEAndroid initiative toward the protection of system components is understandable and consistent with the high priority associated with the protection of core privileged resources. Our approach is the natural extension of that work, which demonstrated a successful deployment, toward a more detailed consideration of the presence of apps.

The document shows that the potential for the application of policy modules associated with each app is quite extensive, supporting scenarios where developers define their own app policy, and scenarios where policies are automatically generated to improve the enforcement of privileges and the isolation of apps. The extensive level of reuse of SELinux constructs that characterizes the language for the appPolicyModules demonstrates the flexibility of SELinux and facilitates the deployment of the proposed solution. An analysis of the evolution of the official SEAndroid project confirms that appPolicyModules identify a concrete need and that Android is evolving in this direction.

Comparison with other proposals

A comparison between the metamodel presented in Section 2.2.2 with the most important proposals for policy languages available today is discussed in the following. It is interesting to evaluate the differences, as they help explain where is the innovation in the IT Security metamodel. It is also useful to provide a justification for the elements where the alternatives available in the literature and in the market offer greater expressive power. For brevity's sake, in this Section the word metamodel will be used to identify the IT Security metamodel.

A.1 XACML

The basic structural model underlying the XACML [89] language is presented in Figure A.1. It is possible to clearly identify a correspondence between the IT Policy metamodel and the XACML model. There are natural correspondences between entities that present identical name in both models: PolicySet collects instances of Policy entities in both cases. The XACML Rule entity corresponds to a generalization of the ITSystemAuthorization and ITRoleAuthorization entities in the metamodel. The XACML Effect entity is represented in the metamodel as an attribute of the ITSecurityAssociation entity, supporting the specification of positive and negative authorizations. The XACML Target entity presents a strong relationship with the metamodel Target entity; there is a difference in the way it is managed, as in XACML the target can be associated with instances of entities PolicySet, Policy, and Rule, whereas in the IT metamodel the Target entity is only associated with the metamodel Policy entity. In terms of expressive power the two structures are equivalent. In both models the

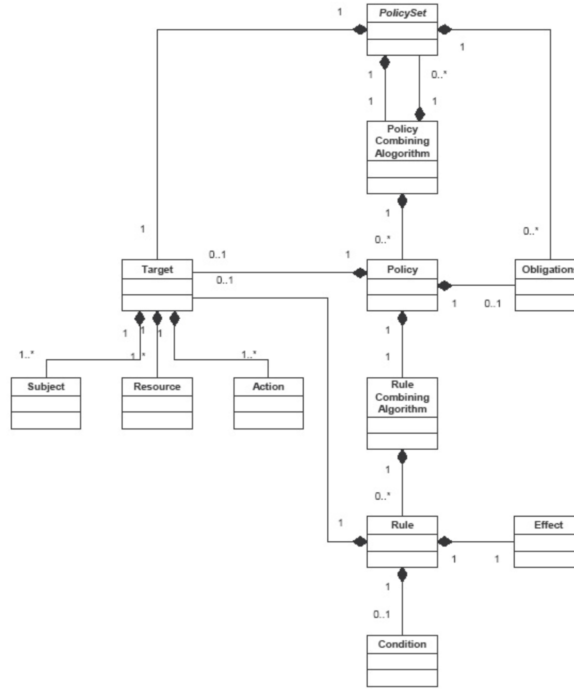


Figure A.1: XACML Language Model

Target entity supports the specification of restrictions on the elements that will be involved in a specific policy or rule. The XACML model offers a rich integration with a language like XPath and offers a number of predefined predicates, whereas in the metamodel support is offered for the representation of regular expressions on the properties of the entities involved in a Policy. XACML in general offers also a Condition entity that does not have a direct correspondent in the metamodel.

The limitation on the support for conditions in the metamodel represents the most significant limitation of this proposal compared to XACML. The motivation for this difference is the obstacles that would be introduced in the high-level analysis of the IT Policy from the integration with a rich and expressive declarative language for the representation of constraints. The choice of a rich integration with XPath works well for XACML, which has been designed to be the target of the representation of policies that a Policy Decision Point (PDP) can then understand and process. In that case, a sophisticated analysis of the structure of each policy and the relationships with the elements of the landscape have a lower priority than the ability to support a flexible Attribute-based Access Control paradigm. For the project, it is instead crucial to support a clear correspondence between elements appearing at different abstraction levels. It is also important to support advanced reasoning functions, that are the

main ingredient for the realization of harmonization services, operating within the IT level and across the distinct levels. These functions are difficult to integrate with a flexible language, because the support for flexible conditions would create significant obstacles to the efficient and robust identification of entities, a prerequisite for the concrete verification of policy correctness.

On the other hand, the metamodel offers several features that are not present in XACML. There are aspects that go beyond the specific representation of authorizations, like the representation of authentication conditions and the specification of channel protection, all beyond the expressive power of XACML. For the representation of the specific features of authorizations, the metamodel offers a richer and more structured characterization of the subjects, with the hierarchy associated with the metamodel ITPrincipal entity. Also, the metamodel offers a clear distinction between role authorizations and system authorizations, supporting in a direct way the representation of Role-Based Access control models, which are supported in XACML only relying on a profile that requires to adapt the basic structure of XACML to the representation of the two kinds of authorizations.

A.2 PCIM

The PCIM initiative [15,30] has the goal of providing a standard for the representation of policies in a format well integrated with the structure of the Common Information Model (CIM) proposal. The CIM model is considered an important reference, as it represents the most used solution for the description of the landscape of systems in a way that aims to be platform- and vendor-independent. Specializations of the CIM proposed by the different vendors unfortunately limit the degree of interoperability that is actually provided.

The PCIM model has as central component the representation of policies as collections of condition-action rules. The rules, in a way similar to the metamodel and to XACML, are organized into policies and policy sets. The conditions are boolean expressions over predicates. Predicates express conditions on properties of the CIM schema, or can use variables specified and updated in the actions of rules. Conditions can also express time restrictions. Figure A.2 shows the overall structure of policies and policy sets. Figure A.3 shows the policy condition model. PCIM offers an XML representation.

A specific attention has been paid to the support for the representation of access control. The specification relies on the use of a number of rules, which can either describe authentication or authorization restrictions. The model was one input to the design of the metamodel. The first component of this proposal is represented in Figure A.4.

The model presents several concepts that appear in the metamodel. A few relationships among entities are represented in a different way in the two models (e.g., the identities are directly related with the system supporting the identity/account, whereas in the metamodel we assume to represent this relationship introducing an authentication rule). The different structure does not lead to a

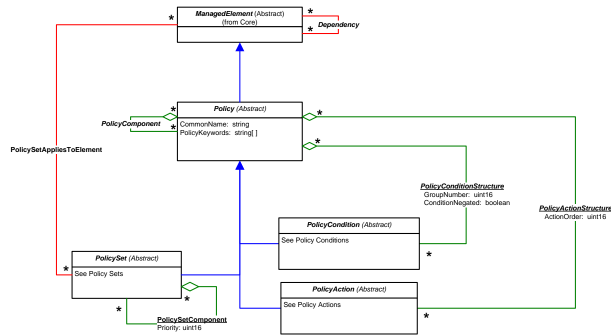


Figure A.2: PCIM: the general policy model.

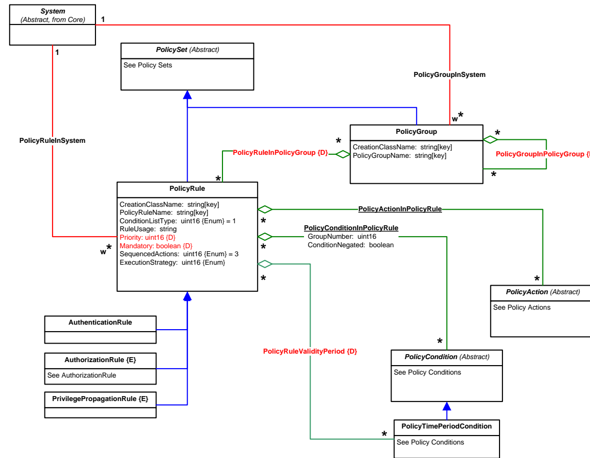


Figure A.3: PCIM: policy condition model.

