INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Access Control in Rich Domain Model Web Applications

João de Albuquerque Penha Pereira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Doutor João António Madeiras Pereira
Orientador:	Doutor João Manuel Pinheiro Cachopo
Co-Orientador:	Doutor Carlos Nuno da Cruz Ribeiro
Vogais:	Doutora Maria Dulce Pedroso Domingos

Novembro 2010

Acknowledgements

First and foremost, I would like to thank my adviser Professor João Cachopo. The scope of my work is ground for many uncertainties, discussions, and directions, and without him helping to guide me through, it would hardly be possible. I have learned much from our discussions and his critical judgement always with an open mind to new ideas. This dissertation would not have been shaped as it is without his experienced guidance.

I would also like to thank Professor Carlos Ribeiro and Professor Dulce Domingos for providing me with different insights in the access control area.

I would like to give a special thanks to everyone at INESC-ID, and specially the members of the ESW group, for all their effort in engaging in discussions and giving new ideas during our meetings, providing me with different and important perspectives.

Finally, and although not contributing in an academic way, I would like to thank my friends and family. Particularly my mother Maria da Conceição, my father Rui Pereira, and my sister Joana Pereira, I thank them for all their patience, time, and encouragement.

Lisboa, November 25, 2010
João de Albuquerque Penha Pereira

Resumo

Os sistemas de informação estão cada vez mais omnipresentes no dia-a-dia das pessoas. Contudo, o seu desenvolvimento continua a ser específico a cada problema. Para melhorar este processo de desenvolvimento, vários programadores começaram a adoptar novas abordagens de desenvolvimento, como o Desenvolvimento Orientado ao Domínio. Esta abordagem defende que o desenvolvimento deve ser centrado nas entidades que compõem o problema a ser resolvido por uma aplicação, as quais, devem ser estruturadas num modelo de domínio. Aplicações de Modelos de Domínio Rico (MDR) apresentam um modelo de domínio com uma estrutura complexa e comportamento rico, apresentando um sério desafio no que toca à implementação de segurança e de controlo de acessos.

Linguagens de Especificação de Políticas são Linguagens Específicas de Domínio desenhadas para definir e expressar a política de um sistema. A Domain Model Authorization Language (DMAPL) é uma Linguagem de Especificação de Políticas cujo objectivo é definir e gerir políticas de controlo de acessos em aplicações com MDR. A DMAPL faz também parte da DMAPL framework que, para além da própria linguagem, contém um modelo e um mecanismo de runtime.

Nesta dissertação, o desenvolvimento da DMAPL framework foi concluído. Foi também feita uma integração com a Fénix Framework, criando para tal um plugin de controlo de acessos. Esta integração permitiu a implementação da DMAPL framework numa aplicação real com MDR, nomeadamente, o FeaRS (Feature Request System). Foi também introduzida uma nova regra de controlo de acesso que permite à DMAPL framework expressar controlo de acessos ao nível do domínio.

Palavras-chave
Controlo de Acessos
Aplicações de Domínio Rico
DMAPL framework
Fénix Framework
Relações de Domínio
FeaRS

Abstract

Information systems are becoming more and more ubiquitous in people's daily life. However, their development process remains rather ad-hoc. To improve this process several approaches are being explored such as the Domain-Driven Design (DDD). DDD concentrates on the domain of a system and strongly supports that complex domains should be based on a model, describing all its relevant entities and the relationships between them. DDD tries to leverage on all the advantages of the object-oriented paradigm, leading to a domain model whose entities contain both data and behavior. To such a model we call Rich Domain Model (RDM). RDM web applications present a serious challenge in the security and access control area.

Policy Specification Languages are Domain Specific Languages designed specifically to define and express a policy of a system. The Domain Model Authorization Policy Language (DMAPL) is a Policy Specification Language which aims specifically at the expression and management of access control policies in RDM Web applications. It is inserted in a wider framework, the DMAPL framework, which also contains a model and a runtime engine.

In this dissertation, I completed the development of the DMAPL framework, and integrated it with the Fénix Framework creating an access control plugin. This allowed me to implement and test the DMAPL framework in a real RDM web application, the FeaRS (Feature Request System). Moreover, I introduced a new type of access control rule to enable the DMAPL framework to express access control at the domain level.

Keywords
Access Control
Rich Domain Model Applications
DMAPL framework
Fénix Framework
Domain Model Relations
FeaRS

Contents

List of Figures

Listings

List of Acronyms

RDM Rich Domain Model

DMAPL Domain Model Authorization Policy Language

FeaRS Feature Request System

RDO Root Domain Object

Chapter 1

Introduction

Nowadays more and more people rely on information systems to help them in their daily life. Whether for work or entertainment, information systems are gaining an increasingly ubiquitous presence in people's life. A key player in this fact is the World Wide Web and all the applications being developed in, and for it. Traditionally, web applications used to be made of a couple of static pages with a client-server architecture. Today, web applications are progressively becoming complex and distributed systems that highly enhance the user interaction with the World Wide Web at several levels.

Despite the important role of web applications now played all over the world, the development process of these remains rather ad-hoc. Also, web applications typically have a short development cycle when in comparison with traditional applications due to time-to-market requirements [29]. To improve this process, and also to be able to build new and better systems, developers are exploring new approaches. One which is getting a lot of attention lately is the Domain-Driven Design (DDD) [7]. DDD concentrates on the domain of a system and strongly supports that complex domains should be based on a model[1], describing all its relevant entities and the relationships between them. By modeling the real world domain, DDD tries to leverage on all the advantages of the object-oriented paradigm.

One of the basic ideas of the object-oriented paradigm is to put data and behavior together. This, combined with DDD, leads to a domain model whose entities contain both data and behavior (this behavior can also be seen as domain logic or business rules). To such a model we call Rich Domain Model (RDM) [1]. RDM web applications often have a thin service layer which invokes the domain objects and their behavior, supplying an entry point to the system's domain objects.

Since RDM web applications usually have large and complex domains, these can be seen as an interconnected web of domain objects that handle large amounts of information [10]. At this point, several security concerns arise upon such systems, and how we can perform any kind of security control over such information becomes a critical problem. However, several obstacles appear when trying to enforce access control policies in RDM web applications due to the complexity of the systems, the different technologies working and communicating together, the limitations of the World Wide Web, among others.

Previous research on this subject was made by Dumiense [5]. In his work, he identified the common problems when trying to enforce access control policies in RDM web applications as being code scattering and tangling, expressing and enforcing complex rules, introducing dependencies between the code, and lack of support for the delegation of rights. He then developed a solution, the DMAPL framework, to solve such problems. The DMAPL framework is composed by three components: a model that supports authorization, amplification of privileges, and delegation of rights; a Domain Specific Language (DSL) called Domain Model Authorization Policy Language (DMAPL) based on the previous model with special constructs to help express complex access control rules; and a runtime engine to enforce the access control

[1] `http://en.wikipedia.org/wiki/Domain_model`

rules specified with the DMAPL upon the application. Though in a good stage of development, this solution still needs several improvements and an implementation in a real system for further study and analysis of its capabilities. Therefore, my research and work was guided to try to improve this solution as well as implement it in a real system to be able to get reliable feedback about its capabilities and possible problems.

1.1 Goals

The main goal of my work was to continue the research and development of the solution described by Dumiense [5]. Specifically, I first intended to review the solution and do any development needed.

Afterwards, I intended to validate the solution with an implementation in a real RDM web application. However, to use the solution with a real RDM web application, I first needed to integrate it with the Fénix Framework. The information obtained regarding the actual solution's capabilities guided the next steps in its development. Specifically, new motivations were found to extend the DMAPL and improve its expressiveness.

Some issues that were not entirely addressed by Dumiense [5] were also part of my working goals. One was to have a framework that can be easily used by a team of Java developers (such as the one behind FénixEdu [8] - another RDM web application used by my University) when specifying access control policies. Therefore, the development of the framework had to take in consideration the Java programming language and the OOP paradigm, and how its developers usually work and develop solutions.

The ultimate goal of this work was to have a usable and extensible access control mechanism specially crafted for Rich Domain Model web applications.

1.2 Methodology for Evaluating the Work

Testing and evaluating any piece of software currently being developed is a crucial step. Only in that case may we get reliable feedback about the software's performance and possible problems. Furthermore, it enables us to study and analyze its capabilities. With the collected information, the appropriate next steps of the software's development can be elaborated.

The system that I used to implement and test the solution in [5] was the FeaRS (Feature Request System). The FeaRS is a real RDM web application developed to manage suggestions of features for several systems. Its purpose is to gather suggestions inputed by users, and through a voting system, rank those suggestions. The ultimate goal is to improve the services provided by a system. The FeaRS has been fully implemented and used in a real environment for some time now, making it a good candidate to test our solution and provide us reliable feedback with real data.

The granularity level of the access control solution in [5] is the Java interface method, corresponding directly to the current access control mechanism available in the FeaRS web application. Based on this, after the implementation of the solution in the FeaRS, I tried to assess if the solution was functioning correctly and solving the desired problems previously mentioned.

1.3 Main contributions

To achieve the goal of having a usable and extensible access control mechanism specially crafted for Rich Domain Model web applications, several contributions were made throughout my work. More specifically:

- The development of the DMAPL runtime engine was concluded.

- The DMAPL framework was integrated with Fénix Framework, where an access control plugin was created for this purpose.
- A new type of access control rule was added to the DMAPL framework, enabling this way the framework to provide access control at the domain level.
- An implementation of the DMAPL framework in a real RDM web application was done, validating the framework's purpose in successfully satisfying all the access control requirements of the target application.

1.4 Dissertation outline

This dissertation is divided in six chapters:

- *Introduction.* This is the present chapter, in which I introduce the scope and goals of my work, as well as its main contributions.
- *Related Work.* In this chapter I give an overview of the work related to the subject of this dissertation.
- *DMAPL framework and the Fénix Framework.* In Chapter 3 I describe the development made in the DMAPL framework, the necessary integration with the Fénix Framework, and the development of an access control plugin.
- *Domain Model Relations.* A new type of access control rule is proposed and described in Chapter 4, with the goal of allowing access control at the domain level.
- *Validation of the DMAPL framework.* The validation of the DMAPL framework is achieved in Chapter 5. In this chapter I describe and discuss the several exercises made with the DMAPL framework in a real RDM web application.
- *Conclusions.* Finally, in Chapter 6 the final conclusions are presented, and future work is proposed.

Chapter 2

Related Work

In this section, I shall present and discuss some of the relevant work that has been made in the area of access control with a special focus on RDM web applications. I will begin by introducing the important Role-based Access Control (RBAC) and discuss some RBAC-based solutions (Subsection 3.1). Afterwards, I will introduce and discuss three solutions based in Java (Subsection 3.2), and relevant Policy Specification Languages (Subsection 3.3). Finally, I will briefly introduce the concept of access control in web applications (Subsection 3.4).

2.1 RBAC-based solutions

Ever since security became a major concern in Information Technology, the security community has developed several models to support the desired security mechanisms. Role-based Access Control (RBAC) has emerged as a promising alternative to traditional mandatory access control (MAC) [19] and discretionary access control (DAC) [2] models, which have some inherent limitations. Several beneficial features make RBAC better suited for handling access control requirements of diverse organizations.

The RBAC model can be described by:

- Entities
  - Users, Roles and Privileges
- Relationships between these entities
- Constraints over these relationships

The RBAC model groups individual users into roles that relate to their position in an organization, and assigns permissions to roles according to their status within the organization.

One important organizational process that affects the access control privilege distribution among users is delegation. Delegation consists of a user passing its authority to another user. Delegation of authority is an important functionality that should be captured by any access control model but, despite this fact, it is not supported by the standard RBAC. However, several extensions have been proposed to add support for delegation in the RBAC model, and I shall now present three of these proposals that are relevant regarding access control in RDM web applications.

2.1.1 User-to-User Delegation

In [26], it is proposed a simple and straightforward method to support user-to-user delegation in RBAC. The core of this work is the support for fine-grained delegation. Instead of a user having to delegate

an entire role to another user, the user can choose which specific set of permissions of that role are to be delegated to the other user. To delegate a permission, a user has to have not only direct access to the specific permission, but also the right do delegate it. When a user delegates the delegation of a permission, the grantee (the user who receives the delegation) automatically gains direct access to the permission to be able to further delegate it.

When delegating, the user can specify if the delegation is **restricted** or **unrestricted**. For this, [26] provides a rich set of controls regarding **delegation chains**. A user can control how a delegation will be further delegated (specifying its maximum depth for example), and also set **generic constraints** upon the delegation. This way, a user may hold **total delegation** or **conditional delegation** rights. With generic constraints, it is possible to preserve certain organizational-level properties in the system regardless of any delegation that may occur.

To summarize, a delegation is only accepted by the system if: (1) the grantor (the user who delegates) has the right to delegate the specific permission; (2) the grantee satisfies all the restrictions in the delegation right hold by the grantor (conditional delegations); (3) and the delegation itself does not violate any generic constraint specified in it.

[26] also supports the revocation of delegations, even when these are inserted in chains of delegations. An interesting extension to the model considered by the authors is the possibility to support time-restricted delegations.

2.1.2 Hybrid Hierarchies

Several RBAC-based delegation models have been developed but most of them considers a general hierarchy type. In the context of [12], multiple types of hierarchies have been proposed and considered desirable to have different semantics associated with roles and thus, allowing for a more fine-grained delegation and access control. Three types of hierarchical relations within the Generalized Temporal Role Based Access Control (GTRBAC) framework [12] have been identified: inheritance-only hierarchies (**I-hierarchy**); activation-only hierarchies (**A-hierarchy**); and inheritance-and-activation hierarchies (**IA-hierarchy**). I-hierarchies support permission-inheritance semantics, A-hierarchies support role-activation semantics (a user can activate any role to which is assigned to acquire the role's permissions), and IA-hierarchies support both. An hybrid hierarchy with the previously described hierarchy types coexisting enables the capture of a fine-grained inheritance semantics, making then more fine-grained delegation semantics with practical applications possible. [11] addresses role-based delegation schemes in the presence of hybrid hierarchies.

Two new concepts are introduced by [11]: **filter roles** to enable partial delegation; and a hybrid hierarchy where **upward delegation**, previously often considered irrelevant, now plays a major role.
Upward delegation is used to facilitate fine-grained delegation in the presence of hybrid hierarchies and to provide more fine-grained support for accountability and delegated authority.
In upward delegation, accountability is highly necessary. For example, it may be necessary for a user who is assigned to a junior role to delegate his authority to a user assigned to a senior role. However, the grantee should be made accountable for any work that he does on behalf of the delegator.
The key to achieve accountability is to record the roles that are active in the grantee's session and the permissions that have been acquired through those roles (see Figure 2.1 on page 7 for an example).

The main supported types of delegation are **total delegation** and **partial delegation**.

Total delegation is the delegation of the entire set of permissions that the grantee can acquire by virtue of his membership to the delegator role.

Partial delegation can be achieved in two ways: through dynamic **Block Assignments** (BA) that are

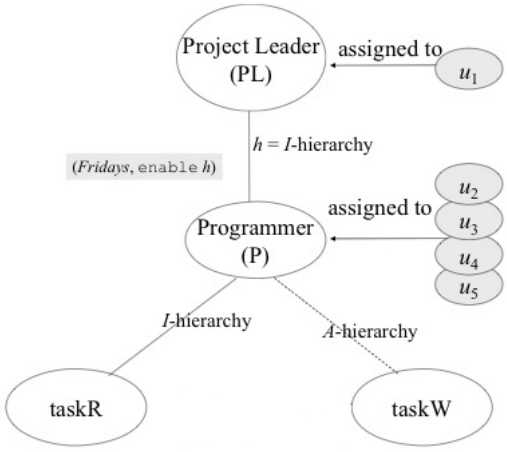

Figure 2.1: Example of an upward delegation in a hybrid hierarchy. The project leader (grantee) mainly supervises the programming tasks. Only the programmers (grantors) do the coding. The project leader can only look at the tasks of the programmers each Friday. Role TaskR contains the read-only permissions whereas role TaskW contains all the write permissions related to the programming task. The Project Leader role becomes the senior of Programmer role only on Fridays. Note that the users assigned to the Project Leader only inherit TaskR permissions and cannot acquire any permissions of TaskW.

supported by filter roles; and through static **Total Explicit Assignment** (TEA). The BA method is preferable when the permissions that are to be blocked from being delegated are relatively small. On the other hand, the TEA scheme is more appropriate when only a small subset of permissions is to be delegated.

To conclude, [11] presents an upward delegation with an important role within the RBAC models. Cross-sectional delegation is not addressed but the authors mention that it is simpler than delegation in the presence of hierarchies. Lack of constraints on delegation (like time intervals and duration) and multi-step delegation and revocation schemes are a part of the future work. The development of a generic analysis framework for verifying the correctness of policies when hierarchical, separation-of-duty, and delegation policies co-exist is also planned.

2.1.3 RT: a Role-based Trust-management framework

Nowadays, more and more independent organizations form coalitions whose membership and very existence change rapidly. The framework described in [14] aims to address access control and authorization in such scenarios with large-scale and decentralized systems.

Inserted in the project of Agile Management of Dynamic Collaboration (AMDC), [14] provides several security features specially crafted for distributed security management. It supports policy language, deduction engine, and features such as application domain specification documents that help distributed users to manage system policies.

In general, Trust Management frameworks cover both centralized and decentralized security management. Although some of this type of work is out of our scope, which is to have a centralized global policy management system, it is important to analyze how [14] combines delegations and roles.

In [14], policy statements take the form of *role definitions*. An example of a simple role definition is

$$K_a.R \longleftarrow K_b$$

$K_a.R$ is the head of the role where K_a is a principal and R simply a role name. K_b is the body of the role

and the previous role definition states that K_b is a member of $K_a.R$. An example of a simple delegation is

$$K_a.R \Longleftarrow Kb : K_c.R_2$$

The part after the colon is optional ($K_c.R_2$). The delegation above specified states that K_a delegates its authority over R to K_b, allowing K_b to assign members to R. When $K_c.R_2$ is present, K_b can only assign members of $K_c.R_2$ to be member of $K_a.R$.

Similarly to [11], this framework also supports **delegation of role activations** which are represented by a *delegation credential*. An example of this is

$$D \xrightarrow{DasA.R} B_0$$

where principal D activates the role $A.R$ to use in session B_0. **Requests** are also represented with a delegation credential that delegates from the requester to the request. An example of such delegation is

$$B_1 \xrightarrow{DasA.R} fileAccess(read, fileA)$$

where B_1 requests to read fileA with the capacity of *"D as A.R"*. This request is accepted if: D is a member of the role $A.R$; the role $A.R$ has read access to fileA; and there is a chain of delegation from D to B_1 about the role activation $A.R$. Here, fileAccess(read, fileA) is not a common principal, instead it is a dummy principal representing the request. The framework assigns a unique dummy principal to each request.

Delegation of role activations is the delegation of the capacity to act in a role. This type of delegation is different from the delegation of authority to define a role.

An important dichotomy to refer about delegation is the difference between the act of delegating, and the act of controlling the delegation. The act of delegating can be seen as a policy management act, defining which authorizations are given to whom. But when discussing delegation, it is common to only define the mechanism that controls the delegation, and not the action of delegation itself. Consequently, the delegation mechanism of a system is often unknown, like in GTRBAC. In RT, the act of delegating is clear, since it is simply an attribution of a role to a user. Whether that role may be attributed to the specified user or not, is already the delegation's control mechanism responsibility.

2.2 Java based solutions

Object-oriented programming (OOP) makes use of objects and the interactions between them to build applications. OOP is an increasingly used paradigm with a great portfolio of success. A sign of that is the appearance of new programming languages supporting this paradigm such as Java [6]. Java, amongst other languages of the same kind, has the advantage of its virtual machine being implemented in several Operating Systems (Windows, Linux, Mac OS X, Solaris, etc), which makes the language highly portable. Therefore, and also due to great community support, Java becomes a great language of choice to develop applications that want to exist independently of the underlying hardware.

RDM web applications, as introduced before, are applications in which domain entities as well as their behavior are represented by objects. This fact makes Java play a key role not only in object-oriented programming but also in this type of web applications.

Java by itself has already some security mechanisms such as visibility qualifiers that may be applied to methods, attributes, among others. However, the language support for access control in general lacks in many concepts and features. In the following, I discuss JAAS, Zás, and ORBAC, which are three Java-based access control mechanisms that try to cover these features.

2.2.1 JAAS

Java Authentication and Authorization Service (JAAS) [13] is a Java security framework that can be used for authentication and authorization. Authentication of users consists in securely determining who is running the code. On the other hand, authorization ensures that users performing some action have the required permissions.

For authentication JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework [25]. This way, authentication in JAAS is modular and independent of the actual technology that performs the authentication, being possible to specify for the same application several forms of authentication and change between them.

JAAS authorization uses Subjects to represent authenticated users. When authenticated, Subjects are associated with a set of Principals, where each Principal represents an identity of the Subject. For example, a Subject may have a Name Principal ("John Smith") or a Phone Number Principal ("987-654-321") by which it can be distinguished from other Subjects. Subjects may also have Credentials, which are security-related attributes.

Every time a Subject tries to execute an access controlled action, JAAS will verify if the Subject, together with his set of Principals, has permission to perform such action. Principal-based permissions are specified in a policy file. An example of a simple permission definition is shown in Listing 2.1, where *user1* has *read* access to file "foo.txt" while executing the code in SampleAction.jar.

Listing 2.1: Example of a permission definition in JAAS.

```
1 grant codebase "file:./SampleAction.jar",
2    Principal sample.principal.SamplePrincipal "user1" {
3  permission java.io.FilePermission "foo.txt", "read";
4 };
```

For each action to be access controlled, changes in the action code are required. More specifically, each access controlled action must implement the `java.security.PrivilegedAction` interface, and thus the method `run` that should contain all the code to be executed (see Listing 2.2). These access controlled actions must then be invoked through the special method `doAsPrivileged(Subject, Action, AccessControlContext)` assigning a subject and an access control context to a specific action's execution (line 3 in Listing 2.3 on page 10).

Listing 2.2: Example of an access controlled action in JAAS.

```
1  public class SampleAction implements PrivilegedAction {
2
3    public Object run() {
4
5      File f = new File("foo.txt");
6      System.out.print("\nfoo.txt does ");
7      if (!f.exists())
8          System.out.print("not ");
9      System.out.println("exist in the current working directory.");
10     return null;
11   }
12 }
```

Listing 2.3: Invocation of an access controlled action in JAAS.

```
1 ...
2         PrivilegedAction action = new SampleAction();
3         Subject.doAsPrivileged(mySubject, action, accessControlContext);
4 ...
```

JAAS has several disadvantages such as permissions' definitions being static only (permissions cannot be changed at runtime), the scattering and tangling of the access control checking code with the rest of the application's code (see Listing 2.2 and 2.3), not to mention that basic and desirable access control features, such as delegations, are not addressed at all.

2.2.2 Zás

Zás [28, 27] is an aspect-oriented authorization mechanism for Java. To handle the concept of aspects, Zás uses AspectJ. AspectJ is an aspect-oriented extension for the Java programming language developed by the Eclipse Foundation, and has become the widely-used de-facto standard for aspect-oriented programming (AOP) by emphasizing simplicity and usability for end users. Aspects and the aspect-oriented programming paradigm support the separation of cross-cutting concerns and encapsulation in the development of applications while increasing its modularity. Therefore, we can see some reason behind Zás intention in using aspects, as access control is often a cross-cutting concern affecting all the application code and thus, causing code scattering and tangling. This scattering and tangling makes the application code more difficult to read and understand, and consequently to maintain or change. Zás tries to reduce these problems, and it also tries to reduce its intrusiveness in the application code while being easy to use.

Zás was inspired by Ramnivas Laddad's proposal [28] to use AOP to modularize JAAS-based authentication and authorization, decreasing the required configuration effort and making access control dynamic. It uses aspects to perform access control verifications and Java annotations to specify permission requirements to access controlled resources (here annotations can be seen as a metadata layer for the application). A developer annotates the non-private resources that should be access controlled and implements in specific aspects the access control verifications.

The annotation of each resource that we want access controlled does not comply with the idea of diminishing code scattering. However, this can be avoided using AspectJ Inter-Type Declarations (ITDs) to inject annotations, making possible to modularize all access control specifications in a single aspect. Dynamic access control and permissions are achieved by the use of property files with wild-cards and permission changing methods. Zás distinguishes between querying and modifying permissions allowing override and inheritance. It also supports propagation of access control as well as ways of bypassing it.

Propagation of access control can be made in two ways: *deep* and *shallow*. If access to a controlled method annotated with the attribute *deep* is granted to a principal, it will not be automatically granted to all other resources in the method's control flow. On the other hand, when in the presence of the attribute *shallow*, the principal will be granted access to all the resources in the method's control flow, turning off access control during its execution.
The two ways Zás provides to bypass access control are also another interesting feature. One way is to use privileged methods, for which execution will always succeed, since access control is turned off to calls within its control flow. Another way is trust. A method can explicitly say it trusts some class, preventing invocations originated from that class being access controlled.

In Listing 2.4 on page 11 is an example of an access control specification over the method `foo`. This

Listing 2.4: Example of an access control specification in Zás with permissions and trust.

```
1  public class A {
2   @AccessControlled(
3     requires = "aPermission",
4     trusts = { B.class }
5    )
6    public void foo() {}
7  }
8
9  public class B {
10  public void bar() {
11   new A().foo();
12  }
13 }
```

specification states that any method invoking `foo` needs the permission "aPermission", unless it is from *class* B (which is trusted, and so bypasses the access control).

Although still in its early stage development, this authorization mechanism is handicapped by the development of AspectJ and its own limitations, which are pointed by the authors themselves such as the inability to capture package annotations or, ITDs adding annotations to code inside JDK's archives. Also, unlike JAAS, Zás cannot enforce access control for JDK's classes. Another issue to notice is that Zás is intended to be used by the programmer, which can raise some security concerns as well as decentralized implementation and control.

2.2.3 ORBAC

In [9] it is presented a fine-grained access control with object-sensitive roles called Object-sensitive RBAC (ORBAC). ORBAC is a generalized RBAC model for object-oriented languages, such as Java, that tries to solve some limitations of the RBAC model. Consider the following example: imagine a medical records system with patients and doctors, where doctors can edit the patients records, and patients can view their records. With RBAC alone it is difficult to express that doctors can only update the records of their own patients, and that patients can only view their own records since in RBAC you cannot distinguish between users with the same role. Therefore, to achieve such policy, programmers have to insert manual access checks, which are error prone and have no assurance that they follow the desired policy. An example using Java EE[1] can be seen in Listing 2.5 on page 12. In lines 17 to 21 we can see the code that checks if the requesting patient's `id` is the same as the patient he's trying to access, and in lines 23 to 27 we can see the code that checks if the requesting doctor is a doctor of the patient he's trying to access.

Java EE supports the specification of an RBAC policy using the `@RolesAllowed` annotation to specify which methods are protected, and which roles are required the user to hold when invoking those methods. Other issue referred by the authors is the reliance solely on dynamic checks by RBAC-based systems. In such systems, it is difficult to ensure that the desired policy is being implemented, and feedback is only given at runtime. This feedback is usually a runtime role failure when trying to invoke a privileged operation, which also makes the problem difficult to diagnose. Summarizing, ORBAC focuses on two key RBAC limitations:

- Lack of expressiveness
- Lack of static checking

[1] `http://www.oracle.com/technetwork/java/javaee`

Listing 2.5: RBAC model applied to a Patient class using Java EE.

```
1  public class Patient {
2   /* factory method to retrieve a patient */
3   @RolesAllowed({"Doctor", "Patient"})
4   public static Patient getPatient(int pid) { ... }
5
6   @RolesAllowed({"Doctor", "Patient"}) { ... }
7   public List<String> getHistory() { ... }
8
9   @RolesAllowed({"Doctor"})
10  public void addPrescription(String prescription) { ... }
11  ...
12 }
13
14 public class PatientServlet {
15  void displayHistory(int pid, Request req, Response rep) {
16
17    if (req.isUserInRole("Patient")) {
18     if (req.userId != pid) {
19      throw new AccessError("Cannot access this patient");
20     }
21    }
22
23    if (req.isUserInRole("Doctor")) {
24     if (!isDoctorOf(pid)) {
25      throw new AccessError("Cannot access this patient");
26     }
27    }
28
29    Patient p = Patient.getPatient(pid);
30    List<String> hist = p.getHistory();
31    ... code to write html representation of hist to resp ...
32 }
```

The idea behind ORBAC is to parameterize roles and privileged operations to distinguish users with the same role from one another. This way, unlike traditional RBAC policies which control access at the class level, ORBAC is able to provide **access control at the level of the object**.
The example in Listing 2.5 on page 12 is reimplemented in Listing 2.6 using an ORBAC policy. In this implementation two roles are used: Patient and DoctorOf, and both are parameterized by a patient identifier. The Patient role is assigned to any patient parameterized with his own identifier, and the DoctorOf role is assigned to any doctor with the identifier of his patient. Each patient will then be able to access only its own records, and each doctor will only be able to access his own patients records.
To implement the solution described above ORBAC makes use of Java annotations. To declare a role parameter the annotation `@RoleParam` is used. This declaration can also occur inside a method's signature defining that parameter as a role parameter. The `@Requires` annotation (user control) is similar to Java EE's @RolesAllowed annotation. In addition, the `@Requires` annotation also specifies which parameter (the patient identifier in this case) will be used to index the required roles. The `@Requires` annotation can be seen as a form of method precondition, while the `@Returns` annotation (code control) is a form of method postcondition. In the given example the `@Returns` annotation on the getPatient method assures that the returned Patient object has a `patientId` role parameter equal to the patient identifier passed to the method.

Listing 2.6: ORBAC model applied to a Patient class.

```
1  public class Patient {
2   @RoleParam public final int patientId;
3
4   /* factory method to retrieve a patient */
5   @Requires(roles={"DoctorOf", "Patient"}, params={"pid", "pid"})
6   @Returns(roleparams="patientId", vals="pid")
7   public static Patient getPatient(@RoleParam final int pid) { ... }
8
9   @Requires(roles={"DoctorOf", "Patient"},
10     params={"this.patientId", "this.patientId"})
11  public List<String> getHistory() { ... }
12
13  @Requires(roles="DoctorOf", params="this.patientId")
14  public void addPrescription(String prescription) { ... }
15  ...
16 }
17
18 public class PatientServlet {
19  @Requires(roles={"DoctorOf", "Patient"},
20   params={"pid", "pid"})
21  void displayHistory(@RoleParam final int pid, Request req,
22      Response resp) {
23   Patient p = Patient.getPatient(pid);
24   List<String> hist = p.getHistory();
25   ... code to write html representation of hist to resp ...
26  }
27 }
```

The `@Requires` annotations do not introduce a dynamic check. Instead, all calling code is statically checked to ensure at least one of the required roles is held. To enable static checking, [9] provides a type system that statically ensures that the desired policy is met. This system is implemented as a pluggable type system for Java in the JavaCOP pluggable types framework [17], and provides early feedback on

potential access control violations. ORBAC can this way avoid manual access checks, since they are now a part of the access control policy reflected in the `@Requires` annotations.
However, not all code can be statically checked, since ORBAC can't always assume users will hold certain roles. To allow an unprotected method to call a protected method by a `@Requires` annotation, ORBAC provides a mechanism for interfacing with the program's authentication and authorization logic through the definition of role predicate methods (user control). These methods, identified by the `@RolePredicate` annotation, are treated as black boxes in the ORBAC's type system and should check if the user holds a certain role. ORBAC only assures that every method with roles requirements using the `@Requires` annotation are called after a role predicate method, incorporating this way dynamic checks into the system (see lines 2 to 6 in Listing 2.7).

Listing 2.7: Example of `@RolePredicate` methods in ORBAC.

```
1  public class Request {
2          @RolePredicate(roles="Patient", params="pid")
3          public boolean hasPatientRole(@RoleParam final int pid) { ... }
4
5          @RolePredicate(roles="DoctorOf", params="pid")
6          public boolean hasDoctorOfRole(@RoleParam final int pid) { ... }
7  }
8
9  public class PatientServlet {
10         void displayHistory(@RoleParam final int pid,
11                             Request req, Response resp) {
12                 if (!(req.hasPatientRole(pid) ||
13                       req.hasDoctorOfRole(pid))) {
14                   throw AccessError("Cannot access this patient");
15                 }
16                 Patient p = Patient.getPatient(pid);
17                 List<String> hist = p.getHistory();
18                 ... code to write html representation
19                 of history to response ...
20         }
```

This type of access control policy specification through annotations added to the codebase can lead to code scattering and tangling (see Listing 2.6 and 2.7). However, the authors claim that ORBAC's dynamic and static checks could also be written in a separated language in a different place other than the codebase, and be automatically inserted in the code at compile time or runtime. Another issue the authors mention is that protected methods with `@Returns` annotations returning collections have to be changed and include dynamic checks not to break the access control policy. [9] has augmented the Open Medical Record System[2] with a fine-grained access control policy using ORBAC, and used the JavaCOP checker to statically ensure the absence of authorization errors.

2.3 Policy Specification Languages

In Information Systems, a policy is a rule that defines a choice in the behavior of a system. Policy Specification Languages (PSLs) are Domain Specific Languages (DLSs) designed specifically to define and express a policy of a system.