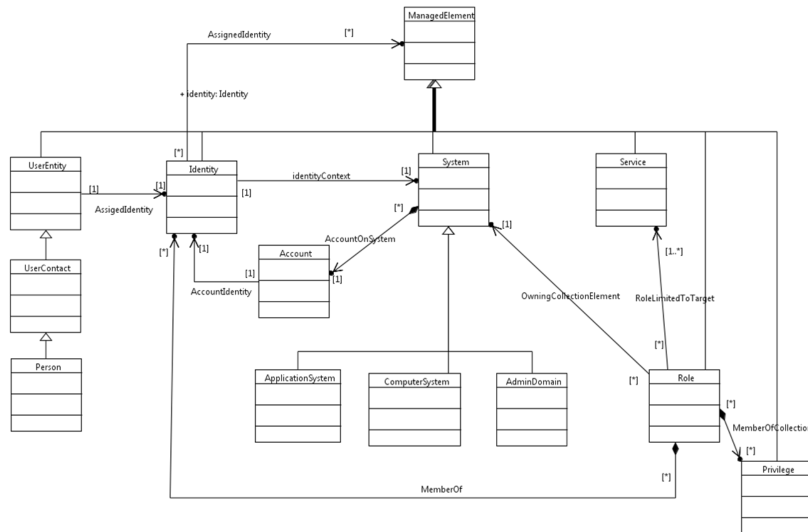


Figure A.4: CIM access control profile, security principals

change in expressive power.

In Figure A.5 the structure of authentication rules is described. This structure presents a clear correspondence with the structure of the authentication rules of the metamodel.

The structures in Figure A.6 describe three different alternatives that can be used to describe in the CIM model classical authorizations. The presence of three different alternatives derives from the expressivity of CIM and its redundancy. Several entities and a variety of structures can be used, depending on the specific solution, but in all cases the structure is consistent with the expressivity of the metamodel. The three solutions depicted in the Figure adopt distinct approaches, but it is interesting to observe that all the representations are compatible with the metamodel and expressible in a more natural and conceptual way in it. The specific advantage offered by the metamodel is the absence of this redundancy, which leads to a clearer approach for the definition of the security policy of a scenario. In turn, this leads to better and more effective support by the tools, which otherwise would have to cover a larger number of varieties with less precision. The correspondence is even more direct considering that the CIM model described here does not offer the representation of the conditions that are part of the PCIM model. The conditions of the PCIM model show an expressivity greater than the one available in the metamodel, but this limitation is justified by the same considerations that have been expressed when evaluating the XACML language, i.e., the need in the metamodel to offer the possibility to translate with (semi-)automatic tools the IT level policy toward the abstract configuration, with the need to avoid the use of a computationally complete language in the representation of the IT policy.

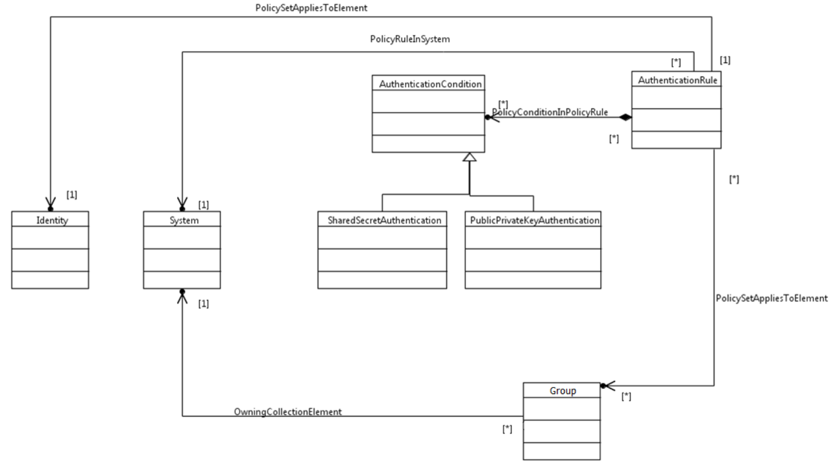


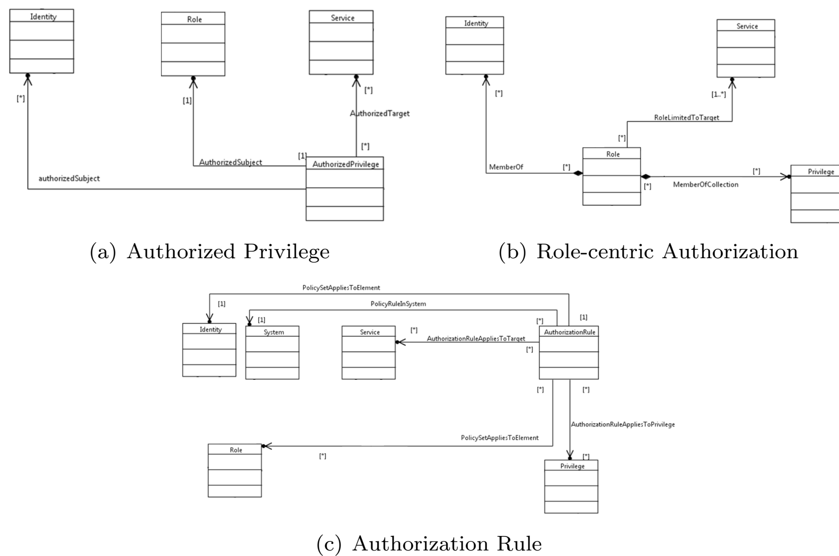
Figure A.5: CIM access control profile, authentication

A.3 KAoS

The last comparison is with the KAoS proposal. The main features of this model have been analyzed in Work Document WD2.2. We can consider the graphical representation of the model that characterizes KAoS, presented in Figure A.7. Comparing the KAoS model with the metamodel, it clearly appears that some of the features of the KAoS model are not supported in the metamodel. These are features that are of interest for the design of privacy management solutions and for some advanced applications of security, but do not have today a significant impact in the scenarios that metamodel is going to focus on. For instance, obligations are a first class citizen of the KAoS model, but they are not part of the metamodel. The motivation is that the metamodel considers Future Internet scenarios in an industrial setting, and today these systems do not support those functions. Apart from the support of obligations, the KAoS schema and the metamodel schema present strong similarities.

A.4 Summary of the analysis

The analysis of existing policy languages has produced at the start of the design valuable information that was taken into account in the design of the metamodel. The analysis of the models designed for other policy languages shows that all the interesting features that are significant for the design of a modern policy language are present in the proposal discussed in Section 2.2.2. There are some features that are limited compared to other proposals, like the support for flexible predicates and the representation and management of obligations, but the motivations presented justify their absence. On the other hand, on



(a) Authorized Privilege

(b) Role-centric Authorization

(c) Authorization Rule

Figure A.6: CIM access control profile, authorization

several measures the metamodel shows to be a solution more adequate for the requirements of the project and in general for the conceptual representation of policies in modern information systems. The metamodel offers a greater scope, supports the integrated representation of security requirements on applications and network connections, gives flexibility in the representation of authorization and authentication rules, and has a native integration with the services of Semantic Web tools. All these reasons justify the development of a novel policy language.

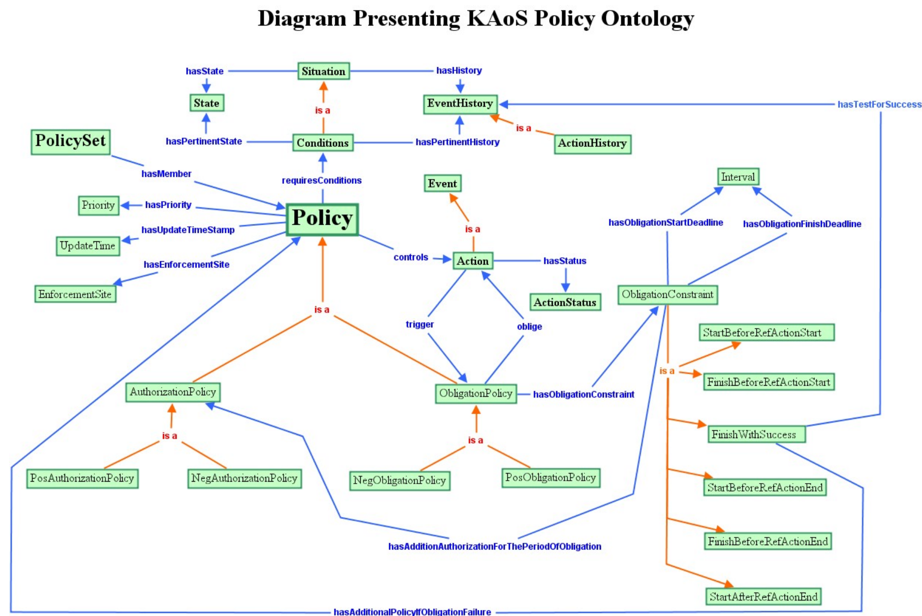


Figure A.7: KAoS Policy model [115]

B

Model specialization example

B.1 Model specialization for Operating Systems

This section describes the PoSecCo metamodel specialization for Operating Systems (OS) which extends the functional system model from Sections 2.1.1 to 2.1.3 adhering to the constraints outlined in Section 2.3.

Operating Systems are defined in Common Information Model (CIM) [31] as

> An **OperatingSystem** is software/firmware that makes a ComputerSystem's hardware usable, and implements and/or manages the resources, file systems, processes, user interfaces, services ... available on the ComputerSystem.

For the PoSecCo Functional System Model, we want to focus on the Operating System that is *currently executed* on a system (*Node*). At most one Operating System can execute at any time on a system (that is a physical node or a physical machine), but it is possible that not always the same OS is running on a given system, e.g. dual or multi boot systems.

The class *OperatingSystem* is a subclass of class *ITResource*, or more specific of *ApplicationContainer*. As such, it has the relationship *runs_on* to the class *Node* that describes the operation system that is currently executed on this node. At any time, at most one OS is executed on a node. The operating system provides the basis to run all other kinds of *ITResource*s on a system and is therefore the lowest component of the *ITResource* hierarchy which is needed to provide a given *ITService*.

The class *OperatingSystem* is subclassed by the different groups of operating

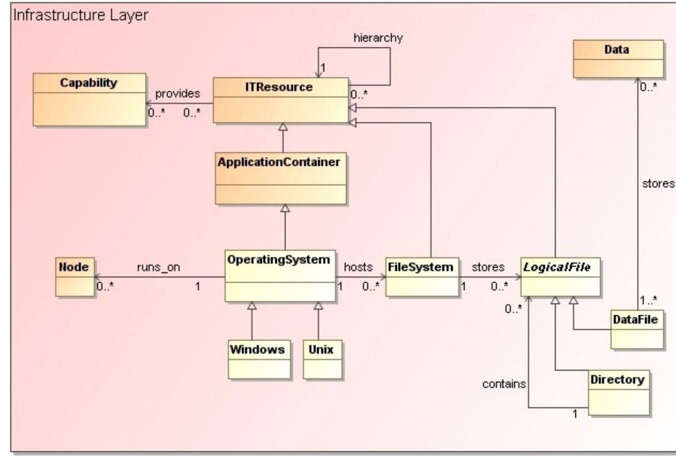


Figure B.1: Operating System Meta Model

systems, like *Windows* and *Unix*. More specializations are possible, e.g. to distinguish the different Unix dialects or to model other types of OS.

An instance of *OperatingSystem* can be further characterized indicating:

- its *CPEName*, a unique software identifier inheritated from class *ITResource* subsuming information about the vendor, product name, release and patch level;
- its OS type (an integer indicating the type of operating system as proposed in the CIM class *CIM_OperatingSystem*).

We can assign the following capabilities (subclasses of class *Capability*) to the *OperatingSystem*:

- *Authentication*;
- *Authorization*;
- *DataProtection*;
- *Auditing*.

B.2 Model specialization for Database

This section describes the PoSecCo metamodel specialization for database management systems (DBMS), with a particular focus on relational databases based on SQL, standardized by ISO/IEC 9075. The specialization shall permit the generation of security-related configurations for DBMS, in particular access control rules based on the *GRANT* statement offered by SQL's Data Control Language (DCL). The required subclasses of the core meta-model are depicted in Figure B.2, and will be explained in the remainder of the section.

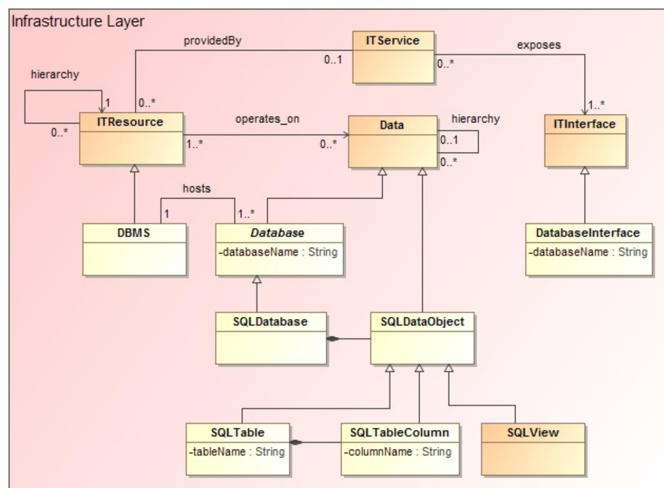


Figure B.2: Database Meta-Model

B.2.1 DatabaseInterface

The class *DatabaseInterface* extends the core meta-model class *ITInterface*. It describes a remote interface to a DBMS, the actual database is hereby identified by the attribute *databaseName*, which is the unique identifier of a database within a given DBMS. Please note that *DatabaseInterface* has not been defined as subclass of *WebInterface*, since database connection strings (well-known from, for instance, JDBC) do not adhere to the RFC-defined format. Examples for database interfaces are JDBC, ODBC, or ADO.NET.

B.2.2 DBMS

The class *DBMS* extends the core meta-model class *ITResource*, and can be used to represent any kind of DBMS, regardless of whether it is based on a relational, object-oriented or object-relational model. The underlying model is determined by the kind of *Database* linked to the DBMS via the association *hosts*. This association refines the generic association *operates-on*, and can be used to link one or multiple databases (with, e.g., individual schemas) to a DBMS.

B.2.3 Database and SQLDatabase

The abstract class *Database* extends *Data*, and comprises the attribute *databaseName*, which is a unique identifier of a database within its DBMS. For the time being, only one subclass has been foreseen, *SQLDatabase*, herewith covering the majority of DBMS in production use. In the scope of PoSecCo, a *SQLDatabase* can be seen as a composition of sensitive *SQLDataObject*s, herewith refining the generic self-reference *hierarchy* defined for *Data*.

B.2.4 SQLDataObject, SQLTable, SQLTableColumn and SQLView

SQLDataObject and its three subclasses *SQLTable*, *SQLTableColumn*, and *SQLView* are subclasses of *Data*, and are referenced by above-introduced class *SQL-Database*. The three non-abstract subclasses *SQLTable*, *SQLTableColumn*, and *SQLView* allow the identification of single tables, columns in a table, or "virtual" tables obtained as result of a SQL query within a given database, information required to both control access to relational data, and to protect the confidentiality or integrity of database content by means of cryptographic functions. The enforcement of respective IT policies does not seem to require, at this point, further details about the complexity of relational data models (e.g., SQL data types used, or information about table keys).