Separate tools are emerging to specify the security concerns amongst several systems as well as to

[2] `http://openmrs.org`

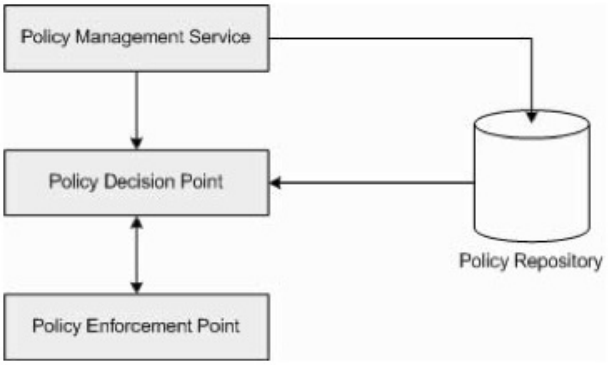

Figure 2.2: IETF's proposed architecture for policy-based management

support a policy-based management. What is lacking is a common language that will support the required concepts, in an unified approach, of the policy models constantly being developed by various research communities. One example is the architecture for policy-based management proposed by IETF [21].

In this architecture (see Figure 2.2) there is a Policy Management Service (PMS), Policy Decision Point (PDP), and a Policy Enforcement Point (PEP). The PDP processes the policies along with other data and decides what policies should be enforced. Afterward, this information is sent to the PEPs that are responsible to implement and enforce those policies. The PMS provides an interface to specify and manage the policies in the system. Here, a policy specification language is highly desired to help in such tasks and also to provide an abstraction of the underlying technology.

When defining a PSL, there are several requirements that the language should satisfy. These requirements include: support for security policies for access control, delegation of rights, and support for policies to express management activity; structuring capabilities to enable policies to relate to large collections of objects instead of individual ones; composite policies which are essential for the administration of policies applied to large organizations; detection of conflicts and inconsistencies (with declarative languages this type of analysis becomes easier); extensibility to ensure that policies can constantly meet new demands (this can be supported by inheritance in an object-oriented language for example); and to be comprehensible and easy to use.

A great property of PSLs is that they're independent of the runtime engine that will enforce the policy in the system. Therefore, and since the policy specification is decoupled from the implementation, it is possible to modify the policy of a system changing dynamically its behavior to adapt it to new needs. PSLs allow, this way, the definition of a single global security policy decoupled from the enforcement and its underlying technology, allowing also the definition of various security policies using the same language even if for different technologies/systems. Other advantage of this abstraction is also the separation of concerns as well as management and flexibility improvement.

I now introduce and compare the following PSLs: XACML; Ponder; Ponder2; SPL; and DMAPL.

2.3.1 XACML

The XACML (eXtensible Access Control Markup Language) is considered by OASIS[3] to be a standard, general purpose access control policy language defined using XML [15]. The XACML was designed to be flexible and able to adapt to most of the system needs. The XACML also defines a request and response authorization format for decision requests with semantics associated to determine the applicability of policies to those requests. This way the XACML becomes suitable to tie together multiple and

[3] `http://www.oasis-open.org`

heterogenous authorization systems.

Although the XACML is not a complete standard solution, it provides a good foundation to build new solutions upon. The XACML defines the syntax for a policy language and the semantics for processing those policies. The request and response format are a standard way of communicating authorization requests between PEPs and a PDPs, following the proposed architecture by IETF [21].

An XACML policy consists of an arbitrary tree of sub-policies. Each tree represents a target, and its leaves a collection of rules applicable to the corresponding target. An XACML authorization request contains attributes about the subject, the target, the action and the context (see Listing 2.8 on page 17). A response contains one of the following decisions: permit, deny, not applicable, or indeterminate. Each policy uses an extensible system of datatypes and functions for interoperability. On top of this, the XACML provides a mechanism to define new datatypes and functions. Policies may also refer other policies, which may in turn be decentralized and spread throughout several authorization systems. This is possible because the XACML provides a mechanism combining several algorithms to reference and retrieve policies as well as requests' attributes.

The fact that the XACML is an open standard defined in XML makes it more attractive for developers to build new tools to use it as an inter-operability point tying up heterogenous or legacy systems. Also, the fact that the XACML is able to work with decentralized policies and easy to integrate with new systems makes it an excellent solution for distributed authorization systems. However, all of this flexibility comes with a cost, in this case of complexity and verbosity in the language itself. It's not easy to deal with the policies directly, and in order for the XACML to be a competitive solution, assisting tools must be developed.

2.3.2 Ponder

Ponder [4] is a policy specification language originally developed at the Imperial College with independent work further developing it. It is a declarative and object-oriented language with a framework for heterogenous platforms policies. It can also be used for security management activities.

In Ponder a subject refers to a set of users, a target refers to a set of objects and the granularity of protection is an interface method. Domains are used to hierarchically group objects to which policies apply (Ponder was implemented using an LDAP service). Ponder uses a subset of the Object Constraint Language [18] (OCL) to specify constraints in its policies. OCL is a declarative language where each expression is conceptually atomic. This means the state of the objects in the system cannot change during its evaluation.

The types of policies supported by Ponder are:

- Access Control Policies
  - authorization, Information Filtering, Delegation, Refrain Policies
- Obligation Policies
- Composing Policy Specifications
  - Groups, Roles, Type Specialization and Role Hierarchies, Relationships, Management Structures

authorization policies specify what actions (methods) a subject can perform on a target and can be positive or negative (see Listing 2.9 on page 18 for an example). Conflicts between positive and negative authorizations can be detected through static analysis of the policy specification.

Listing 2.8: A file read privilege encoded as an XACML rule component

```
<Rule RuleId="File-Privilege-Rule" Effect="Permit">
 <Target>
  <Subjects>
   <Subject>
    <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function
    :x500Name-equal">
     <AttributeValue DataType="urn:oasis:names:tc:xacml:1.0:data-type
     :x500Name">
      CN=Sumit Shah (sshah),OU=Virginia Tech User,OU=Class 2,O=vt,C=US
     </AttributeValue>
    <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml
    :1.0:subject:subject-id" DataType="urn:oasis:names:tc:xacml:1.0
    :data-type:x500Name" />
   </SubjectMatch>
  </Subject>
 </Subjects>

  <Resources>
   <Resource>
    <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function
    :anyURI-equal">
     <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      gridftp://zuni.cs.vt.edu/data/collaboration/results.dat
     </AttributeValue>
     <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml
     :1.0:resource:resource-id"
     DataType="http://www.w3.org/2001/XMLSchema#anyURI" />
    </ResourceMatch>
   </Resource>
  </Resources>

  <Actions>
   <Action>
    <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function
    :string-equal">
     <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
      Read
     </AttributeValue>
     <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml
     :1.0:action:action-id"
     DataType="http://www.w3.org/2001/XMLSchema#string" />
    </ActionMatch>
   </Action>
  </Actions>
 </Target>
</Rule>
```

Information filtering policies may also be included by positive authorization policies when it is needed to transform the information input or output parameters in an action (negative authorization policies forbid actions and thus, no information is inputed or outputted). These are used when an operation has to be performed and then a decision made on the results transformation. This decision can be based on attributes of the subject or target, or on system parameters (e.g., time).

Delegation policies specify which actions subjects can delegate to other subjects. Delegation policies are always associated with positive authorization policies and specify which access rights can be delegated. Since they create an authorization for delegation, the subject must already possess the access rights to be delegated. These policies specify the authority to delegate, they do not control the actual delegation and revocation of access rights. When a subject executes the delegate method, it is created in runtime a separate authorization policy corresponding to that delegation with the subject as its grantor. Negative delegation policies forbid delegation and do not contain delegation constraints.

Refrain policies define the actions that subjects must not perform. They are similar to negative authorization policies but are enforced by subjects rather than target access controllers. Therefore, they are used when the targets are not trusted to enforce the policies (same syntax as the negative authorization policies).

Obligation policies are event-triggered and define the actions subjects must perform on objects when certain events occur. Composite events can be specified using event composition operators and concurrency operators specifying whether actions should be executed sequentially or in parallel, and therefore separating the actions in an obligation policy. Unlike authorization policies, actions may be internal to the subject.

The composition of policy specifications, as mentioned before, is very important specially when dealing with large organizations. To enable this, Ponder provides the following constructs: Groups; Roles; Type Specialization and Role Hierarchies; Relationships; and Management Structures.

Since all of Ponder's policy types are organized hierarchically, it allows new policy classes that may be identified in the future to be added as sub-classes of existing policy classes.

Listing 2.9: Example of a positive authorization definition where it is stated that all subjects under the /NetworkAdmin domain can perform the listed actions on the targets under the /Nregion/switches domain.

```
1 inst auth+ switchPolicyOps {
2         subject  /NetworkAdmin;
3         target   <PolicyT> /Nregion/switches;
4         action   load(), remove(), enable(), disable();
5 }
```

Ponder's policy compiler can also resolve different types of constraints at compile time, and separate the constraints to aid in the analysis of policies.

Ponder is a deprecated language since it has been replaced by Ponder2. However, for analysis purposes, we conclude that Ponder is not able to express history-based policies nor is able to override policies with amplification of privileges. It does not make use of Java annotations or wild-cards, and its specification is oriented towards directory services instead of objects in a rich domain.

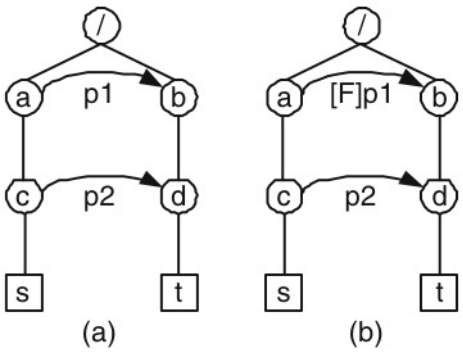

Figure 2.3: Example of priority based on domain nesting (a) and final status (b)

2.3.3 Ponder2

Ponder2 [22] is a policy specification language created based on the original Ponder and its concepts. It is currently being developed at the Imperial College where the main idea is to generalize the authorization model previously supported by Ponder and improve and further develop the language and its framework.

Like Zás, Ponder2 makes use of Java annotations to mark resources to be access controlled. It also introduces a new feature which are the Policy Enforcement Points (PEPs) based on the architecture for policy-based management proposed by IETF [21].

Ponder2, like Ponder, continues to use domains to hierarchically group objects to which policies apply. There is a property that specifies if, by default, all actions are permitted or not. But since its value propagates to all the sub-domains, it can only be applied to the root domain to avoid possible conflicts.

2.3.4 Conflict resolution

Similar to Ponder, Ponder2 also makes static analysis of the policy set to identify and solve conflicts between policies. Unfortunately, conflicts of policies that depend on runtime state can only be detected in runtime. Ponder2 introduces a conflict-resolution strategy for such conflicts [24].

The strategy is called domain nesting and consists in giving priority to policies that apply to a more specific instance of objects. However, there is also a way of defining special policies called *final* that will override more specific ones.

The basic algorithm to conflict-resolution is to choose the most general final policy in the domain, or if no final policies are present, choose the most specific policy in the domain. In both cases, if conflicts arise, negative policies are given higher priority. The previous algorithm can still fail to solve some conflicts, these are solved using the path length of the policies in the domain. Finally if this method also fails, the default policy is applied.

In the Figure 2.3, in the first example (a) the policy *p2* takes precedence over policy *p1* being *p2* more specific than *p1*. In the second example (b) the policy *p1* overrides policy *p2* due to its *final* status.

Applying several filters to various PEPs may also lead to filtering conflicts. A strategy to solve this type of conflicts is also provided by Ponder2.

2.3.5 SPL

SPL [23] is a policy-oriented constraint-based language for expressing security specifications developed at INESC-ID[4] by Professor Carlos Ribeiro and Professor Paulo Ferreira. It has a specific algebra with

[4] `http://www.inesc-id.pt`

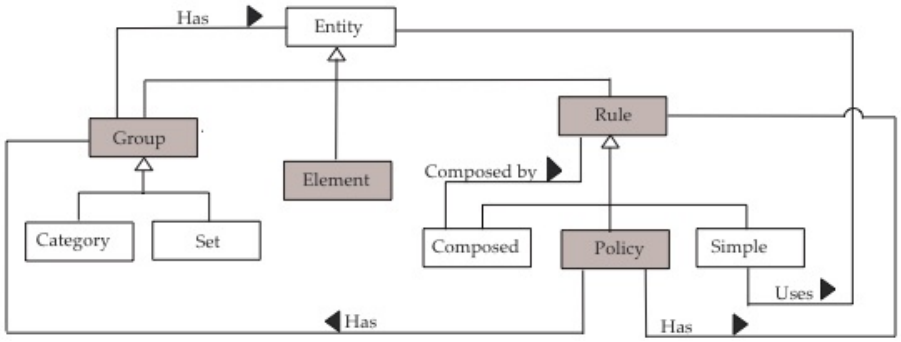

Figure 2.4: SPL's structure and basic blocks

several quantifiers for the composition of policies that provides the necessary flexibility to express many different types of policies. It can express the concepts of permission and prohibition, and some restricted forms of obligation like history-based and obligation-based security policies.

SPL's structure and basic blocks are composed by four entities (see Figure 2.4):

- Elements
- Groups
- Rules
- Policies

Elements are entities with an explicit interface through which their properties can be queried. Each property may be a reference to another element, group or basic type. An element is basically a proxy to an entity in the underlying platform allowing access to the entity's context information. An element can also specialize extending another element.

Groups are classified as categories or sets and can be internal or external.

Rules decide which actions can be performed and can take three values: allow; deny; and notapply. A rule can be simple or composed. A simple rule has an optional label and two logic expressions (see Listing 2.10): the first one determines the domain of applicability of the rule (*domain-expression*); and the second one determines if the request is accepted or not (*decide-expression*). Composed rules are obtained combining simple rules with logic operators. An implicit parameter representing the request also exists in every rule (denoted by 'ce' in the example of Listing 2.11 on page 21).

Listing 2.10: Syntax of a simple rule

```
1 [nonelabel :] domain-expression :: decide-expression
```

Policies are collection of rules and groups. Each policy has one **Query Rule** that relates all the rules specified in the policy providing the ability to merge policies into more complex ones (in SPL roles can be seen as a policy template). Policy inheritance is also supported.

SPL supports two types of special constraints: history-based and obligation constraints.

Listing 2.11: Example of a simple rule where it is stated that payment order approvals cannot be done by the owner of the payment order.

```
1 DutySep: ce.target.type="paymentOrder" &
2          ce.action.name="approve"
3          :: ce.author != ce.target.owner;
```

History-based constraints are based on logs. Here the authors mention a scalability logging problem that can be solved with a simple optimization algorithm.

Obligation constraints are defined as constraints with dependencies in future requests which can be seen as equivalent to triggered obligations. Two generic situations where an obligation-based policy is required are when two actions involved oblige each other, or when the obligatory action is causally dependent on the trigger action. Trigger and obligatory actions are executed inside an ACID[5] transaction allowing the possibility to translate a policy with a dependency in the future into a history-based policy (this process is called *aging*). However, depending on the side effects of the actions this translation may not be possible. For example, if an action prints a document it is something that later on it cannot be undone.

SPL does not provide a specific mechanism for delegation. Instead, it relies on the ability of each user to dynamically insert rules and policies into groups of rules. It also supports the definition of depth and width of a delegation based on the tracking of history-based policies. With history-based policies, it is possible to verify how many times a right has been delegated (depth), and how many times a right has been delegated by a specific entity (width).

SPL also provides mechanisms for the resolution of conflicts between policies. One relevant strategy is to organize the active policies in a hierarchical tree. By doing so, if there are two policies in the tree conflicting upon the same request (one policy allowing and the other denying), at some point in the tree they must be combined by an algebraic expression that inherently solves the conflict. Although this may be an automatic and effective way of solving policy conflicts, it can also mask real problems that may be introduced by the policy administrator for example.

In conclusion, the SPL's construction of rules with domain and decide expressions is more powerful than the permission and prohibition construction, and while supporting history-based policies, these are limited to decide on requests that occur after their activation (the system only starts logging the necessary information to the history-based policies after their activation). SPL also contains a tool to automatically check the coherency of policies, a compiler to Java to enforce the policies with a security monitor, and a graphical interface under development to solve some low level usability problems. Finally, as the authors state, SPL should be mainly used to assemble big policy blocks.

2.3.6 DMAPL

In [5], Dumiense described a complete framework to support access control policies in RDM web applications. This solution aims directly at the problems and challenges faced when trying to enforce access control policies in this type of applications. Such problems may be decentralized security policy management, code scattering and tangling, difficulty in expressing complex access control constraints with a high level of granularity, dependencies between the code, or lack of delegation support. Ultimately, [5] tries to ease the access control task in applications by attempting to solve the problems mentioned above.

The framework is composed by three components:

[5] `http://en.wikipedia.org/wiki/ACID`

- Model
- Domain Model authorization Policy Language (DMAPL)
- Runtime Engine

The model introduces and conceptually supports the following types of access control rules:

Authorization rules are the simplest type of rules and state if a certain user can have access to a given resource under certain circumstances. There are two types of authorization: **positive** and **negative**. A positive authorization states that the authorization can be granted, whereas a negative authorization forbids that authorization from being granted. In case of conflict or doubt, if a positive and negative authorization both exist regarding the same access, the negative authorization prevails and the access is denied. Also, if no positive authorization exists, even if no valid negative authorization exists either, the access is denied. An example of a positive authorization is given in Listing 2.12.