The SQL statement *GRANT* has the following syntax, *object_name* hereby denotes the sensitive information to be protected, e.g., a database table as identified by *SQLTable*:

GRANT privilege_name ON object_name TO user_name — PUBLIC — role_name [WITH GRANT OPTION];

C

Policy Incompatibility

C.1 Policy Incompatibility SWRL Rules

The following SWRL rules can be used to detect modality conflicts and compose them following the "*Most specific wins*" criterion:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1), sign(?a1,?s1),
     on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr2), sign(?a2,?s2),
     containsResource+(?r2,?r1), canActAs(?pr2,?pr1),
     DifferentFrom(?s1,?s2) -> winsVs(?a2,?a1)

on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr), sign(?a1,?s1),
     on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr), sign(?a2,?s2),
     containsResource+(?r2,?r1), DifferentFrom(?s1,?s2) ->
     winsVs(?a2,?a1)

on(?a1,?r), toDo(?a1,?act), grantedTo(?a1,?pr1), sign(?a1,?s1),
     on(?a2,?r), toDo(?a2,?act), grantedTo(?a2,?pr2), sign(?a2,?s2),
     canActAs(?pr2,?pr1), DifferentFrom(?s1,?s2) -> winsVs(?a2,?a1)
```

Listing C.1: SWRL rule that allows to infer links among nodes.

When the conflict involves two policies that have the same specificity level, we apply the "*Denials take precedence*" criterion, with the following rule:

```
SystemAuthorization(?a1), PositiveAuthorization(?a1), on(?a1,?r),
     toDo(?a1,?act), grantedTo(?a1,?pr), SystemAuthorization(?a2),
     NegativeAuthorization(?a2), on(?a2,?r), toDo(?a2,?act),
     grantedTo(?a2,?pr) -> winsVs(?a2,?a1)
```

Listing C.2: SWRL rule that allows to infer links among nodes.

Some conflicts cannot be solved automatically, and, thus, to detect these conflicts, which should be solved directly by the security team, we defined the following SWRL rules:

```
on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1), sign(?a1,?s1),
    on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr2), sign(?a2,?s2),
    containsResource+(?r2,?r1), canActAs(?pr1,?pr2),
    DifferentFrom(?s1,?s2) -> unsolvableConflict(?a1,?a2)

on(?a1,?r1), toDo(?a1,?act), grantedTo(?a1,?pr1), sign(?a1,?s1),
    on(?a2,?r2), toDo(?a2,?act), grantedTo(?a2,?pr2), sign(?a2,?s2),
    containsResource+(?r1,?r2), canActAs(?pr2,?pr1),
    DifferentFrom(?s1,?s2) -> unsolvableConflict(?a1,?a2)
```

Listing C.3: SWRL rule that allows to infer links among nodes.

C.2 Preprocessing Algorithms

The preprocessing algorithm is presented in Algorithm 5. It takes as input the set of roles R, the set of system authorizations SA, the set of role authorizations RA, and the role hierarchy RH. The algorithm, for each system authorization $sysauth$, creates a pair of roles, related to the action a and the resource r associated with $sysauth$. Then, it creates two new system authorizations in order to grant to the just created roles the positive and negative authorizations to do the action a on the resource r. Then, it removes the authorization $sysauth$ and finally replaces it with adequate role authorizations in order to preserve the semantics of the policy. It uses the $preprocessHierarchy$ function, shown in Algorithm 6, which, recursively, goes deeper in the resource hierarchy and creates the adequate roles.

C.3 Permission-Based and Object-Based SoD

A first way to handle *Permission-based* and *Object-based* SoD constraints is by extending the ontology with new axioms. For *Permission-based SoD* we defined a new property *canDo* : *Principal* $\rightarrow$ *Action* that is equivalent to $grantedTo^{-1} \circ toDo$. To keep track of the conflicts, we have defined a new class $PSoDConflict \sqsubseteq Principal$ and in order to identify the actions involved in constraints we defined the property *PermBasedSod* : *Action* $\rightarrow$ *Action*. We express SoD constraints using the following axioms:

$$\forall a_1, a_2 \in Action : PermBasedSod(a_1, a_2)$$
$$PSoDConflict \equiv \exists canDo.\{a_1\} \sqcap \exists canDo.\{a_2\}$$

and to enforce the SoD, we simply have to add to the ontology the axiom $PSoDConflict \sqsubseteq \bot$. In the same way we can define the *Object-based SoD*. In DL formulas we write *canActOn* : *Principal* $\rightarrow$ *Resource*, $canActOn \equiv grantedTo^{-1} \circ on$, *ObjBasedSod* : *Resource* $\rightarrow$ *Resource*, $RSoDConflict \sqsubseteq Principal$. In order to enforce the *Object-based SoD*, we add

Algorithm 5: Preprocessing procedure

```
Input  : R, SA, RA, RH
Output: R, SA, RA, RH
begin
    for sysauth ∈ SA do
        s = sign(sysauth);
        r = createNewRole();
        if r ∉ R then
            R = R ⋃{r};
            SystemAuthorization sa;
            grantedTo(sa) = r;
            on(sa) = on(sysauth);
            toDo(sa) = toDo(sysauth);
            sign(sa) = s;
            SA = SA ⋃{sa};
            if s = + then
                s′ = −;
            else
                s′ = +;
            r′ = createRole(p, s′);
            if r′ ∉ R then
                R = R ⋃{r′};
                SystemAuthorization sa′;
                grantedTo(sa′) = r′;
                on(sa′) = on(sysauth);
                toDo(sa′) = toDo(sysauth);
                sign(sa′) = s′;
                SA = SA ⋃{sa′};
            cannotBe = cannotBe ⋃{(r, r′)} ⋃{(r′, r)};
            preprocessHierarchy(on(sysauth), r, r′, toDo(sysauth), R, SA, RA, RH);
        RoleAuthorization ra;
        enableRole(ra) = r;
        grantedTo(ra) = grantedTo(sysauth);
        grantor(ra) = grantor(sysauth);
        RA = RA ⋃{ra};
        SA = SA − {sysauth};
```

Algorithm 6: Preprocess Hierarchy procedure

```
Input  : res ∈ Resources, r1, r2 ∈ R, s ∈ {+, −},action ∈ Actions,R, SA, RA, RH
Output: R, SA, RA
begin
    for res′ ∈ contains(res) do
        r = createNewRole();
        if r ∉ R then
            R = R ⋃{r};
            SystemAuthorization sa;
            grantedTo(sa) = r;
            on(sa) = res′;
            toDo(sa) = a;
            sign(sa) = s;
            roleHierarchy = roleHierarchy ⋃{(r1, r)};
            SA = SA ⋃{sa};
            if s = + then
                s′ = −;
            else
                s′ = +;
            r′ = createNewRole();
            if r′ ∉ R then
                R = R ⋃{r′};
                SystemAuthorization sa′;
                grantedTo(sa′) = r′;
                on(sa′) = res′;
                toDo(sa′) = a;
                sign(sa′) = s′;
                roleHierarchy = roleHierarchy ⋃{(r2, r′)};
                SA = SA ⋃{sa′};
            cannotBe = cannotBe ⋃{(r, r′)} ⋃{(r′, r)};
            preprocessHierarchy(res′, r, r′, a, R, SA, RA, RH);
        RoleAuthorization ra;
        enableRole(ra) = r;
        grantedTo(ra) = grantedTo(sysauth);
        grantor(ra) = grantor(sysauth);
        RA = RA ⋃{ra};
        SA = SA − {sysauth};
```

to the ontology the following set of axioms:

$$\begin{aligned}&\forall res_1, res_2 \in Resource : ObjBasedSod(res_1, res_2)\\&\quad RSoDConflict \equiv \exists canActOn.\{res_1\} \sqcap \exists canActOn.\{res_2\}\\&RSoDConflict \sqsubseteq \bot\end{aligned}$$

However, this approach has two drawbacks: (1) it adds complexity to the ontology, (2) it represents all these SoD constraints in a redundant way, because the only difference between all of them is the hierarchy on which the constraint is applied. A second approach is to implement *Permission-based* and *Object-based SoD* constraints using an ad-hoc role hierarchy and, thus, representing them as *Role-based SoD*. Thus, we can define, for each pair composed by an action $a \in Actions$ and a resource $res \in Resource$, a new role $r_{a,res}$ that represents a high level role. Each role $r_{a,res} \in Role$ has assigned to it only a $SystemAuthorization$, which allows it to do the action a on the resource res. Then, we define for each action $a \in Actions$ a role r_a, and for each resource $res \in Resources$ a role r_{res}. These roles are low level roles; they do not have any authorization assigned to them, but they are used to express the SoD constraints. In order to create the adequate relationships between high level and low level roles, we add the following axioms to the ontology:

$$\begin{aligned}&\forall res \in Resource, \forall a \in Actions :\\&\quad roleHierarchy(r_{a,res}, r_{res})\\&\quad roleHierarchy(r_{a,res}, r_a)\end{aligned}$$

These axioms create a low level of abstract roles, parameterized only on the action or on the resource, that are sub-roles of high level roles. This second level aggregates the roles on a resource and action basis. Now, we can simply express *Permission-based SoD* and *Object-based SoD* using the axioms presented for *Role-Based SoD*, but considering only low level roles, respectively the ones associated with actions or with resources. For instance, in order to express SoD between two resources res_1 and res_2, we can create a negative role authorization r_{auth} such that $grantedTo(r_{auth}, r_{res_1})$ and $enabledRole(r_{auth}, r_{res_2})$, whereas to express the SoD between two actions a_1 and a_2 we can create a negative role authorization r_{auth} such that $grantedTo(r_{auth}, r_{a_1})$ and $enabledRole(r_{auth}, r_{a_2})$.

C.4 Optimization

The reasoning phase can be made more efficient if we adequately prune the model, removing from it all authorizations, identities and roles that cannot be involved in any SoD conflict. The pruning procedure is presented in Algorithm 7. It takes as input the sets of identities, roles and role authorizations and prunes the individuals that cannot be involved in any SoD constraint. The function $contained+(id)$ returns the set of groups in which the id is contained,

directly or indirectly.

Algorithm 7: Pruning procedure

```
Input  : R, RA, ID
Output : R, RA, ID
begin
    for id ∈ ID do
        if ∄ra ∈ RA : grantedTo(ra) = id then
            ID = ID − {id};
            grantedTo = grantedTo − {(x, y) : y = id};

    for r ∈ R do
        if ∄(x, t) ∈ roleHierarchy : y = r then
            if ∄ra ∈ RA : enabledRole(ra) = r then
                R = R − {r};
                RA = RA − {auth ∈ RA : enabledRole(auth) = r};

    for auth ∈ RA do
        r = enabledRole(auth);
        id = grantedTo(auth);
        containedIn = {i ∈ ID : contained + (id) = i};
        if ∄id2 ∈ ID : ((id2 ∈ containedIn ∨ id ≠ id2) ∧ ∃ra ∈ RA : grantedTo(ra) =
        id2 ∧ enabledRole(ra) ≠ r) then
            ID = ID − {id};
            RA = RA − {auth};
```

The pruning removes the identities without any authorization, the roles that are not assigned in any role authorization, and the redundant role authorizations. The pruning phase can substantially improve the performance of the reasoning phase, and, thus, adding a pruning phase before the reasoning leads to lower global analysis time and better performance.

D
Technology

Here we will show an overview of the technologies used in the develoment of the IT Policy Tool, implemented as an Eclipse plug-in. The tool supports the import/export of the IT Policy in several formats, such as: XMI [90] and XML [17]. Furthermore the current version of the tool supports the generation of an OWL representation of the IT Policy and permits a quick invocation of the reasoner over the produced ontology. The plug-in also offers, through the use of Protégé [111], functionalities to manipulate and edit the OWL representation in order to enrich the IT Policy ontology.

D.1 Eclipse

Eclipse [43] is the most significant open source application development framework, offering a high degree of flexibility and supporting the extension of its functionality through the implementation of plugins. The Eclipse platform encourages the reuse of the functions of other plugins and modules of the Eclipse framework, speeding up and improving the quality of the development process. The choice of an Eclipse plugin to support the creation of the IT Policies is due to the fact that the services that Eclipse offers to support the construction of a large software system can be immediately adapted to the design of a complex IT Policy. In both scenarios, there is the need to offer rich editing and exploration support over information that describes a multiplicity of instances that follow a prescribed syntax and have a number of relationships among them. Eclipse offers a common plugin interface to facilitate the integration among different plugins (see Figure D.1).

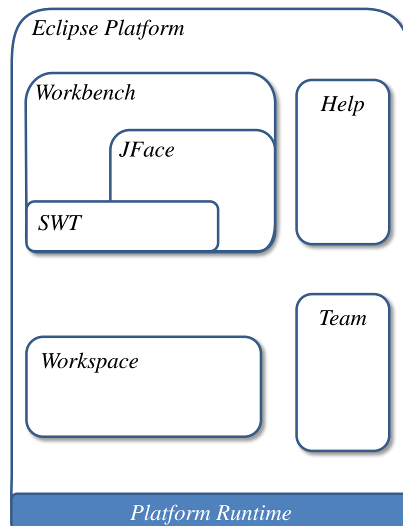


Figure D.1: Eclipse plugin interface.

The components of the Eclipse Platform are: a *Platform Runtime*; a low level widget toolkit called *Standard Widget Toolkit* (SWT); a UI toolkit called *JFace* that provides helper classes for developing UI features with the SWT faster; a *Workbench* implementation that provides the look and feel and therefore the UI personality of the Eclipse Platform; a *Workspace* that consists of one or more top-level projects, where each project maps to a corresponding user-specified directory in the file system; a *Team* support that allows a project in the workspace to be placed under version and configuration management with an associated team repository; a *Help* mechanism that allows tools to define and contribute documentation to one or more online books.

D.2 Web Ontology Language

The *Web Ontology Language* (OWL) [79] is a formal language for representing ontologies in the Semantic Web. Ontologies can be defined as the information about categories of objects and how they are interrelated. OWL can not only represent objects, but also information about these objects (RDF annotation properties). OWL, which today is a World Wide Web Consortium (W3C) standard, was developed by the W3C Web Ontology Working Group. The language is influenced by representation languages such as XML and RDF, OIL and DAML+OIL, as well as by Description Logics and frames. OWL supports the specification and use of ontologies that consist of terms representing individuals, classes of individuals, properties, and axioms that assert constraints over them.

The use of OWL to define policies has several very important advantages

that become critical in distributed environments involving coordination across multiple organizations. First, most policy languages define con-
straints over classes of targets, objects, actions and other constraints (e.g., location). A substantial part of the development of a policy is often devoted to the precise specification of these classes. This is especially important if the policy is shared between multiple organizations that must adhere to or enforce the policy even though they have their own native schemas or data models for the domain in question. The second advantage is that OWL's grounding in logic facilitates the translation of policies expressed in OWL to other formalisms, either for analysis or for execution.

The IT Policy Tool, for the reasons above, provides functionaliies to manage ontologies. It increases the expressive power of the model and supports sophisticated correctness verification. This integration is possible through the use of the OWL API library [52], which offers a complete set of interface to manage the ontologies. OWL API is based on the OWL DL language. Furthermore, thanks to the great extensibility of the OWL API we can add modules to provide specific services, for instance SPARQL-DL [108] to query the ontology and SWRL [54] to define semantic rules on the ontology.

D.2.1 OWL API

The integration with the tools managing the ontological aspects is implemented within the plug-in through the latest version of OWL API [52]. OWL API has been designed to meet the needs of programmers developing OWL based applications, OWL editors and OWL reasoners. It is a high level API that is closely aligned with the OWL 2 specification. It includes first class change support, general purpose reasoner interfaces, validators for the various OWL profiles, and support for parsing and serializing ontologies in a variety of syntaxes. The API also has a flexible design that allows third parties to provide alternative implementations for all the major components.

The current version of OWL API provides a suite of interfaces along with a reference implementation that facilitates the use of OWL in a wide variety of applications. At its core, the OWL API consists of a set of interfaces for inspecting, manipulating and reasoning with OWL ontologies. The OWL API supports loading and saving ontologies in a variety of syntaxes. However, none of the model interfaces in the API reflect, or are biased to any particular concrete syntax or model. For example, unlike other APIs such as Jena [56], or the Protégé 3.X API, the representation of class expressions and axioms is not at the level of RDF triples. Indeed, the design of the OWL API is directly based on the OWL 2 Structural Specification [84].

D.2.2 Reasoner

A key benefit of the OWL representation is the ability to support reasoning. Reasoners are used to check the consistency of ontologies, check to see whether the signature of an ontology contains unsatisfiable classes, compute class and

property hierarchies, and check to see if axioms are entailed by an ontology. The OWL API has various interfaces to support the interaction with OWL reasoners. A reasoner is a key component for working with OWL ontologies. In fact, virtually all querying of an OWL ontology (and its imports closure) should be done using a reasoner. This is because knowledge in an ontology might not be explicit and a reasoner is required to deduce implicit knowledge so that the correct query results are obtained. At the time of this writing, FaCT++ [113], HermiT [58], Pellet [106], and Racer Pro [47] reasoners provide OWL API wrappers. In the PoSecCo IT Policy Tool we use HermiT as reasoner. HermiT is a publicly-available OWL reasoner based on a "hypertableau" calculus, which provides more efficient reasoning than many other algorithms. This performance gain is important for PoSecCo, which may have to manage policies containing a large number of elements.

D.2.3 Semantic Web Rule Language

Semantic Web Rule Language (SWRL) [55] was designed to be the rule language of the Semantic Web. SWRL is based on a combination of the OWL DL and OWL Lite sublanguages of the *Web Ontology Language* (OWL) the Unary/Binary Datalog sublanguages of the *Rule Markup Language*. SWRL allows users to write Hornlike rules expressed in terms of OWL concepts to reason about OWL individuals. The rules can be used to infer new knowledge from the existing OWL knowledge bases. The SWRL Specification does not impose restrictions on how reasoning should be performed with SWRL rules. In this way, SWRL provides a convenient starting point for integrating rule systems to work with the Semantic Web.

SWRL rules are written as antecedent / consequent pairs. In SWRL terminology, the antecedent is referred to as the rule body and the consequent is referred to as the head. The head and body consist of a conjunction of one or more atoms. The intended meaning is consistent with the classical semantics that characterizes most rule paradigms: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. Thanks to the introduction of these rules, the verification process of the model can be automated. The use of SWRL offers interesting opportunities in the PoSecCo context and is well supported by the reasoners.

D.2.4 Simple Protocol And RDF Query Language - Description Logic

Another common extension to DL engines is the ability to execute queries against the semantic model. *Simple Protocol And RDF Query Language - Description Logic* (SPARQL-DL) [108] is a graph-matching query language. Given a data source D, a query consists of a pattern which is matched against D, and the values obtained from this matching are processed to give the answer. The data source D to be queried can be composed of multiple sources. SPARQL-DL is a distinct subset of the most famous *Simple Protocol And RDF Query*

Language (SPARQL) [97].

SPARQL-DL supports two different types of queries: ASK and SELECT. An ASK-query returns a Boolean result whereas a SELECT-query returns all possible bindings of the provided variables. As within SPARQL, the *DISTINCT* keyword removes automatically all redundant bindings within the result set.

E
Refinement Modules

As for the policy enrichment, the policy refinement process is executed by orchestrating a set of refinement modules. Each refinement module can identify the set of model elements readily available for refinement and produce a new description fragment that describes abstract configurations. In order to manage, in a flexible way, the refinement process we have decided to adopt the same approach of policy enrichment, thus we have implemented the refinement process with the use of an *extension point*.