Listing 2.12: Example of a positive authorization rule in DMAPL.

```
1 GiveGradeAccess:
2  allow role Teacher
3  to studentrecord.Student
4      .giveGrade(Course course, noneint grade)
5  where { user.getPerson().getTeacher()
6      .teachesCourse(course) }
```

In this example it is stated that a teacher can give grades to the students of the courses it teaches. The *where* statement is used to specify constraints that must be verified for the authorization to be accepted. An important feature of the DMAPL is its similarity with the Java programming language syntax. Since Java is a widely-used language for the development of RDM web applications, developers who want to use the DMAPL to express security policies will easily adapt to its syntax.

Amplification of Privileges is accomplished using authorization tickets. For example, when a person buys a ticket to see a specific movie at the cinema, the ticket amplifies the privileges of the person allowing her to see the specified movie. After entering the cinema the ticket is revoked from the person, forbidding her from going in again. To specify when a user should receive a ticket, the DMAPL uses a special type of policy rules called amplification rules. Once again, negative authorizations rules have priority over amplification rules.

Listing 2.13: Example of a ticket and amplification rule in DMAPL.

```
1 ticket gradeTicket(Student student)
2  to studentrecord.Grade.getGrade()
3   where { receiver.getStudent() == student }
4
5 OfficerAverageAccess:
6  on studentrecord.Student.getAverage()
7  give role officer ticket gradeTicket(receiver)
```

In Listing 2.13 we can see an example of a ticket and an amplification rule. When an officer tries to get the average of a student, it receives a ticket allowing the officer to obtain the grade of the

student in question. The controlled amplification of privileges makes access control more flexible providing a way to temporarily override previously accepted access control rules.

Delegation of Rights corresponds to the action of a user delegating to another the access to certain rights. Delegation rules define what can be delegated in the system, and delegation rights represent, in runtime, the rights granted to the grantee by the grantor. In fact, a delegated right is nothing more than a new authorization corresponding to the action specified in the delegation rule (as it can be confirmed by the example in Listing 2.14).

Listing 2.14: Example of a delegation rule with constraint and validity constraint in DMAPL.

```
1  TeacherGiveGradeOnVacationDeleg:
2   allow delegation of GiveGradeAccess
3   grantor role Teacher
4   grantee role Officer
5   to studentrecord.Student
6       .giveGrade(Course course, noneint grade)
7   where { grantor.getPerson().getTeacher()
8       .isAlmostOnVacation() }
9   valid { grantor.getPerson().getTeacher()
10      .isOnVacation() }
```

In the example in Listing 2.14, the teacher delegates the action of giving student grades to the officer. The *where* and *valid* statements provide ways of specifying when should the action be delegated, and until when it is valid. Once again, negative authorization rules have priority over delegated rights.

The Domain Model Authorization Policy Language is a DSL defined to express security policies with the described access control rules. It provides special constructs to enable and facilitate the expression of complex access control rules for RDM web applications. In [5], Dumiense states that the DMAPL provides constructs that fit the way developers think when reasoning about security policies. The goals of the DMAPL are essentially: *expressiveness*; *compositionality*; *human readable*; and *enable automation*.

The runtime engine's purpose is to make the enforcing of the access control rules during the application execution. However, further implementation of the engine is needed to support delegation of rights. Testing in a real, large and complex application such as FénixEdu [8] is also highly desired to assess its real performance.

Interesting future work with this framework includes: playing with the expressiveness and efficiency of the engine to find the best solution for RDM web applications; explore the possibility of integration of the engine with other RDM development platforms like the JVSTM [1] to increase its efficiency; study if the constructs provided by the DMAPL are the most adequate for its purpose; and study how it would be possible to support cascade delegation of rights.

2.3.7 Comparison

All the Policy Specification Languages presented are declarative languages, which facilitates the detection of conflicts and inconsistencies.

The XACML is an XML-based language most suitable to tie up large heterogenous authorization systems, which falls out of the scope of this work. Ponder is an object-oriented language making it suitable to specialize policies by inheritance. However, Ponder and Ponder2 are not able to override policies with amplification of privileges like the DMAPL, which is a highly desired feature that enables the enforcement

of access control policies in a compositional manner. Also, Ponder does not make use of Java annotations or wild-cards which is also desired not only by the advantages they present in avoiding code scattering and tangling, but also because they make the framework more suitable for Java developers by using familiar concepts. Ponder and Ponder2 are also oriented towards directory services instead of objects in a rich domain, like the DMAPL.

SPL makes use of a different and more powerful construction of rules in comparison with Ponder, Ponder2, and the DMAPL. However, its syntax has no similarities with the Java programming language which can be a disadvantage when trying to build a framework suitable for Java developers. SPL also lacks of a specific mechanism for expressing delegation (although it supports the concept) which can confuse developers when trying to do so. Moreover, due to SPL's enhanced algebra for composition of policies and the ability to express many different types of policies, the authors state that SPL is better used to assemble big policy blocks.

In conclusion, of the five PSLs presented in this Subsection, the DMAPL appears to be the most capable and well adapted to specify access control policies in RDM web applications. The DMAPL is also a PSL which clearly states to aim at the problems encountered when trying to specify access control policies in such applications. Therefore, I choose the DMAPL to support my following work.

2.4 Access Control in Web Applications

As previously stated, web applications are becoming large and complex systems. Due to its nature, the number of technologies used for their development are enormous, giving web applications a great variety of implementations. When it comes to security specification and management, web applications are far from being optimal, since it is extremely difficult to develop a standard way of enforcing security and access control policies in such applications. One particular web application that follows the same characteristics is the FénixEdu [8] web application. FénixEdu is a large and complex RDM web application that gives all the administrative and educational support required for a University domain. It has already some history behind it and it is nowadays fully used in several Universities.

Research about FénixEdu's current access control mechanisms has been made by Malheiro [16]. In FénixEdu, Malheiro identifies five distinct mechanisms of access control in five different points of the system:

- Services
- Objects
- Functionalities
- Infrastructure
- Implicit in the code

Malheiro [16] concludes that FénixEdu's approach to access control is flexible and reasonably effective while requiring little development effort. But on the other hand, the use of such mechanisms does not support a unified or centralized approach, nor support for the management of a global security policy. Therefore, the current access control mechanisms do not provide to FénixEdu flexibility in its behavior as well as methodologies to monitor and audit its security. The use of the mentioned mechanisms also cause a great deal of code scattering and tangling mixing the access control concerns with the business logic over the entire system.

Chapter 3

DMAPL Framework and the Fénix Framework

As stated before, my work extends previous work by Dumiense [5]. One of the first and important steps that I intended to do with it was to have a more practical approach to the DMAPL framework by integrating it and using it with a real RDM web application, to determine which was exactly the current development and use status of the framework. With this exercise, I came to realize that the DMAPL framework was still not ready to be used as it could be. It had some issues to be solved and further development that could be made. In this chapter I discuss these issues and how I solved them. Next I describe how I integrated the DMAPL framework with the Fénix Framework to use it and test it with real RDM web applications.

3.1 Overview of the DMAPL Framework

Just to give a brief overview, the DMAPL framework has three main components:

- The Model
- The Domain Model Authorization Policy Language (DMAPL)
- The Runtime Engine

The **model** supports the three types of access control rules: authorization; amplification of privileges; and delegation. The **Domain Model Authorization Policy Language** (DMAPL) is the language created to express and define those access control rules. Finally the **runtime engine** is the responsible for enforcing such rules in the designated application during its execution. In short, a developer specifies the access control rules supported by the model in a policy file using the DMAPL, which will in turn be interpreted and enforced in the target application by the runtime engine.

In the beginning of my work both the model and the language were already completely defined and implemented. The runtime engine, however, lacked several fixes and further development to perform as expected. One very important difference between what Dumiense [5] projected and what is implemented in the runtime engine, is the fact that the runtime engine implements an *open policy*, meaning that in the absence of rules the default action is to allow access. Instead, Dumiense [5] intended for the runtime engine to implement a *closed policy* (in the absence of rules the default action is to deny access). It is worth to bear in mind this fact when judging the runtime engine, and planning further development for it.

3.2 Runtime Engine Fixes

I will now describe the fixes that were made in the DMAPL framework's runtime engine, to be able to use it.

3.2.1 Tickets without Amplification Rules

As previously explained in Section 2.3.6, one of the DMAPL framework's type of access control rules are the amplification rules. The amplification rules were created to support the concept of amplification of privileges, which allows to, inside a specific execution context, override some access control rules. An amplification rule works together with a ticket, where the amplification rule specifies when the ticket is to be assigned to the designated user, and the ticket represents the new right to be acquired. An example of an amplification rule and the corresponding ticket can be seen in Listing 2.13 on page 22. However, we may want to specify a ticket without an amplification rule associated with it yet.

For instance, imagine there is a method A() to which we want to give access only in certain circumstances. To achieve this goal we only need to create a ticket for A(), and the corresponding amplification rules covering the situations where we want to assign the ticket to the user in question. However, if no amplification rules exist associated with such ticket, the engine should deny the execution of A(), since A() is a protected method after all, and needs the right ticket to be executed. This was the case that the runtime engine was ignoring, and thus allowing the execution of A() in such cases. This issue has now been fixed.

3.2.2 Multiple code injections

One issue that could increase the overhead of using the DMAPL framework was the multiple code injections. The mechanism that the DMAPL framework uses to enforce the access control policy in an application is the following: after defining the access control rules in the policy file, and before the application's execution starts, the protected methods of the application (the methods covered by the access control rules) are injected with a security check in the beginning of their body. The injection is done using Javassist [3], and the security check is basically a call to the runtime engine that will, in turn, evaluate if the current user is allowed to execute the protected method.

When the access control policy in the DMAPL framework is changed, there is no need to recompile the application's code for the new policy to become effective. Instead, the application's code is injected again so that the new methods will be protected as well. However, if the application's code is injected multiple times (in case of several security policy changes), the result are several injections for the same protected method. More specifically, the following situations would occur:

1. Previously protected methods covered by the new access control policy end up having additional security checks. This results in multiple calls of the runtime engine with the same practical effect as having only one, causing then a bigger overhead in the DMAPL framework's execution.

2. Previously protected methods no longer covered by the access control policy continue to have the previously injected security check. This results in a call of the runtime engine with no practical effect at all, since no rules exist regarding such methods.

Both situations have been addressed, and whenever a method is to be protected, regardless of the number of injections previously performed, it will only contain one security check. As for methods no longer protected, the security checks will be removed accordingly.

3.2.3 Protected methods with null arguments

Once the runtime engine is called through the execution of a protected method in the application, a matching between that method and the set of rules in the access control policy is done, to verify if any rule applies to it (this should return at least one applicable rule, otherwise the injection of the security check in the method would not have been done in the first place). In this matching process the user executing the protected method, its role, and the signature of the protected method are compared. Belonging to the method's signature are its arguments, which are also taken into comparison. The early version of the runtime engine was comparing such arguments using directly the objects representing their values. If one of these values was null, the runtime engine would throw an exception since it is not possible to traceback to the original argument type which was eventually passed as null. The current version fixes this issue by also passing to the runtime engine an array containing the class types of the arguments, and not only their values. This way, the arguments are safely compared regardless of their value at the point of comparison.

3.2.4 Mandatory authorization rules constraints

As mentioned before, authorization rules can have an optional constraint associated introduced with the keyword *where* (see Listing 3.1).

Listing 3.1: Example of a positive authorization rule with constraint in DMAPL.

```
1 GiveGradeAccess:
2  allow role Teacher
3  to studentrecord.Student
4      .giveGrade(Course course, noneint grade)
5  where { user.getPerson().getTeacher()
6      .teachesCourse(course) }
```

But contrary to the model presented, the runtime engine parsing mechanism, that uses the well-known language tool ANTLR [20], was not contemplating this option and was enforcing every authorization rule to have a constraint associated. This issue has been corrected and it is now possible to have authorization rules without constraints (see Listing 3.2).

Listing 3.2: Example of an authorization rule without constraint in DMAPL.

```
1 OfficersCancelCourse:
2  allow role Officer
3  to studentrecord.Student
4      .cancelCourse(Course course, Student student)
```

3.2.5 Policy file parsing optimization

The access control rules written in the policy file will form the access control policy to be enforced in the target application. The first step of the DMAPL framework's execution is to parse the policy file with the runtime engine. In its early version, the runtime engine parses the policy file a first time searching for authorization rules, and a second time searching for tickets and amplification rules. The overhead caused by this was probably as little as the code change needed to parse the policy file only once. But in an

attempt to follow the good practices of software development, the current version of the runtime engine only requires to parse the policy file once to search for authorization rules, tickets and amplification rules.

3.3 Fénix Framework Integration

As previously mentioned, one of my work's goals was to use and test the DMAPL framework in a real RDM web application, to assess the framework's real performance and highlight possible problems. My University[1] has its own information systems and applications, some of them RDM web applications, that give all the administrative and educational support required for a University domain. After having the DMAPL framework finished and ready to use, I decided to use one of these applications to test the DMAPL framework since they are real, large and complex RDM web applications able to provide us with reliable feedback. However, most of the relevant applications to my work's scope run on top of the Fénix Framework.[2]

The Fénix Framework, also developed *in house*, allows the development of Java-based applications that need a transactional and persistent domain model. It was originally created to support the development of web applications but it may also be used to develop any other kind of application that needs to keep a persistent set of data. With the Fénix Framework, the database is completely hidden from the programmers, encouraging them to make use of all of the normal object-oriented programming coding patterns without worrying about persistence. In conclusion, to be able to test the DMAPL framework with any of the applications mentioned before, I had first to integrate it with the Fénix Framework. In the following sections I describe the work done to achieve this goal.

3.3.1 Building applications with the Fénix Framework

Developing an application with the Fénix Framework starts by specifying the structure of its domain model using the domain-specific language called Domain Modeling Language (DML) [1]. The DML is a language specially crafted for the purpose of implementing the structure of an application domain model. After this step, the Fénix Framework generates a generic version of the domain classes. All the domain objects later created (objects that materialize the classes specified in the domain model), will be automatically persisted in a database by the Fénix Framework, and the rest of the application is developed in plain Java.

All the domain objects of an application are linked to at least one object inside the Fénix Framework, which is the Root Domain Object (RDO). The RDO acts as an entry point to the application's domain, and by definition, it should be a *singleton* object. From it, a developer can access all the domain objects of the application. The goal of integrating the DMAPL framework with the Fénix Framework is to have the DMAPL framework's access control mechanisms available to the developer using the Fénix Framework. In other words, through the Fénix Framework, the developer would be able to place the application's users and roles under the DMAPL framework's access control mechanisms, and enforce a security policy by specifying access control rules for them.

3.3.2 Linking domains

During the development of the DMAPL framework, one important aspect always taken into consideration was to make it decoupled from the underlying application. This is important to prevent the DMAPL framework from setting restrictions to the application's domain and development. Even so, the DMAPL framework needs to somehow be aware of the application's users and roles to enforce the access control

[1] Instituto Superior Técnico - `http://www.ist.utl.pt`

[2] `https://fenix-ashes.ist.utl.pt/trac/fenix-framework`

rules upon them. In a regular Java application, the mechanism the DMAPL framework uses to link its own domain with the application's domain, is to enforce the application's users and roles to extend two interfaces: the `AccessControlUser` interface (see Listing 3.3), and the `AccessControlRole` interface (see Listing 3.4). However, with the Fénix Framework between the DMAPL framework and the application's domain, a different approach had to be taken.

Listing 3.3: DMAPL framework's `AccessControlUser` interface.

```
1 package pt.ist.dmapl;
2
3 public interface AccessControlUser {
4  public boolean hasRole(AccessControlRole role);
5 }
```

Listing 3.4: DMAPL framework's `AccessControlRole` interface.

```
1 package pt.ist.dmapl;
2
3 public interface AccessControlRole {
4
5 }
```

To integrate the DMAPL framework with the Fénix Framework while keeping it decoupled from any particular application, a specific access control domain, internal to the Fénix Framework, was created. The purpose of this access control domain is to make the access control infra-structure (the DMAPL framework) available to any application that executes on top of the Fénix Framework. This was accomplished by creating a second RDO inside the Fénix Framework, which holds the access control domain.

However, in the early versions of the Fénix Framework only one RDO was allowed. Because of this, if an application such as the DMAPL framework wanted to make available its own domain and code to the application's domain, it needed to modify some of the internal classes of the Fénix Framework to add and comply with a second RDO. Only this way it was possible to make the DMAPL framework's access control mechanisms available to the application's domain. Also, all the classes of the access control domain associated with the second RDO had to be manually generated. As mentioned before, the classes of the domain model of an application are automatically generated using the DML and the Fénix Framework. So, whenever the Fénix Framework suffers a significant upgrade, most likely the classes generated will also suffer modifications. This meant that, every time the Fénix Framework would suffer an upgrade, the access control domain classes had to be manually generated again, to comply with the new version of the Fénix Framework. Another disadvantage about this solution was that, for an application to use the DMAPL framework's access control mechanisms, and therefore relate to the access control domain, had to explicitly declare it in a special domain model file to be passed to the Fénix Framework. As we can see, this was not a very clean and modular solution.

Throughout the time, several extensions began to be developed for the Fénix Framework, enhancing it with more features. This growing need of adaptability led to the development of a plugin system,[3] to facilitate the development of these applications and bind them with the Fénix Framework and the

[3] `https://fenix-ashes.ist.utl.pt/trac/fenix-framework/wiki/FenixFrameworkPlugins`

application's domain. Therefore, this plugin system provided a really good solution for the integration of the DMAPL framework with the Fénix Framework.

3.3.3 Access Control Plugin

I this section, I will describe how the integration between the DMAPL framework and the Fénix Framework was achieved, using the new plugin system.

Recently, the Fénix Framework was upgraded to support a plugin system that allows the developer to provide self contained components to the framework. One notable change introduced by this plugin system, is that it is now **possible to have several RDOs linked to the Fénix Framework**, and consequently, to the application's domain. In the case of the Fénix Framework, having a plugin architecture brings several advantages. Not only it encourages the reuse of code between the different applications, but it also helps to keep the domain model files of the application cleaner, focusing only in the true application domain and leaving possible infra-structure domain contained in a plugin. This matches the exact problem mentioned in Section 3.3.2, in the first attempt of the integration between the DMAPL framework and the Fénix Framework.

A plugin should be seen as any other application that is based in the Fénix Framework, but with some special properties. It may or may not provide a domain model, and it needs to implement a specific plugin interface (see Listing 3.5).

Listing 3.5: Plugin interface for the Fénix Framework. The `getDomainModel` method should return a list of URLs to the DML files that the plugin needs. The method `initialize` should contain any initialization code.

```
1  package pt.ist.fenixframework;
2
3  import java.net.URL;
4  import java.util.List;
5
6  public interface FenixFrameworkPlugin {
7
8   public List<URL> getDomainModel();
9
10  public void initialize();
11 }
```

In this case, the DMAPL framework provides its own domain model which is the access control domain model, as mentioned before in Section 3.3.2.

3.3.3.1 Access control domain model

In the access control domain model, three classes are defined:

ACRoot which is the RDO of the access control domain, making it available to the Fénix Framework application.

ACUser to represent the users under access control.

ACRole to represent the roles under access control.

Consequently, instead of having the application's users and roles extending the DMAPL framework's interfaces, they should now extend the `ACUser` (AC stands for Access Control) and `ACRole` classes accordingly. In turn, the `ACUser` and `ACRole` classes implement the DMAPL framework's `AccessControlUser` and `AccessControlRole` interfaces. The complete access control hierarchy can be seen in Figure 3.1.

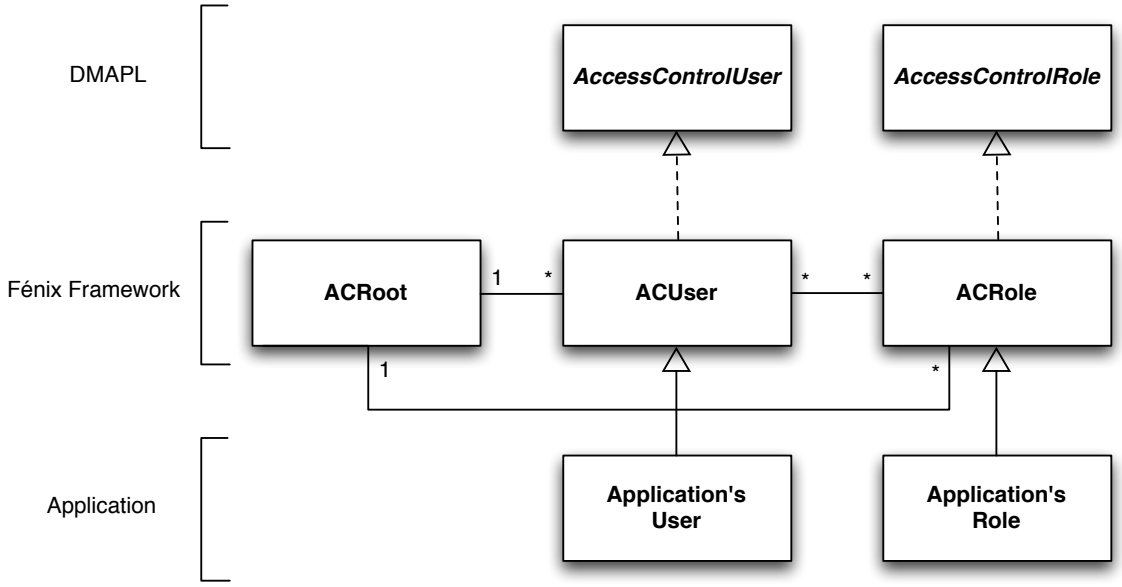

Figure 3.1: UML representation of the access control hierarchy.

This way, if the class representing the users of the application extends the `ACUser` class, the application's users automatically become subjects to the access control rules interpreted and enforced by the DMAPL framework. The same applies for the class representing the roles of the application when extending the `ACRole` class. As a result, it is extremely easy for the developer to place his users and roles under access control.

The interface of the `ACRoot` class can be seen in Listing 3.6. The implementation of the `ACUser` and `ACRole` classes can be seen in Listings 3.7 and 3.8 on page 33. Finally, the class of the plugin interface for the access control plugin is implemented in Listing 3.9 on page 34.

Listing 3.6: The class defining the `ACRoot` RDO.

```
1  public class ACRoot extends ACRoot_Base {
2
3   public static ACRoot getInstance();
4
5   public void initAccessControl(String dmaplSourcePath);
6
7   public static void beginAccessControl(ACUser user);
8
9   public static void endAccessControl();
10
11  public static ACUser getUserInSession();
12
13  public ACRole getRoleByName(String name);
14
15  public ACUser getUserByUserId(String id);
16
17 }
```

Listing 3.7: `ACUser` class.

```
1  package pt.ist.fenixframework.plugins.accessControl.domain;
2
3  import pt.ist.dmapl.AccessControlRole;
4  import pt.ist.dmapl.AccessControlUser;
5
6  public class ACUser extends ACUser_Base implements AccessControlUser {
7
8   public ACUser() {
9    super();
10   ACRoot.getInstance().addAcUsers(this);
11  }
12
13  public boolean hasRole(AccessControlRole role) {
14   for (ACRole myRole : getAcRoles()) {
15    if (myRole.getAcId().compareTo(((ACRole) role).getAcId()) == 0) {
16     return true;
17    }
18   }
19   return false;
20  }
21
22 }
```

Listing 3.8: `ACRole` class.

```
1  package pt.ist.fenixframework.plugins.accessControl.domain;
2
3  import pt.ist.dmapl.AccessControlRole;
4
5  public class ACRole extends ACRole_Base implements AccessControlRole {
6
7   public ACRole() {
8    super();
9    ACRoot.getInstance().addAcRoles(this);
10  }
11
12 }
```

Listing 3.9: Plugin interface for the access control plugin. The file argument `/accesscontrol-plugin.dml` on line 20 corresponds to the access control domain model introduced in Section 3.3.3.1.

```
1  package pt.ist.fenixframework.plugins;
2
3  import java.net.URL;
4  import java.util.Collections;
5  import java.util.List;
6
7  import pt.ist.fenixframework.FenixFrameworkPlugin;
8  import pt.ist.fenixframework.plugins.accessControl.domain.ACRoot;
9
10 public class AccessControlPlugin implements FenixFrameworkPlugin {
11
12  private String dmaplSourcePath;
13
14  public AccessControlPlugin(String dmaplSourcePath) {
15   this.dmaplSourcePath = dmaplSourcePath;
16  }
17
18  @Override
19  public List<URL> getDomainModel() {
20   URL resource = getClass().getResource("/accesscontrol-plugin.dml");
21   return Collections.singletonList(resource);
22  }
23
24  @Override
25  public void initialize() {
26   ACRoot.getInstance().initAccessControl(dmaplSourcePath);
27  }
28 }
```

3.3.3.2 Using the access control plugin

To configure and use the access control plugin is very easy. The initialization of the plugin is inserted in the initialization of the Fénix Framework itself, as seen on lines 8 to 10 in Listing 3.10 on page 35. The attribute `plugins` is an array of `pt.ist.fenixframework.FenixFrameworkPlugin` that is in the *classpath* of the application and will be used by the application. The file argument */myschool.dmapl* is the path to the file containing the access control rules to be interpreted and enforced by the DMAPL framework's runtime engine. This file can also be seen as the definition of the security policy to be enforced in the target application.

Listing 3.10: Initialization of the access control plugin.

```
1  Config config = new Config() {{
2   domainModelPath = "/myApplication.dml";
3   dbAlias = "//localhost:3306/mydb";
4   dbUsername= "myuser";
5   dbPassword = "";
6   rootClass = MyAppRoot.class;
7   updateRepositoryStructureIfNeeded = true;
8   plugins = new FenixFrameworkPlugin[] {
9    new AccessControlPlugin("/myschool.dmapl")
10  };
11 }};
12
13 FenixFramework.initialize(config);
```

From this point on, the application can interact with the plugin. However, to relate to its domain entities, we have to define them in the application. This is done by defining the plugin's entities as *external* classes in the application's domain model file. In the example of Listing 3.11 on page 36, the `ACUser` and ACRole classes (part of the access control domain model introduced in Section 3.3.3.1) are defined as *external* (lines 3 to 6). Next, the application's `Student` class is defined extending the `ACUser` class (line 8), and the application's `Role` class is defined extending the `ACRole` class (line 13). All the `Student` and `Role` objects are now subjects of the DMAPL framework's access control mechanisms.

Listing 3.11: Example of a domain model specification using the DML, and making use of the access control plugin. Both the `ACUser` and `ACRole` classes are declared as *external* (lines 3 to 6). The class `Student` extends `ACUser` (line 8), and the class `Role` extends `ACRole` (line 13).

```
1  external class .pt.ist.fenixframework.plugins
2          .accessControl.domain.ACUser as ACUser;
3  external class .pt.ist.fenixframework.plugins
4          .accessControl.domain.ACRole as ACRole;
5
6  class Student extends ACUser {
7   String id;
8  }
9
10 class Role extends ACRole {
11  String id;
12 }
13
14 relation StudentsHaveRoles {
15  Student playsRole students {
16   multiplicity *;
17  }
18  Role playsRole roles {
19   multiplicity *;
20  }
21 }
```

Finally, as stated in [5], the DMAPL framework is not responsible for the authentication of users. Instead, this must be done by any other application which must in turn, store the authenticated user in an access control session. An access control session is implemented by the DMAPL framework, using a thread-local variable to store the authenticated user. For each thread that stores a user in an access control session, there is an independently initialized copy of the variable. To store a user in an access control session, the method `beginAccessControl` in the `ACRoot` class must be invoked. Whenever the method `getUserInSession` from the `ACRoot` class is invoked, the corresponding user to the current thread is returned. To remove the user from an access control session, the method `endAccessControl` in the `ACRoot` class must be invoked, clearing the user's thread-local variable.

Now, with everything configured and initialized, the enforcing of the security policy can take place.

3.4 Conclusions

My contribution with this chapter was to have a *ready-to-use* version of the DMAPL framework. I began by covering what was needed to fix and develop in the DMAPL framework's runtime engine, to be able to use it and to test it with real RDM web applications. Having access to the information systems and applications of my University, it seemed like a good approach to use one of these applications to test the DMAPL framework since they are real, large and complex RDM web applications able to provide us with reliable feedback. However, most of the relevant applications run on top of the Fénix Framework, which created the necessity of integrating the DMAPL framework with this framework. I described how

this integration was achieved, including the creation of a specific access control domain model, and the adaptation to a plugin system creating the access control plugin. Finally, I described how to configure and use the access control plugin, and gave a simple example of how easily any application using the Fénix Framework can use the DMAPL framework's access control mechanisms through the access control plugin.

The result was an access control plugin with the DMAPL framework included, and fully integrated with the Fénix Framework. As benchmarks to measure the Fénix Framework's performance are available, this also seems like a very good solution to measure the plugin's performance in the future.

Chapter 4

Domain Model Relations

In Section 2.3.6 I described the three types of access control rules supported by the DMAPL framework's model. To remind them again, they are:

- Authorization Rules
- Amplification of Privileges
- Delegation of Rights

As mentioned earlier, RDM applications are based on the Domain-Driven Design (DDD) [7] development approach. DDD concentrates on the domain of a system and strongly supports that complex domains should be based on a model describing all its relevant entities, also known as domain entities. This model should also describe all the relations between the domain entities. In this chapter I will explain what is a domain model relation, introduce the concept of the relation rule and the motivation behind it, and finally discuss its implementation.

4.1 Defining Domain Model Relations

Consider an application using the Fénix Framework and the DML [1]. The DML is responsible for defining the domain entities, and the relations between them. Let us consider the example of a domain model shown in Listings 4.1 on page 40 and 4.2 on page 41:

Listing 4.1: Example of a domain model where six domain entities are defined: `MySchool`, `Officer`, `Teacher`, `Student`, `Role`, and `Course`.

```
external class .pt.ist.fenixframework.plugins
        .accessControl.domain.ACUser as ACUser;
external class .pt.ist.fenixframework.plugins
        .accessControl.domain.ACRole as ACRole;

class MySchool;

class Officer extends ACUser {
 String id;
 }

class Teacher extends ACUser {
 String id;
}

class Student extends ACUser {
 String id;
 Course course;
}

class Role extends ACRole {
 String id;
}

class Course {
 String name;
}
```

Listing 4.2: Definition of the relations between the domain entities defined in Listing 4.1.

```
1  relation MySchoolHasOfficers {
2   MySchool playsRole mySchool;
3   Officer playsRole officers { multiplicity *; }
4  }
5
6  relation MySchoolHasTeachers {
7   MySchool playsRole mySchool;
8   Teacher playsRole teachers { multiplicity *; }
9  }
10
11 relation MySchoolHasStudents {
12  MySchool playsRole mySchool;
13  Student playsRole students { multiplicity *; }
14 }
15
16 relation MySchoolHasRoles {
17  MySchool playsRole mySchool;
18  Role playsRole roles { multiplicity *; }
19 }
20
21 relation MySchoolHasCourses {
22  MySchool playsRole mySchool;
23  Course playsRole courses { multiplicity *; }
24 }
25
26 relation TeachersHaveCourses {
27  Teacher playsRole teachers { multiplicity *; }
28  Course playsRole courses { multiplicity *; }
29 }
30
31 relation CoursesHaveStudents {
32  Course playsRole courses { multiplicity *; }
33  Student playsRole students { multiplicity *; }
34 }
35
36 relation OfficersHaveRoles {
37  Officer playsRole officers { multiplicity *; }
38  Role playsRole roles { multiplicity *; }
39 }
```

In this example, six domain entities are defined: `MySchool` (the RDO), `Officer`, `Teacher`, `Student`, `Role`, and `Course`. The relations between these entities are also defined. For example, in Listing 4.2 on page 41 from lines 6 to 9, the relation between `MySchool` (the RDO) and `Teacher` is defined (relation `MySchoolHasTeachers`), where `Teacher` has a multiplicity qualifier of *many* (represented by '*', see the UML diagram in Figure 4.1). The goal of this relation is to model that a single school has several teachers. However, a developer should be able to define who in the system can assign new teachers to the school. Another example can be seen, again in Listing 4.2 on page 41, from lines 31 to 34. In this relation, the goal is to model the fact that a course can have several students, and a student can have several courses (relation `CoursesHaveStudents`). Both `Course` and `Student` have multiplicity qualifiers of *many* (see the UML diagram in Figure 4.2). Again, one should be able to define who can enroll students in courses.

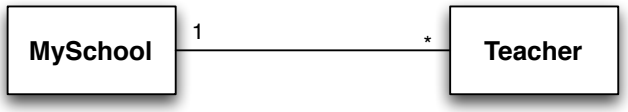

Figure 4.1: UML representation of the relation `MySchoolHasTeachers`.

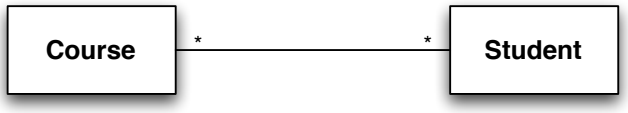

Figure 4.2: UML representation of the relation `CoursesHaveStudents`.