```
<element name="module">
     <complexType>
        <attribute name="systemType" use="required">
           <simpleType>
              <restriction base="string">
                 <enumeration value="DBMS"></enumeration>
                 <enumeration value="OS"></enumeration>
                 . . .
              </restriction>
           </simpleType>
        </attribute>
        <attribute name="elementType" use="required">
           <simpleType>
              <restriction base="string">
                 <enumeration value="ITSystemAuthorization"></enumeration>
                 <enumeration value="ITRoleAuthorization"></enumeration>
                 <enumeration value="ITAuthenticationRule"></enumeration>
                 . . .
              </restriction>
           </simpleType>
        </attribute>
        <attribute name="class" type="string" use="required">
           <annotation>
```

```
          <appinfo>
            <meta.attribute kind="java" basedOn=":eu.posecco.refinement.
                                                 core.IRefinementModule"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="name" type="string" use="required" />
    </complexType>
  </element>
```

Listing E.1: Definition of Refinement Extension Point

The extension point, whose definition is presented in Listing E.1, has four attributes:

1. *name* represents the name of the refinement module,
2. *class* represents the class that implements the refinement module for the specific IT element,
3. *elementType* represents the type of the IT element (e.g., ITSystemAuthorization, ITRoleAuthorization),
4. *systemType* represents the type of the concrete system involved in the policy (e.g., DBMS, OS)

Figure E.1 shows an example of the implementation of a refinement module for DBMS (systemType attribute) involved in *ITSystemAuthorizations* (elementType attribute).

Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

- eu.posecco.itpolicytool.ITRefinement
- eu.posecco.refinement.core.RefinementModule
  - DBMS System Authorization (module)
  - Tomcat System Authorization (module)
  - OS System Authorization (module)
  - OS Role Authorization (module)
  - DBMS Role Authorization (module)
  - DBMS Authentication (module)
  - OS Authentication (module)
  - OS Protected Data (module)
  - DBMS Protected Data (module)
  - OS Encrypted Data (module)
  - DBMS Encrypted Data (module)
  - DBMS Signed Data (module)
  - OS Signed Data (module)

Add... Remove Up Down

Extension Element Details

Set the properties of "module". Required fields are denoted by "*".

systemType*: DBMS

elementType*: ITSystemAuthorization

class*: eu.posecco.refinement.core.modules.systemauthorizations.DbmsSystemAuthorizationRefinementModul Browse...

name*: DBMS System Authorization

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

Figure E.1: Definition of a Refinement extension.

Enrichment Modules

The *Policy Enrichment* plug-in implements several enrichment modules which are executed during the enrichment process. They are used to introduce additional information as described in Section 5. In general, the enrichment process is led by the CPE name attribute of the class *ITResource* related to the IT element. The *Policy Enrichment* defines an extension point, called *Enrichment*, that allows other plugins to contribute with new enrichment modules.

```
<element name="module">
      <complexType>
         <attribute name="name" type="string" />
         <attribute name="CPEName" type="string" use="required">
            <annotation>
               <documentation>
                  The CPE of the element enriched by the module.
               </documentation>
            </annotation>
         </attribute>
         <attribute name="ontologyIRI" type="string" use="required">
            <annotation>
               <documentation>
                  The IRI of the ontology used in the EM.
               </documentation>
            </annotation>
         </attribute>
         <attribute name="wizard" type="string">
            <annotation>
               <documentation>
                  The dedicated wizard for the EM.
               </documentation>
               <appinfo>
                  <meta.attribute kind="java"
                  basedOn="eu.posecco.enrichment.core.wizard.EnrichmentModuleWizard:"/>
```