4.2 Motivation

Considering the two examples described above regarding the relations `MySchoolHasTeachers` and `CoursesHaveStudents` (specified in Listing 4.2 on page 41), using the DMAPL framework, how can we express who can assign new teachers to the school? Or who can enroll students in courses? The granularity level of the DMAPL framework's access control mechanisms is the Java interface method, and although methods to handle the relations are automatically generated by the Fénix Framework, one relation can still be modified through different methods in different points of the code. This makes it difficult to control the modifications of one specific relation. Considering all the emphasis of the DDD development approach in the RDM application domain model, and being the DMAPL framework oriented towards the problems and challenges faced when trying to enforce access control policies in RDM applications, it would be desirable for the framework to support access control, not only at the Java method level, but also at the domain level. To enable the DMAPL framework to handle access control at the domain model, the relation rule was created.

4.3 Relation Rule

The creation of the relation rule is motivated by the desired possibility of enforcing access control at the domain level. With it, it is possible to specify who can alter a relation between two domain entities, and

when. For example, for the domain model presented above, it would make sense to allow the `Officer` to assign students to courses.

4.3.1 Specifying relation rules using the DMAPL framework

The syntax of a relation rule tries to follow the same construction of the other types of rules supported by the DMAPL framework's model. The exact syntax can be seen in Listing 4.3. One important aspect about the relation rules, is that they are **always positive**. This leads to a scenario of a *closed policy*. Whenever the addition or removal of a relation between two entities is attempted and there is no corresponding relation rule allowing it, the modification will be denied by the DMAPL framework.

A relation rule is composed by a:

Subject that refers to whom the rule applies. Can be either a user or a role.

Change type to qualify the type of change allowed in the relation. It may be the value `add-to` when the rule applies only to the addition of relations between entities, `remove-from` when the rule applies only to the removal of relations between entities, and `change` when the rule applies to both addition and removal of relations between entities.

Target relation which is the fully qualified name of the target relation of the rule.

Optionally, a relation rule may have a constraint introduced with the keyword `where`. Simply put, a constraint is a Java expression between brackets that evaluates to the boolean values `true` or `false`. Both the `RuleName`, the `Subject` and the `Constraint` follow the same syntax and grammar rules as specified in [5].

Listing 4.3: DMAPL's syntax for relation rules.

```
RelationRule
        RuleName^opt  RelationRuleBody

RelationRuleBody
         allow Subject to ChangeType relation TargetRelationList
         Constraint^opt

ChangeType
        add–to
        remove–from
        change

TargetRelationList
         TargetRelation
         TargetRelationList , TargetRelation

TargetRelation
        QualifedName . Identifier
```

4.3.2 Examples

To make the introduced syntax and semantic more clear, I present below two examples.

Once again, let us consider the example in Listings 4.1 on page 40 and 4.2 on page 41, and imagine that there is a role named `Management` in the application. It makes sense to allow the role `Management` to assign new teachers to the school. With a relation rule, this exact requirement is satisfied by simply allowing the role `Management` to add new elements to the relation `MySchoolHasTeachers`, as depicted in Listing 4.4. Notice the use of the keyword `add-to`, allowing only to add new teachers, and not removing existing ones. For that, the keyword `change` would have to be used instead.

Users assigned with the role `Management` should also be able to add and remove students from courses. With the relation rules, this is again easily satisfied. We just need to write another rule allowing the role in question to modify the relation `CoursesHaveStudents`, as it can be seen in Listing 4.5. However, it makes sense to allow such actions only when inside the school's enrollment period. This restriction is enforced with the introduction of a constraint with the predicate `isEnrollmentPeriod`. One good candidate to have the role `Management` assigned would be a user of type `Officer`.

Listing 4.4: Relation rule specifying that the role `Management` can add new elements to the relation `MySchoolHasTeachers`.

```
1 TeacherSchoolAssignment:
2  allow role Management
3  to add-to relation MySchoolHasTeachers
```

Listing 4.5: Relation rule specifying that the role `Management` can add and remove elements from the relation `CoursesHaveStudents`, when inside the enrollment period.

```
1 CourseStudentAssignment:
2  allow role Management
3  to change relation CoursesHaveStudents
4  where { MySchool.isEnrollmentPeriod() }
```

4.4 Implementing and Enforcing Relation Rules

Similarly to what happens with the other types of rules, the enforcement of the relation rules is done by the runtime engine. Thus, a runtime representation is needed to be handled by the engine. In this section, analogous to what was done for the other types of rules in [5], I will introduce the runtime representation of the relation rules.

4.4.1 Runtime model

A relation rule contains at least one `Subject`, one or more `Targets`, and can have at most one `Constraint`. These components are depicted in the class diagram in Figure 4.3 on page 45.

4.4.1.1 Subject and Constraint

Both the `Subject` and the `Constraint` are common to all the other types of rules, having the exact same runtime representation as defined in [5].

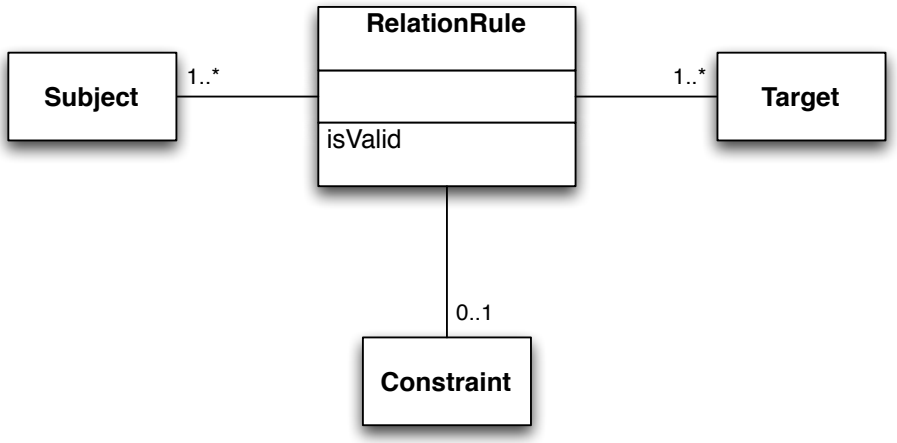

Figure 4.3: Class diagram of a relation rule.

4.4.1.2 Relation Target

In [5], Dumiense stated that a `Target` could be of two types: `MethodTarget` and `AnnotationTarget`. With the introduction of the relation rule, a `Target` can be of a third type: `RelationTarget`. The `RelationTarget` identifies univocally a relation defined in the domain model, by holding its fully qualified name. To verify if a modified relation corresponds to the target, a `matches` predicate exists to compare the fully qualified names of both relations. The complete class diagram of the `Target` component can be seen in Figure 4.4.

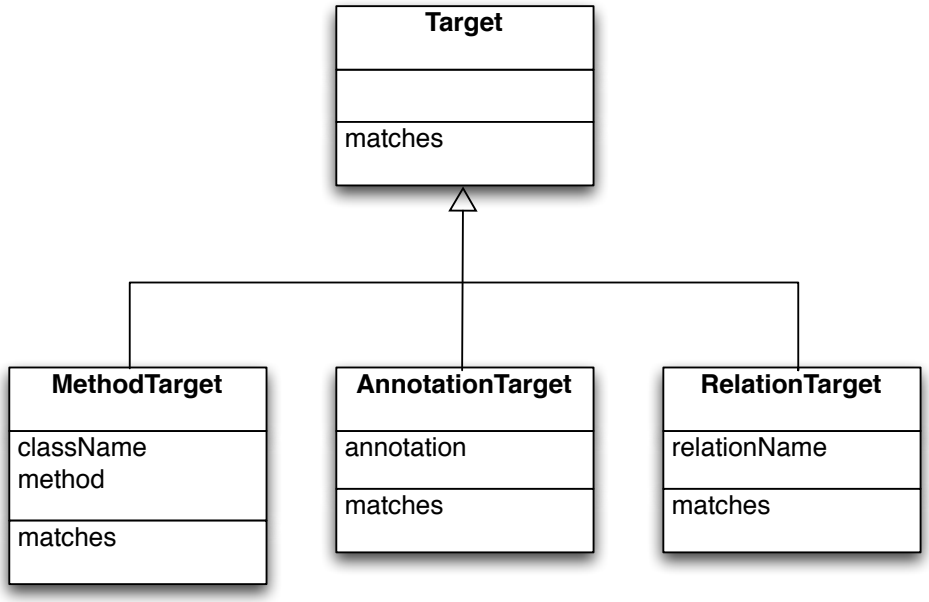

Figure 4.4: Class diagram of the `Target`, including the new `RelationTarget`.

4.4.1.3 Rule matching

A relation rule is valid if, and only if, the user and the modified relation match the subject and the target of the rule. In case there is also a constraint defined, it must evaluate to true. Listing 4.6 contains the implementation of the operation `isValid`, which is responsible for implementing this exact behavior.

Listing 4.6: Implementation of the operation `isValid` for the relation rule.

```
1  public boolean isValid(AccessControlUser user, String relation,
2      Object first, Object second) throws ConstraintEvaluationException {
3   if (this.matches(user, relation)) {
4    if (this.constraint != null) {
5     List<Parameter> constraintVars = new ArrayList<Parameter>();
6     constraintVars.add(new Parameter("user", user));
7     constraintVars.add(new Parameter(first.getClass().getSimpleName()
8                         .toLowerCase(), first));
9     constraintVars.add(new Parameter(second.getClass().getSimpleName()
10                        .toLowerCase(), second));
11    return this.constraint.eval(constraintVars);
12   } else {
13    return true;
14   }
15  }
16  return false;
17 }
```

4.4.2 Enforcing Relation Rules Decisions

After introducing all the components of the relation rules, I will now explain how they are used in the enforcement of a security policy upon the application executing together with the Fénix Framework.

4.4.2.1 Access Control Listener

During the execution of an application, and whenever a domain relation is modified, the DMAPL framework's runtime engine has to be invoked to verify if the modification to the domain relation is allowed. The Fénix Framework is a transactional system that provides mechanisms to inspect each transaction. Moreover, it enables the developer to know, at the end of each transaction, which domain relations were modified. This is extremely useful to the runtime engine, since it can discover during runtime, and at the end of each transaction, which domain relations were modified. The Fénix Framework allows the developer to associate *listeners* to its execution, having this way pieces of code set to be invoked at a specific point of the execution of the application. A `CommitListener` can be invoked *before*, or *after* a transaction's *commit*, being the *before* option perfectly suitable for the DMAPL framework (see Listing 4.7 on page 47). As a result, I created an *AccessControlListener* with the purpose of invoking the DMAPL framework's runtime engine before the *commit* of each transaction. The implementation of the *AccessControlListener* can be seen in Listing 4.8 on page 47.

To use the *AccessControlListener*, we simply have to add it to the Fénix Framework after its initialization (see Listing 4.9 on page 47). Once this is done, just before any transaction's *commit*, the *AccessControlListener* is invoked. The *AccessControlListener* will then, for each modified relation, invoke the operation `checkRelationsAccess` in the DMAPL framework's runtime engine (line 7 in Listing 4.8 on page 47). In this operation, the runtime engine will try to match the modified relation with any of the existing relation rules. If there is a match, the execution is normally returned to the Fénix Framework

Listing 4.7: Interface for the `CommitListener`.

```
1 public interface CommitListener {
2  public void beforeCommit(TopLevelTransaction tx);
3  public void afterCommit(TopLevelTransaction tx);
4 }
```

Listing 4.8: Implementation of the *AccessControlListener*.

```
1  public class AccessControlListener implements CommitListener {
2
3   public void beforeCommit(TopLevelTransaction tx)
4               throws AccessControlException {
5    if (tx.isWriteTransaction()) {
6     for (RelationChangelog relationChangelog
7                   : tx.getRelationsChangelog()) {
8      AccessControlEngine.checkRelationsAccess(
9        relationChangelog.relation,
10       relationChangelog.remove, relationChangelog.first,
11       relationChangelog.second);
12    }
13   }
14  }
15 }
```

that will try to commit the transaction. If there is no match, an access control exception is thrown (line 33 in Listing 4.10 on page 48) and the transaction is aborted. The implementation of the operation `checkRelationsAccess` can be seen in Listing 4.10 on page 48.

Listing 4.9: Initialization of the *AccessControlListener*.

```
1 ...
2  FenixFramework.initialize(config);
3  TopLevelTransaction.addCommitListener(new AccessControlListener());
4 ...
```

Listing 4.10: Implementation of the operation `checkRelationsAccess` in the DMAPL framework's runtime engine.

```
1  public static void checkRelationsAccess(String relation, boolean remove,
2          Object first, Object second) throws AccessControlException {
3   AccessControlUser user = AccessControlSession.getUser();
4   if (user == null) {
5    return;
6   }
7   try {
8    for (RelationRule rule : PolicyContainer.getPolicyContainer()
9                            .getChangeRelationPolicy()) {
10    if (rule.isValid(user, relation, first, second)) {
11     return;
12    }
13   }
14   if (remove) {
15    for (RelationRule rule : PolicyContainer.getPolicyContainer()
16                                 .getRemoveRelationPolicy()) {
17     if (rule.isValid(user, relation, first, second)) {
18      return;
19     }
20    }
21   } else {
22       for (RelationRule rule : PolicyContainer.getPolicyContainer()
23                                 .getAddRelationPolicy()) {
24        if (rule.isValid(user, relation, first, second)) {
25         return;
26        }
27       }
28   }
29  } catch (Exception ex) {
30     throw new AccessControlException(ex.getMessage());
31  }
32
33  throw new AccessControlException("No rule found that allowed it.");
34 }
```

4.5 Conclusions

I began this chapter by explaining what a domain model relation was, and the motivation to enhance the DMAPL framework with a mechanism to provide access control at the domain level in RDM applications. This led me to introduce a new type of rule, the relation rule, that satisfies this exact requirement. The relation rule is always a positive rule, leading to a scenario of a *closed policy*. A complete definition of the relation rule was given, supported by two practical examples. Later I discussed the implementation of the relation rule and its runtime model, including the creation of a new type of rule `Target`, the `RelationTarget`. Finally, I explained how this new type of rules is enforced by the DMAPL framework's runtime engine on the application. This enforcement is mainly supported by an *AccessControlListener*, that, once added to the Fénix Framework, invokes the runtime engine before each transaction's *commit*, and for each modified relation.

My main contribution with this chapter was to have a new type of rule, the relation rule, *ready-to-use* with the DMAPL framework. In my work, the implementation of this rule led to the enforcing of a *closed policy* by DMAPL framework's runtime engine (where the default action is to deny access). However, this could easily have been done differently, and enforce an *open policy* instead (where the default action is to allow access). Although *closed policies* are usually safer, and *open policies* are usually more suitable for collaborative work, in the particular case of RDM applications, I feel this is an open issue that should be further investigated to try to assert if there is a more suitable policy. The performance of the DMAPL framework's runtime engine, when enforcing relation rules on the application, should also be evaluated. Invoking the runtime engine before every transaction's *commit*, and for each modified relation, may lead to an unwanted overhead over the application's execution when trying to enforce access control. Also, until a transaction is intercepted by the *AccessControlListener* just before the *commit*, all the modifications to the domain relations are performed normally without any control. Developers should bear this in mind when implementing code dependent of the domain relations, since there is a risk of their modifications to be denied later when the transaction is evaluated by the DMAPL framework's runtime engine.

Chapter 5

Validation of the DMAPL framework

One goal of my work was to use and test the DMAPL framework in a real RDM application. Doing so, embodied an important step of my work, since no practical validation of the DMAPL framework had been made so far. It is important to test new software in a real application to receive reliable feedback, assess the software's real performance, and discover unforeseen problems. Also, further development should be guided by previous experiences of testing the software in real environments.

As previously mentioned, my University[1] has its own information systems and applications, some of them RDM web applications, that give all the administrative and educational support required for a University domain. Therefore, to test the DMAPL framework, I decided to use one of these applications called FeaRS[2] (Feature Request System). To do so, an integration of the DMAPL framework with the Fénix Framework was needed, because the FeaRS, as well as several other applications, runs on top of the Fénix Framework. After having the DMAPL framework's runtime engine *ready-to-use* (see Section 3.2), this integration was accomplished in Section 3.3, making the DMAPL framework ready to be used in the FeaRS.

In this chapter I will describe what is the FeaRS, and its domain model. Then, I introduce the FeaRS access control requirements, and how they are satisfied by the DMAPL framework. The FeaRS access control requirements will be used to exercise three different ways of expressing access control requirements. Finally, I close the chapter by comparing and discussing the results obtained.

5.1 The FeaRS

The FeaRS (Feature Request System) is an RDM web application developed to manage suggestions of features for several systems. Its purpose is to gather suggestions made by users, and through a voting system, rank those suggestions. Users can create new suggestions and vote on existing ones. These suggestions may be requests for new features, or for the modification of existing ones. Through the number of votes of the suggestions, developers can have a clear perspective of which are the most requested features, and decide which ones to implement first. The ultimate goal of the FeaRS is to improve the services provided by a system. The FeaRS should not be mistaken for a bug tracking system, or a place to report errors.

5.1.1 The FeaRS domain model

To better understand the FeaRS domain model, the Figure 5.1 on page 52 shows its UML representation.

[1] Instituto Superior Técnico - `http://www.ist.utl.pt`
[2] `http://fears.ist.utl.pt`

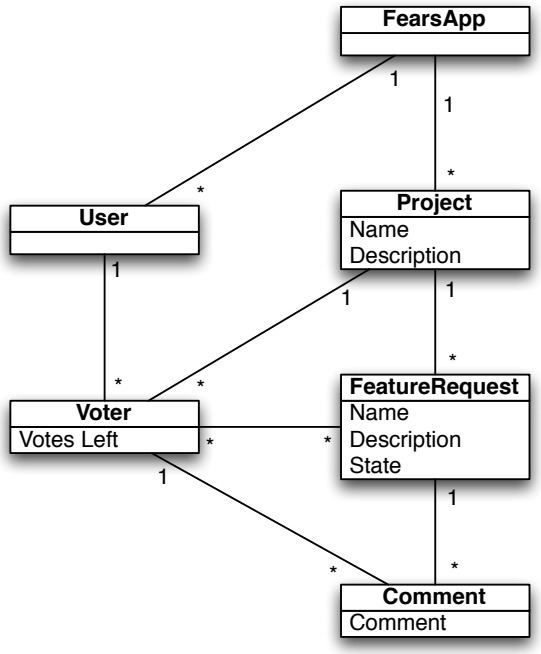

Figure 5.1: UML representation of the FeaRS domain model.

5.2 Using the DMAPL framework in the FeaRS

I decided to use the FeaRS to test the DMAPL framework due not only to its implementation and use in a real environment, but also because it demonstrated to have an accessible complexity both in the domain model and in the code.

5.2.1 Configuring the DMAPL framework

To use the DMAPL framework in the FeaRS, the access control plugin has to be configured. Configuring the access control plugin is quite simple, and it can be summarized in three steps (see Section 3.3.3.2):

1. Initialization of the access control plugin (as seen on lines 8 to 10 in Listing 3.10 on page 35).
2. Definition of the plugins entities (part of the access control domain model) as external classes in the applications domain model file (as seen on lines 3 to 6 in Listing 3.11 on page 36).
3. Adjustment of the authentication mechanism to invoke the `beginAccessControl` and `endAccessControl` methods accordingly.

After configuring the access control plugin, the FeaRS is ready to use the DMAPL framework. However, when initial tests were performed, one error kept occurring of having the wrong user in the access control session. In Section 3.3.3.2, the concept of access control session implemented by the DMAPL framework was explained. The access control session assumes that for each user logged in in the system, there is only one thread to handle that user. As a result, an access control session has a thread-local variable to store the user logged in (each thread will have an independently initialized copy of the variable). Nevertheless, when dealing with web applications this specific scenario may change. In the particular case of the FeaRS web application, the web server used is the Apache Tomcat.[3] Tomcat is an open source

[3] `http://tomcat.apache.org`

servlet container that reuses threads to answer several requests of the same client. This means there is not just one thread for a single client, causing the access control session to loose its effectiveness. To workaround this problem a specific filter was implemented. As a result, before responding to a request, each thread's thread-local variable is set with the correct user obtained from the HTTP[4] session.

5.2.2 The FeaRS access control requirements

The FeaRS has the following access control requirements:

- Only registered users can vote, add features, and add comments.
- A registered user can remove votes, but only his own votes.
- Only administrators can create, edit, and delete projects.
- Only administrators can see the current list of administrators.
- Only administrators can grant a registered user the role of administrator, and revoke it (except for themselves).
- Only administrators can assign a registered user as a project administrator, and remove it (except for themselves).
- Only administrators and project administrators can change the project's features state.
- Seeing existing projects and features requests is public.

Before using the DMAPL framework, the enforcing of these requirements was being done through the continuous invocation of two methods: `isLoggedIn` and `isAdmin`. Before any action would be performed, an invocation to one of these two methods would be made inquiring if the user in session was logged in, or if it was an administrator (having to previously be logged in). From this simple analysis, the necessary roles to use with the DMAPL framework can automatically be extracted: the role *LoggedIn*; and the role *Admin*. The roles are hierarchical, with an `Admin` being also `LoggedIn`.

After better understanding the FeaRS and its access control requirements, I replaced its access control mechanisms with the DMAPL framework's access control mechanisms. I then tried to express the FeaRS access control requirements using the access control rules provided by the DMAPL framework. As a result, three exercises were made. With each exercise I tried to express the FeaRS access control requirements with different access control mechanisms of the DMAPL framework. The approach taken in each exercise was:

1. Using only authorization rules, tickets, and amplification of privileges.
2. Using only authorization rules, tickets, and amplification of privileges, but with annotations.
3. Using only domain model relations (relation rules).

I will now describe each of these exercises.

[4]Hypertext Transfer Protocol - Commonly used in the communication between web applications and their clients.

5.2.2.1 Exercise 1: Authorization rules, tickets, and amplification of privileges

In the first exercise I tried to express the FeaRS access control requirements using only authorization rules, tickets, and amplification of privileges. This meant adding access control to 18 methods, which resulted in an access control policy with 17 authorization rules, one ticket, and one amplification of privileges. An example of one authorization rule used is depicted in Listing 5.1. The ticket and the amplification of privileges were used to express that only project administrators can change the project features state. This required a small modification in the original FeaRS method to be access controlled. Both the ticket and the amplification of privileges can be seen in Listings 5.2 and 5.3. Using both the authorization rules, tickets, and amplification of privileges, provided great expressiveness to specify access control rules. As a result, all of the FeaRS access control requirements were successfully covered by the DMAPL framework's access control mechanisms.

The complete access control policy can be seen in Section A.1 on page 61.

Listing 5.1: Authorization rule that states that only registered users (assigned with the corresponding role `LoggedIn`) can vote. (exercise 1)

```
1 GiveVoteAccess:
2  allow role LoggedIn
3  to eu.ist.fears.server.FearsServiceImpl
4   .vote(String projectID, String name, String sessionID)
```

Listing 5.2: Ticket that allows project administrators to change the project features state. (exercise 1)

```
1 ticket ChangeFeatureStateTicket(FearsServiceImpl fears)
2  to eu.ist.fears.server.FearsServiceImpl
3   .changeFeatureState(Project p, FeatureRequest feature, State newState)
4  where { p.isProjectAdmin(fears.getUserFromSession()); }
```

Listing 5.3: Amplification of privileges that gives to registered users the ticket `ChangeFeatureStateTicket`. (exercise 1)

```
1 GiveChangeFeatureStateTicket:
2  on eu.ist.fears.server.FearsServiceImpl
3   .addComment(String projectID, String featureName, String comment,
4               State newState, String sessionID)
5  give role LoggedIn ticket ChangeFeatureStateTicket(receiver)
```

5.2.2.2 Exercise 2: Annotations

In the second exercise I used the same authorization rules, tickets, and amplification of privileges, together with annotations. This means that instead of having access control rules with `MethodTargets`, I assigned the FeaRS application's methods, to be access controlled, with access control annotations and used those annotations as rule targets (`AnnotationTargets`, see Figure 4.4 on page 45).

I began to define three annotations with semantics associated according to the FeaRS access control requirements (see Listing 5.4 on page 55 for the definition of one annotation):

`@UserManagement` for all the user management operations, such as adding an administrator to the system.

`@ProjectManagement` for all the project management operations, such as editing a project.

`@UserFeatureInteraction` for all the operations responsible for the interaction between users and features, such as casting a vote.

Having well defined semantics associated with the access control annotations, helps developers decide which annotations to use in which methods and access control rules. However, not all FeaRS access control requirements were satisfied with annotations. For example, methods with the same annotation but with different access control constraints, are targets of one single authorization rule where we can only specify one constraint. This suggests that annotations are hard to use with constraints. Also, it was not possible to use annotations with the ticket and the amplification of privileges. Similar to methods with different access control constraints, tickets and amplification of privileges are usually more specific cases of access control. This is opposite to the annotations purpose, which is to be used in more generic cases. Finally, the method `logoff` was also required to be access controlled, but did not fit in any of the three defined annotations.

Using annotations resulted in an access control policy with six authorization rules (three of them using the access control annotations), one ticket, and one amplification of privileges. An example of one authorization rule, with an `AnnotationTarget`, used in this exercise is depicted in Listing 5.5. The complete access control policy can be seen in Section A.2 on page 64.

Listing 5.4: Definition of the `@UserManagement` annotation. (exercise 2)

```
1 package eu.ist.fears.server.domain.annotations;
2
3 @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
4 public @interface UserManagement {
5
6 }
```

Listing 5.5: Authorization rule with an `AnnotationTarget`. This rule states that administrators (users with the role `Admin`) have access to the methods annotated with the `@UserManagement` annotation. (exercise 2)

```
1 GiveUserManagementAccess:
2  allow role Admin
3  to @eu.ist.fears.server.domain.annotations.UserManagement
```

5.2.2.3 Exercise 3: Domain Model Relations

In the third and last exercise, I tried to express the FeaRS access control requirements using only relation rules. Using relation rules translates into enforcing the access control requirements at the domain level, and thus, using the domain model relations (see Chapter 4 on page 39). In the FeaRS domain model there are 12 relations. However, access control requirements that did not affect any domain relation were not possible to express using relation rules. The semantics of tickets and amplification of privileges was also not suitable for relation rules. In [5], the constraints introduced to be used with the access control rules, can include the arguments passed on to the access controlled methods giving these constraints greater expressiveness. When using relation rules, this expressiveness is somewhat lost, due to the lack of arguments in the `RelationTargets` (an example can be seen in line 3 of Listing 5.6 on page 56).

This exercise resulted in an access control policy with 21 rules (16 relation rules and five authorization rules), one ticket, and one amplification of privileges. An example of one relation rule used in this exercise can be seen in Listing 5.6, and the complete access control policy can be seen in Section A.3 on page 65.

Listing 5.6: Relation rule that allows a registered user (users with the role `LoggedIn`) to modify the domain relation `FeatureRequestVoters`. In other words, it allows the registered user to vote in a feature. (exercise 3)

```
1 GiveChangeFeatureRequestVotersAccess:
2  allow role LoggedIn
3  to change relation eu.ist.fears.server.domain.FeatureRequestVoters
```

5.2.2.4 Impact on access control policy change

After having done three exercises to express the FeaRS access control requirements with different approaches, I tried to assess what kind of impact would a change in the access control policy have in each of these approaches. To do so, I introduced a new functionality to the FeaRS, and tried to express its access control requirement with each approach. The new functionality was the possibility of users to remove features, with the access control requirement to only allow administrators to remove features.

I will now describe, for each approach, what were the necessary modifications in the access control policy to satisfy the new access control requirement.

Authorization rules, tickets, and amplification of privileges (exercise 1)**:** one authorization rule had to be added (see Listing 5.7).

Annotations (exercise 2)**:** the annotation `@ProjectManagement` had to be added to the method `removeFeature` (no access control rule was added).

Domain model relations (exercise 3)**:** a feature always has a project associated, and may or may not have votes and comments. To remove the feature from the associated project, only one relation rule had to be added (see Listing 5.8 on page 57). However, to be able to remove also the feature's votes and comments, two more rules had to be added (see Listings 5.9 on page 57 and 5.10 on page 57).

Listing 5.7: Authorization rule that allows administrators (users with the role `Admin`) to remove features. (exercise 1)

```
1 GiveRemoveFeatureAccess:
2  allow role Admin
3  to eu.ist.fears.server.FearsServiceImpl
4     .removeFeature(String featureID, String sessionID)
```

5.2.2.5 Comparison

After describing all the exercises made with the DMAPL framework and the FeaRS, a comparison was made between each approach.

Next I present the advantages and disadvantages encountered when doing the previously described exercises.

Exercise 1: Authorization rules, tickets, and amplification of privileges

Listing 5.8: Relation rule that allows administrators (users with the role `Admin`) to remove objects from the relation `ProjectFeatureRequests`. In other words, it allows administrators to remove features from projects. (exercise 3)

```
1 GiveRemoveFeatureFromProjectAccess:
2  allow role Admin
3  to remove-from relation
4     eu.ist.fears.server.domain.ProjectFeatureRequests
```

Listing 5.9: Relation rule that allows administrators (users with the role `Admin`) to remove objects from the relation `FeatureRequestVoters`. In other words, it allows administrators to remove votes from features. (exercise 3)

```
1 GiveRemoveVoteFromFeatureAccess:
2  allow role Admin
3  to remove-from relation eu.ist.fears.server.domain.FeatureRequestVoters
```

Advantages:

- great expressiveness to specify access control requirements
- small impact on access control policy change

Disadvantages:

- implementation of the ticket and the amplification of privileges required a small modification to the original method

Exercise 2: Annotations

Advantages:

- very small impact on access control policy change
- access control policy file more readable

Disadvantages:

- hard to use with constraints
- not suitable for tickets and amplification of privileges

Exercise 3: Domain Model Relations

Advantages:

Listing 5.10: Relation rule allows administrators (users with the role `Admin`) to remove objects from the relation `FeatureRequestComments`. In other words, it allows administrators to remove comments from features. (exercise 3)

```
1 GiveRemoveCommentFromFeatureAccess:
2  allow role Admin
3  to remove-from relation
4     eu.ist.fears.server.domain.FeatureRequestComments
```

- no in-depth knowledge of the code required
- more fine-grained access control policy

Disadvantages:

- lost of expressiveness in the constraints
- not suitable for tickets and amplification of privileges

5.3 Conclusions

In this chapter I have successfully validated the DMAPL framework. I began by introducing The FeaRS (Feature Request System), and configuring the DMAPL framework in it. After extracting the FeaRS access control requirements, the FeaRS access control mechanisms were replaced with the DMAPL framework's access control mechanisms. With the FeaRS ready to use the DMAPL framework, three exercises were done to test and compare the several access control mechanisms of the DMAPL framework.

When expressing the access control policy using authorization rules, tickets, and amplification of privileges (exercise 1), a developer should know all the entry points (methods) to the protected data/behavior, specially, if in the presence of a service-oriented policy like the FeaRS has. This approach revealed to provide great expressiveness for access control rules, successfully covering all of the FeaRS access control requirements.

With the use of annotations (exercise 2), the developer can assign annotations regardless of the access control policy, relying solely on their semantics. But although annotations provide a more readable access control policy, they are not suitable for the semantics associated with tickets and amplification of privileges, and are hard to use when several constraints exist.

To express an access control policy using the relation rules (exercise 3), instead of an in-depth knowledge of the code, a good knowledge over the application's domain model is required. One simple action in the application may translate into the modification of several domain relations, which also causes a bigger impact on the access control policy. However, access control requirements that do not affect domain relations are not possible to express with relation rules. Semantics associated with tickets and amplification of privileges are not suitable, and there is also the loss of some expressiveness in the constraints. The success of the relation rules is tightly coupled with the complexity of the application's domain model, allowing or not for a more fine-grained access control policy.

Still regarding the domain model relations, in Chapter 4, I mentioned that the relation rules were implemented enforcing a *closed policy*. That is, if one access control requirement affects several domain relations, even if not directly, all of the affected domain relations will have to be covered by the access control policy. If otherwise, in the lack of rules the DAMPL framework's runtime engine denies the access. Having a *closed policy* made the use of the relation rules in exercise 3 harder. On the other hand, this can also suggest that maybe an *open policy* would be more appropriate with domain level access control. Further testing should be made, possibly with a more complex domain model, to try to reach a consensus.

In conclusion, having tested the DMAPL framework was a very positive step, being always able to fulfill all the access control requirements. Moreover, further use cases of validation of the DMAPL framework should be made, and also testing to the performance of the runtime engine. Knowing how the access control mechanisms perform would be a very good indication of further development in the framework.

Chapter 6

Conclusions

There is no doubt that nowadays RDM web applications present a serious challenge when it comes to security and access control policy specification and management. After discussing some related work, the solution and access control mechanisms described by Dumiense [5] present the most flexible and adapted framework to deal with access control policies in RDM web applications. My research and work was then guided to try to improve the solution introduced and described in [5], the DMAPL framework.

I close this dissertation with the present chapter, where I describe my main contributions, and introduce some ideas for future work.