```
            </appinfo>
          </annotation>
        </attribute>
      </complexType>
    </element>
```

Listing F.1: Definition of Enrichment Extension Point

The extension point, whose definition is presented in Listing F.1, has four attributes:

1. *name* represents the name of the enrichment module,
2. *CPEName* represents the CPE name of the element which has to be enriched,
3. *ontologyIRI* represents the enrichment ontology IRI,
4. *wizard* represents the class that implements a dedicated wizard in order to help the user, in the addition to concepts belong to the enrichment ontology.

As described before, the plug-in that wants to contribute with new functionalities has to define an extension satisfying the extension point. Figure F.1 shows an example of the implementation of the extension point.

Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

- eu.posecco.itpolicytool.ITEnrichment
- eu.posecco.enrichment.core.EnrichmentModule
  - DBMS MySQL 5.5 Authz (module)
  - DBMS MySQL 5.4 Authz (module)

Add... | Remove | Up | Down

Extension Element Details

Set the properties of "module". Required fields are denoted by "*".

CPE*: cpe:/a:oracle:mysql:5.5.22

ontologyIRI*: http://www.posecco.eu/ontologies/enrichment/DBMS_MySQL5.5_authz.owl

wizard: eu.posecco.enrichment.core.internal.wizard.mysql55.authz.DbmsMySql55AuthzWizard Browse...

name: DBMS MySQL 5.5 Authz

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

Figure F.1: Definition of an Enrichment extension.

F.1 NeOn ToolKit

In order to aid the user in the import of the enrichment ontology, the *Policy Enrichment* provides the functionality to use a custom wizard for each enrichment ontology. Furthermore, the tool provides the functionality to edit manually the enriched ontology through the use of the *NeOn ToolKit*. It is the ontology engineering environment originally developed as part of the NeOn project [1] from FP6, it is composed of a set of Eclipse plugins to support, among other interesting things, ontology-guided model description.

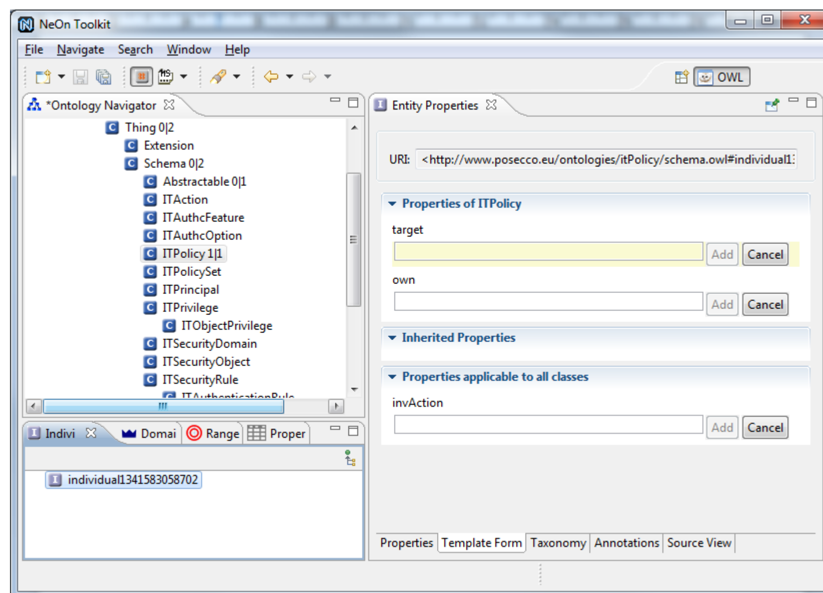


Figure F.2: NeOn toolkit used to model IT policies

Figure F.2 shows a fragment of IT Policy edited with the NeOn toolkit. NeOn toolkit offers to the user forms for the querying and insertion of elements in the enrichment ontology that depend on an automatic analysis of the structure of the ontology itself, without the need to build an ad-hoc software module for each enrichment option.

[1] http://www.neon-project.org

Bibliography

[1] E.S. Al-Shaer and H.H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *Proc. of IEEE/IFIP Integrated Management*, 2003.

[2] E.S. Al-Shaer and H.H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. of IEEE Infocom*, 2004.

[3] E.S. Al-Shaer and H.H. Hamed. Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.

[4] AnTuTu labs. AnTuTu Benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark` .

[5] A. Armando, R. Carbone, and S. Ranise. Automated analysis of semantic-aware access control policies: a logic-based approach. In *Proceedings of the Fifth IEEE International Conference on Semantic Computing (ICSC)*. IEEE, 2011.

[6] Alessandro Armando and Silvio Ranise. Automated symbolic analysis of arbac-policies. In Jorge Cuellar, Javier Lopez, Gilles Barthe, and Alexander Pretschner, editors, *Security and Trust Management*, volume 6710 of *Lecture Notes in Computer Science*, pages 17–34. Springer Berlin / Heidelberg, 2011.

[7] Mark Balanza, Kervin Alintanahin, Oscar Abendan, Julius Dizon, and Bernadette Caraig. Droiddreamlight lurks behind legitimate android apps. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 73–78. IEEE, 2011.

[8] C. Basile, A. Cappadonia, and A. Lioy. Network-level access control policy analysis and transformation. *Networking, IEEE/ACM Transactions on*, (99), 2011.

[9] David Basin, Samuel J. Burri, and Günter Karjoth. Separation of duties as a service. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 423–429, New York, NY, USA, 2011. ACM.

[10] David A. Basin, Samuel J. Burri, and Günter Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *CSF*, pages 99–113, 2011.

[11] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. *In IEEE Computer Security Foundations Workshop*, 2004.

[12] Moritz Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In Joachim Biskup and Javier López, editors, *Computer Security ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 203–218. Springer Berlin / Heidelberg, 2007.

[13] T. Berners-Lee, D. Connolly, and S. Hawke. Semantic web tutorial using n3. In *Twelfth International World Wide Web Conference*, 2003.

[14] Annett Laube BFH, Guillaume Gagnerot BFH, Henrik Plate SAP, Gerhard Hassenstein BFH, Matteo Casalino SAP, Stefano Paraboschi, Cataldo Basile, and Theodoor Scholte. D3. 3-configuration meta-model. 2012.

[15] Bob Moore and Ed Ellesson and John Strassner and Andrea Westerinen. Policy core information model. Internet Engineering Task Force RFC 3060, February 2001.

[16] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM TISSEC*, 5(1), 2002.

[17] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[18] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012.

[19] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices.* ACM, 2011.

[20] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium.* USENIX, 2013.

[21] T. Cadenhead, M. Kantarcioglu, and B. Thuraisingham. Scalable and efficient reasoning for enforcing role-based access control. *Data and Applications Security and Privacy XXIV*, pages 209–224, 2010.

[22] Huoping Chen, Youssif B Al-Nashif, Guangzhi Qu, and Salim Hariri. Self-configuration of network security. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 97–97. IEEE, 2007.

[23] L. Cholvy and F. Cuppens. Analyzing consistency of security policies. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 103–112. IEEE, 1997.

[24] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. A logic programming approach to conflict resolution in policy management. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000)*, pages 121–132. Morgan Kaufman, 2000.

[25] Lorenzo Cirio, Isabel F. Cruz, and Roberto Tamassia. A role and attribute based access control system using semantic web technologies. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'07, pages 1256–1266, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, and Arosha Bandara. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 239–250, New York, NY, USA, 2009. ACM.

[27] Xi Deng, Volker Haarslev, and Nematollaah Shiri. Resolution based explanations for reasoning in the description logic alc. In *IN: PROCEEDINGS OF THE CANADIAN SEMANTIC WEB WORKING SYMPOSIUM*, pages 55–61. Springer, 2006.

[28] J. DeTreville. Binder: a logic-based security language. *In IEEE Symposium on Security and Privacy*, pages 95–103, 2002.

[29] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[30] Distributed Management Task Force (DMTF). Cim policy model white paper. DMTF Technical Report DSP0108, June 2003.

[31] DMTF Distributed Management Task Force. Common Information Model (CIM). Online: `http://dmtf.org/standards/cim`.

[32] Dolphin Browser. Dolphin Browser for Android. `https://play.google.com/store/apps/details?id=mobi.mgeek.TunnyBrowser` .

[33] Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer Berlin / Heidelberg, 2006.

[34] Daniel Dougherty, Claude Kirchner, Hélène Kirchner, and Anderson Santana de Oliveira. Modular access control via strategic rewriting. In Joachim Biskup and Javier López, editors, *Computer Security ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer Berlin / Heidelberg, 2007.

[35] PCI DSS. Pci security standards council, payment card industry (pci) data security standard. *URL: http://www. pcisecuritystandards. org.*

[36] Olivier Bettan (ed). WD1.8 – Development Environment and test-bed for coordinating prototyping. Working document, PoSecCo, 2012.

[37] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, 2010.

[38] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[39] R. Ferrini and E. Bertino. Supporting rbac with xacml+ owl. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 145–154. ACM, 2009.

[40] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham. R owl bac: representing role based access control in owl. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 73–82. ACM, 2008.

[41] Simon N Foley and William M Fitzgerald. Management of security policy configuration using a semantic threat graph approach. *Journal of Computer Security*, 19(3):567–605, 2011.

[42] Center for Strategic and International Studies. Securing cyberspace for the 44th presidency, December 2008.

[43] Eclipse Foundation. Eclipse classic. `http://www.eclipse.org/`.

[44] Steve Graham, Anish Karmarkar, Jeff Mischkinsky, Ian Robinson, and Igor Sedukhin. Web services resource 1.2 (ws-resource). *OASIS, April*, 2006.

[45] M. Guarnieri, E. Magri, and S. Mutti. Automated management and analysis of security policies using Eclipse. In *Proc. of the Eclipse-IT 2012*.

[46] Marco Guarnieri, Mario Arrigoni Neri, Eros Magri, and Simone Mutti. On the notion of redundancy in access control policies. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 161–172. ACM, 2013.

[47] Volker Haarslev and Ralf Möller. Racer: A core inference engine for the Semantic Web. In York Sure and Oscar Corcho, editors, *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), Sanibel Island, Florida, USA, Oct 20*, volume 87 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[48] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4):21:1–21:41, July 2008.