6.1 Main Contributions

A very important step missing in the validation of the DMAPL framework was an implementation in a real system to be able to get reliable feedback about its performance, discover unforeseen problems, and guide its development towards the next logical steps. Having access to the information systems and applications of my University, I decided to use one of these applications to test the DMAPL framework since they are real, large and complex RDM web applications able to provide us with reliable feedback. A good candidate to test the framework was the FeaRS (Feature Request System), because it demonstrated to have an accessible complexity both in the domain model and in the code. The FeaRS is a real RDM web application developed to manage suggestions of features for several systems. Its goal is to improve the services provided by a system. The FeaRS has been fully implemented and used in a real environment for some time now, being able to provide us reliable feedback with real data. Working towards the goal of implementing the DMAPL framework in the FeaRS, my work ended up in contributing for the DMAPL framework in several ways. Specifically:

- By covering what was needed to fix and develop in the DMAPL framework's runtime engine, since the DMAPL framework was still not ready to be used as it could be.
- The integration of the DMAPL framework with the Fénix Framework. Most of the relevant applications to the test the DMAPL framework, including the FeaRS, run on top of the Fénix Framework. To be able to use and test the DMAPL framework with any of these applications, an integration with the Fénix Framework was needed. An access control plugin was created providing an easy for the developers to use the DMAPL framework's access control mechanisms in any application using the Fénix Framework.
- By introducing a new type of access control rule: the relation rule. The creation of this new type of rule was motivated by all the emphasis given to the domain model of an RDM application by the

DDD development approach. With the relation rules, developers are now able to express domain level access control within the DMAPL framework.

- The validation of the DMAPL framework with an implementation in a real RDM web application, namely, the FeaRS. Four exercises were done testing the several DMAPL framework's access control mechanisms, where all the FeaRS access control requirements were successfully satisfied.

Also, regarding the RDM web application FénixEdu analyzed in Section 2.4 on page 24, five distinct mechanisms of access control in five different points of the system were enumerated along with their advantages and disadvantages. Considering this, the DMAPL framework is capable of covering the access control mechanisms in the objects and services, while having a centralized approach and without any code scattering and tangling.

6.2 Future Work

The access control area is a very broad subject, and there are several possible directions to take when it comes to further develop and test the DMAPL framework. However, I tried to enumerate the ones I think are the most important, and that will make the DMAPL framework a more suitable framework to express and enforce access control policies in RDM applications.

- Regarding the implementation of the relation rules, it would be very interesting to test their usage with the runtime engine enforcing an *open policy*, instead of a *closed policy*, and try to assert if there is a more suitable policy when expressing access control for RDM applications.

- The performance of the DMAPL framework's runtime engine should also be evaluated. For example, in the case of the relation rules, invoking the runtime engine at end of every transaction, and for each modified relation, may lead to an unwanted overhead over the application's execution when trying to enforce access control. Moreover, since benchmarks to measure the Fénix Framework's performance are available, they may also be used to test the DMAPL framework's runtime engine performance.

- Although Dumiense introduced and described a delegation model in [5], its implementation in the DMAPL framework's runtime engine was never done. This is a feature worth pursuing for, since it is highly desirable when expressing access control policies.

- Currently, when the access control policy in the DMAPL framework changes, the application has to be restarted for the new policy to become effective. A very interesting feature would be to be able to change the access control policy during runtime, and make it effective without stoping the application.

- Further use cases of validation of the DMAPL framework should also be made. The FeaRS has a relatively small domain model with little complexity. A bigger and more complex RDM application, such as the FénixEdu, should be used to further test the DMAPL framework, and see what the usage experience tells about the framework.

Appendix A

FeaRS access control policies

A.1 Exercise 1: Authorization rules, tickets, and amplification of privileges

Listing A.1: The complete access control policy of the FeaRS using authorization rules, tickets, and amplification of privileges.

```
1  GiveVoteAccess:
2   allow role LoggedIn
3   to eu.ist.fears.server.FearsServiceImpl.vote(String projectID,
4      String name, String sessionID)
5
6  GiveAddFeatureAccess:
7   allow role LoggedIn
8   to eu.ist.fears.server.FearsServiceImpl.addFeature(String projectID,
9      String name, String description, String sessionID)
10
11 GiveAddCommentAccess:
12  allow role LoggedIn
13  to eu.ist.fears.server.FearsServiceImpl.addComment(String projectID,
14     String featureName, String comment, State newState, String sessionID)
15
16 GiveChangeFeatureStateAccess:
17  allow role Admin
18  to eu.ist.fears.server.FearsServiceImpl.changeFeatureState(Project p,
19     FeatureRequest feature, State newState)
20
21 ticket ChangeFeatureStateTicket(FearsServiceImpl fears)
22  to eu.ist.fears.server.FearsServiceImpl.changeFeatureState(Project p,
23     FeatureRequest feature, State newState)
24  where { p.isProjectAdmin(fears.getUserFromSession()); }
```

Listing A.2: The complete access control policy of the FeaRS using authorization rules, tickets, and amplification of privileges. (cont)

```
1  GiveChangeFeatureStateTicket:
2   on eu.ist.fears.server.FearsServiceImpl.addComment(String projectID,
3      String featureName, String comment, State newState, String sessionID)
4   give role LoggedIn ticket ChangeFeatureStateTicket(receiver)
5
6  GiveAddProjectAccess:
7   allow role Admin
8   to eu.ist.fears.server.FearsServiceImpl.addProject(String name,
9      String description, int nvotes, String sessionID)
10
11 GiveEditProjectAccess:
12  allow role Admin
13  to eu.ist.fears.server.FearsServiceImpl.editProject(String projectID,
14     String name, String description, int nvotes, String sessionID)
15
16 GiveDeleteProjectAccess:
17  allow role Admin
18  to eu.ist.fears.server.FearsServiceImpl.deleteProject(String name,
19     String sessionID)
20
21 GiveRemoveVoteAccess:
22  allow role LoggedIn
23  to eu.ist.fears.server.FearsServiceImpl.removeVote(String projectID,
24     String feature, String sessionID)
25
26 GiveGetAdminsAccess:
27  allow role Admin
28  to eu.ist.fears.server.FearsServiceImpl.getAdmins(String sessionID)
29
30 GiveAddAdminAccess:
31  allow role Admin
32  to eu.ist.fears.server.FearsServiceImpl.addAdmin(String userName,
33     String sessionID)
34
35 GiveRemoveAdminAccess:
36  allow role Admin
37  to eu.ist.fears.server.FearsServiceImpl.removeAdmin(String userName,
38     String sessionID)
39  where { !userName.equals(getUserFromSession().getName()); }
40
41 GiveAddProjectAdminAccess:
42  allow role Admin
43  to eu.ist.fears.server.FearsServiceImpl.addProjectAdmin(String newAdmin,
44     String projectID)
```

Listing A.3: The complete access control policy of the FeaRS using authorization rules, tickets, and amplification of privileges. (cont)

```
1  GiveRemoveProjectAdminAccess:
2   allow role Admin
3   to eu.ist.fears.server.FearsServiceImpl.removeProjectAdmin(
4      String oldAdmin, String projectID)
5   where { !oldAdmin.equals(getUserFromSession().getName()); }
6
7  GiveLogoffAccess:
8   allow role LoggedIn
9   to eu.ist.fears.server.FearsServiceImpl.logoff(String sessionID)
10
11 GiveProjectUpAccess:
12  allow role Admin
13  to eu.ist.fears.server.FearsServiceImpl.projectUp(String projectId,
14     String cookie)
15
16 GiveProjectDownAccess:
17  allow role Admin
18  to eu.ist.fears.server.FearsServiceImpl.projectDown(String projectId,
19     String cookie)
20
21 GiveUserCreatedFeatureAccess:
22  allow role LoggedIn
23  to eu.ist.fears.server.FearsServiceImpl
24     .userCreatedFeature(String cookie)
```

A.2 Exercise 2: Annotations

Listing A.4: The complete access control policy of the FeaRS using access control annotations.

```
1  GiveUserManagementAccess:
2   allow role Admin
3   to @eu.ist.fears.server.domain.annotations.UserManagement
4
5  GiveProjectManagementAccess:
6   allow role Admin
7   to @eu.ist.fears.server.domain.annotations.ProjectManagement
8
9  GiveUserFeatureInteractionAccess:
10  allow role LoggedIn
11  to @eu.ist.fears.server.domain.annotations.UserFeatureInteraction
12
13 ticket ChangeFeatureStateTicket(FearsServiceImpl fears)
14  to eu.ist.fears.server.FearsServiceImpl.changeFeatureState(Project p,
15     FeatureRequest feature, State newState)
16  where { p.isProjectAdmin(fears.getUserFromSession()); }
17
18 GiveChangeFeatureStateTicket:
19  on eu.ist.fears.server.FearsServiceImpl.addComment(String projectID,
20     String featureName, String comment, State newState, String sessionID)
21  give role LoggedIn ticket ChangeFeatureStateTicket(receiver)
22
23 GiveRemoveAdminAccess:
24  allow role Admin
25  to eu.ist.fears.server.FearsServiceImpl.removeAdmin(String userName,
26     String sessionID)
27  where { !userName.equals(getUserFromSession().getName()); }
28
29 GiveRemoveProjectAdminAccess:
30  allow role Admin
31  to eu.ist.fears.server.FearsServiceImpl.removeProjectAdmin(String
32     oldAdmin, String projectID)
33  where { !oldAdmin.equals(getUserFromSession().getName()); }
34
35 GiveLogoffAccess:
36  allow role LoggedIn
37  to eu.ist.fears.server.FearsServiceImpl.logoff(String sessionID)
```

A.3 Exercise 3: Domain Model Relations

Listing A.5: The complete access control policy of the FeaRS using relation rules.

```
1  GiveChangeFeatureRequestVotersAccess:
2   allow role LoggedIn
3   to change relation eu.ist.fears.server.domain.FeatureRequestVoters
4
5  GiveAddToProjectHasVotersAccess:
6   allow role LoggedIn
7   to add-to relation eu.ist.fears.server.domain.ProjectHasVoters
8
9  GiveAddToProjectFeatureRequestsAccess:
10  allow role LoggedIn
11  to add-to relation eu.ist.fears.server.domain.ProjectFeatureRequests
12
13 GiveAddToFeatureRequestAuthorAccess:
14  allow role LoggedIn
15  to add-to relation eu.ist.fears.server.domain.FeatureRequestAuthor
16
17 GiveAddToFeatureRequestCommentsAccess:
18  allow role LoggedIn
19  to add-to relation eu.ist.fears.server.domain.FeatureRequestComments
20
21 GiveAddToCommentAuthorAccess:
22  allow role LoggedIn
23  to add-to relation eu.ist.fears.server.domain.CommentAuthor
24
25 GiveChangeFearsAppHasProjectsAccess:
26  allow role Admin
27  to change relation eu.ist.fears.server.domain.FearsAppHasProjects
28
29 GiveAddToProjectAuthorAccess:
30  allow role Admin
31  to add-to relation eu.ist.fears.server.domain.ProjectAuthor
32
33 GiveAddToProjectHasVotersAccess:
34  allow role Admin
35  to add-to relation eu.ist.fears.server.domain.ProjectHasVoters
36
37 GiveAddToUserHasVotersAccess:
38  allow role Admin
39  to add-to relation eu.ist.fears.server.domain.UserHasVoters
40
41 GiveAddToUserHasVotersAccess:
42  allow role LoggedIn
43  to add-to relation eu.ist.fears.server.domain.UserHasVoters
```

Listing A.6: The complete access control policy of the FeaRS using relation rules. (cont)

```
1  GiveAddToFearsAppHasUsersAccess:
2   allow role Admin
3   to add-to relation eu.ist.fears.server.domain.FearsAppHasUsers
4
5  GiveChangeACUsersHaveACRolesAccess:
6   allow role Admin
7   to change relation
8      pt.ist.fenixframework.pstm.accesscontrol.ACUsersHaveACRoles
9
10 GiveAddToFearsAppHasAdminAccess:
11  allow role Admin
12  to add-to relation eu.ist.fears.server.domain.FearsAppHasAdmin
13
14 GiveAddToProjectHasAdminAccess:
15  allow role Admin
16  to add-to relation eu.ist.fears.server.domain.ProjectHasAdmin
17
18 GiveRemoveFromACUsersHaveACRolesAccess:
19  allow role LoggedIn
20  to remove-from relation
21     pt.ist.fenixframework.pstm.accesscontrol.ACUsersHaveACRoles
22
23 GiveChangeFeatureStateAccess:
24  allow role Admin
25  to eu.ist.fears.server.FearsServiceImpl.changeFeatureState(Project p,
26     FeatureRequest feature, State newState)
27
28 ticket loggedInChangeFeatureStateTicket(FearsServiceImpl fears)
29  to eu.ist.fears.server.FearsServiceImpl.changeFeatureState(Project p,
30     FeatureRequest feature, State newState)
31  where { p.isProjectAdmin(fears.getUserFromSession()); }
32
33 GiveLoggedInChangeFeatureStateTicket:
34  on eu.ist.fears.server.FearsServiceImpl.addComment(String projectID,
35     String featureName, String comment, State newState, String sessionID)
36  give role LoggedIn ticket loggedInChangeFeatureStateTicket(receiver)
37
38 GiveEditProjectAccess:
39  allow role Admin
40  to eu.ist.fears.server.FearsServiceImpl.editProject(String projectID,
41     String name, String description, int nvotes, String sessionID)
42
43 GiveGetAdminsAccess:
44  allow role Admin
45  to eu.ist.fears.server.FearsServiceImpl.getAdmins(String sessionID)
```

Listing A.7: The complete access control policy of the FeaRS using relation rules. (cont)

```
1  GiveRemoveAdminAccess:
2   allow role Admin
3   to eu.ist.fears.server.FearsServiceImpl.removeAdmin(String userName,
4      String sessionID)
5   where { !userName.equals(getUserFromSession().getName()); }
6
7  GiveRemoveProjectAdminAccess:
8   allow role Admin
9   to eu.ist.fears.server.FearsServiceImpl
10     .removeProjectAdmin(String oldAdmin, String projectID)
11  where { !oldAdmin.equals(getUserFromSession().getName()); }
```

Bibliography

[1] J. Cachopo. Development of Rich Domain Models with Atomic Actions. Instituto Superior Técnico, 2007.

[2] N. C. S. CENTER. *A Guide To Understanding Discretionary Access Control In Trusted Systems.* Fort George G. Meade, Maryland 20755-6000, September 1987.

[3] S. Chiba. A reflection-based programming wizard for java. In *Java Developer's Journal*, volume 9, January 2004.

[4] N. Damianou and N. Dulay. The ponder policy specification language. In *Lecture Notes in Computer Science*, pages 18–38. Springer-Verlag, 2001.

[5] G. M. Dumiense. Enforcing Complex Access Control Policies in Rich Domain Applications. Unpublished Master Thesis, 2008.

[6] B. Eckel and Z. F. Sysop. Thinking in java. In *PTR, Upper Saddle River, NJ.* Prentice Hall, 1998.

[7] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

[8] FénixEdu. Fénixedu. homepage: http://fenixedu.sourceforge.net., 2005.

[9] J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-grained access control with object-sensitive roles.

[10] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] J. B. D. Joshi. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In *Proc. of the Symp. on Access Control Models and Technologies (SACMAT).* ACM Press, 2006.

[12] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. In *IEEE Transactions on Knowledge and Data Engineering*, volume 17, pages 4–23, Pittsburgh University, PA, USA, January 2005. Springer-Verlag.

[13] C. Lai and L. Gong. User authentication and authorization in the java(tm) platform. In *In ACSAC 99: Proceedings of the 15th Annual Computer Security Applications Conference*, page 285. IEEE Computer Society, 1999.

[14] N. Li and J. C. Mitchell. RT: A Role-based Trust-management Framework. In *Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX'03)*, 2003.

[15] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First Experiences Using XACML for Access Control in Distributed Systems. In *Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, Fairfax, Virginia, USA, October 2003. ACM.

[16] H. G. Malheiro. Controlo de Acesso no Sistema Fénix. Unpublished Master Thesis, 2007.

[17] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. Javacop: Declarative pluggable types for java. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 32, New York, NY, USA, January 2010. ACM.

[18] OMG. Object Constraint Language. Homepage: http://www.omg.org/docs/formal/06-05-01.pdf.

[19] S. Osborn. Mandatory access control and role-based access control revisited. In *Proceedings of the second ACM workshop on Role-based access control*, pages 31–40, Fairfax, Virginia, United States, 1997. ACM Press.

[20] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. May 2007.

[21] D. Pendarakis and R. Guerin. A framework for policy-based admission control. 2000.

[22] Ponder2. The ponder2 project. homepage: http://ponder2.net/.

[23] C. Ribeiro and P. Ferreira. A policy-oriented language for expressing security specifications, 2005.

[24] G. Russello, C. Dong, and N. Dulay. Authorization and conflict resolution for hierarchical domains.

[25] V. Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10, New York, NY, USA, 1996. ACM.

[26] J. Wainer. A fine-grained, controllable, user-to-user delegation method in rbac. In *Proc. of the Symp. on Access Control Models and Technologies (SACMAT)*, pages 59–66. ACM Press, 2005.

[27] P. Zenida, M. M. de Sequeira, and D. Domingos. Zás Aspect-Oriented Authorization Services (second take).

[28] P. Zenida, M. M. de Sequeira, D. Henriques, and C. Serrao. Zás - Aspect-oriented Authorization Services. In *ICSOFT 2006*, 2006.

[29] S. Ziemer. An architecture for web applications essay in dif 8914 distributed information systems, 2002.