[49] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. In *Linux Conference*, volume 3, 2004.

[50] N. Heilili, Y. Chen, C. Zhao, Z. Luo, and Z. Lin. An owl-based approach for rbac with negative authorization. *Knowledge Science, Engineering and Management*, pages 164–175, 2006.

[51] M. Horridge and S. Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

[52] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[53] Ian Horrocks and Peter F Patel-Schneider. Reducing owl entailment to description logic satisfiability. In *The Semantic Web-ISWC 2003*, pages 17–29. Springer, 2003.

[54] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and ruleML. W3C member submission, World Wide Web Consortium, http://www.w3.org/Submission/SWRL/, 2004.

[55] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. Swrl: A semantic web rule language combining owl and ruleml. W3c member submission, World Wide Web Consortium, 2004.

[56] hp. Jena - A Semantic Web Framework for Java. available: http://jena.sourceforge.net/index.html, 2002.

[57] H. Hu, G. Ahn, and K. Kulkarni. Anomaly discovery and resolution in web access control policies. In *Proc. of SACMAT '11*. ACM.

[58] University of Oxford Information Systems Group. HermiT reasoner. `http://www.hermit-reasoner.com/`.

[59] Klaus Julisch. D2. 1 a framework for business level policies. 2011.

[60] L. Kagal, T. Berners-Lee, D. Connolly, and D. Weitzner. Using semantic web technologies for policy management on the web. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 1337. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[61] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of owl dl entailments. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 267–280. Springer Berlin Heidelberg, 2007.

[62] Hiroaki Kamoda and Krysia Broda. Policy conflict analysis using free variable tableaux for access control in web services environments. In *In Policy Management for the Web Workshop*, pages 5–12, 2005.

[63] Hiroaki Kamoda, Masaki Yamaoka, Shigeyuki Matsuda, Krysia Broda, and Morris Sloman. Access control policy analysis using free variable tableaux. *Information and Media Technologies*, 1(2):1155–1169, 2006.

[64] Günter Karjoth. D5.1 - Decision Types and Frameworks. PoSecCo Deliverable, PoSecCo Consortium, 2012.

[65] Martin Hans Knahl. A conceptual framework for the integration of it infrastructure management, it service management and it governance. *Proceedings of the world academy of science, engineering and technology*, 40, 2009.

[66] V. Kolovski. Logic-based access control policy specification and management.

[67] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 677–686. ACM, 2007.

[68] Vladimir Kolovski and Bijan Parsia. Ws-policy and beyond: Application of owl defaults to web service policies, 2005.

[69] Vladimir Kolovski, Bijan Parsia, Yarden Katz, and James Hendler. Representing web service policies in owl-dl. In Yolanda Gil, Enrico Motta, V. Benjamins, and Mark Musen, editors, *The Semantic Web ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 461–475. Springer Berlin / Heidelberg, 2005.

[70] James R Langevin, Michael T McCaul, Scott Charney, and Harry Raduege. Securing cyberspace for the 44th presidency. Technical report, DTIC Document, 2008.

[71] Jay Lepreau, Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, and David Andersen. The flask security architecture: System support for diverse security policies, 2006.

[72] Ninghui Li and John Mitchell. Datalog with constraints: A foundation for trust management languages. In Veronica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin / Heidelberg, 2003.

[73] Ninghui Li and Qihua Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *J. ACM*, 55:12:1–12:46, August 2008.

[74] A. Liu and M. Gouda. Complete redundancy detection in firewalls. In *Data and Applications Security XIX*, volume 3654. 2005.

[75] Adrian Ludwig. Android - practical security from the ground up, October 2013. `http://goo.gl/z0RIwu` .

[76] E.C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852–869, 1999.

[77] Fabio Massacci. Reasoning about security: A logic and a decision method for role-based access control. In *Proceedings of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning*, pages 421–435, London, UK, 1997. Springer-Verlag.

[78] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, NJ, USA, 2006.

[79] Deborah L. Mcguinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004.

[80] D.L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.

[81] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo. Mining roles with multiple objectives. *ACM TISSEC 2010*.

[82] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. Mining roles with multiple objectives. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):36, 2010.

[83] T. Moses et al. Extensible access control markup language (xacml) version 2.0. *Oasis Standard*, 200502, 2005.

[84] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Mike Smith. OWL 2 web ontology language: Structural specification and functional-style syntax. Last call working draft, W3C, 2008.

[85] Collin Mulliner, William Robertson, and Engin Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '14, pages 459–470, New York, NY, USA, 2014. ACM.

[86] S. Mutti, M. Arrigoni Neri, and S. Paraboschi. An Eclipse plug-in for specifying security policies in modern information systems. In *Proc. of the Eclipse-IT 2011*.

[87] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[88] Mario Arrigoni Neri, Marco Guarnieri, Eros Magri, Simone Mutti, and Stefano Paraboschi. Conflict detection in security policies using semantic web technology. In *Satellite Telecommunications (ESTEL), 2012 IEEE First AESS European Conference on*, pages 1–6. IEEE, 2012.

[89] Oasis. *XACML Profile for Role-Based Access Control (RBAC)*, 2004.

[90] OMG. *XML Metadata Interchange*, 2011.

[91] David Oppenheimer. The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, 2003.

[92] Stefano Paraboschi, Mario Arrigoni Neri, Simone Mutti, Marco Guarnieri, Eros Magri, Marie-Noelle Lepareaux, Beatriz Gallego-Nicasio Crespo, and Aldo Basile. D2. 4–policy harmonization and reasoning. 2012.

[93] Stefano Paraboschi, Mario Arrigoni Neri, Simone Mutti, Giuseppe Psaila, Paolo Salvaneschi, Mario Verdicchio, and Aldo Basile. D2. 5–it policy meta-model and language (final). 2013.

[94] David A Patterson et al. A simple way to estimate the cost of downtime. In *LISA*, volume 2, pages 185–188, 2002.

[95] Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 130–145. Springer Berlin / Heidelberg, 2006.

[96] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[97] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, January 2008.

[98] 2009 Data Breach Investigations Report. Data breach investigations report 2009.

[99] UK Security Breach Investigations Report. Uk security breach investigations report 2010.

[100] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, pages 137–196. Springer, 2001.

[101] R.S. Sandhu. Role-based access control. *Advances in computers*, 46:237–286, 1998.

[102] Theodoor Scholte. D3. 5-models to refine the it policy at service level. *PoSecCo Deliverable, PoSecCo Consortium*, 2012.

[103] Yuru Shao, Xiapu Luo, and Chenxiong Qian. Rootguard: Protecting rooted android phones. *Computer*, 47(6):32–40, 2014.

[104] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, 2008.

[105] Christian Sillaber. D2.3 -Software for a Model-Driven Policy Design. PoSecCo Deliverable, PoSecCo Consortium, 2012.

[106] Sirin, B Parsia, BC Grau, A Kalyanpur, and Y Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5:51–53, June 2007.

[107] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[108] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7, 2007*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[109] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Network and Distributed System Security Symposium (NDSS 13)*, 2013.

[110] Softweg. Benchmark. `https://play.google.com/store/apps/details?id=softweg.hw.performance` .

[111] Stanford University. *Protege*, 2011.

[112] Titanium Track. Titanium Backup. `https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup` .

[113] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[114] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 160–169, New York, NY, USA, 2006. ACM.

[115] A. Uszok and J. Bradshaw. Kaos ontologies. `http://ontology.ihmc.us/ontology.html`.

[116] K. Wang, D. Billington, J. Blee, and G. Antoniou. Combining description logic and defeasible logic for the semantic web. *Rules and Rule Markup Languages for the Semantic Web*, pages 170–181, 2004.

[117] WS-Policy. Web services policy framework. `http://www.106.ibm.com/developerworks/library/specification/ws-polfram/`.

[118] Chao Yang, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Detecting money-stealing apps in alternative android markets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 1034–1036, New York, NY, USA, 2012. ACM.

[119] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, and P. Mohapatra. FIREMAN: a toolkit for firewall modeling and analysis. In *Proc. of IEEE S&P*, 2006.

[120] C. Zhao, N.M. Heilili, S. Liu, and Z. Lin. Representation and reasoning on rbac: A description logic approach. *Theoretical Aspects of Computing–ICTAC 2005*, pages 381–393, 2005.

[121] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.