

Access Control in Rich Domain Model Web Applications

João de Albuquerque Penha Pereira
joao.pereira@ist.utl.pt

Instituto Superior Técnico

Abstract. Rich Domain Model Web applications present a serious challenge in the security and access control field. Policy Specification Languages are Domain Specific Languages designed specifically to define and express a policy of a system. Policy Specification Languages are also a very good candidate to express and manage access control policies in this type of applications. The Domain Model Authorization Policy Language is a Policy Specification Language which aims at the expression and management of access control policies in Rich Domain Model Web applications. It is inserted in a wider framework which also contains a model and a run-time engine. However, this framework, as well as the language itself, still requires further study and development.

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. This means that there are no standard ways of planning, testing, and assuring the quality of an application of this type. Also, Web applications typically have a short development cycle when in comparison with traditional applications due to time-to-market requirements [1]. In order to try to formalize 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) [2]. DDD focuses on the domain of a system and strongly supports that complex domains should be based on a model¹ describing all its relevant entities and the

¹ http://en.wikipedia.org/wiki/Domain_model

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) [3]. 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 [4]. For obvious reasons, 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 arise 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. And since there is no standard technology or methodology for the development of such applications, a standard way of performing access control is also lacking.

Previous research on this subject was made by Dumienne [5]. He first 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 to solve such problems which 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 Language (DMAPL) based on the previous model with special constructs to help express complex access control rules; and a run-time engine to enforce the access control 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 is guided to try to improve this solution as well as implement it in a real, large, and complex system to be able to get a reliable feedback upon its performance and possible problems.

The remaining of this report is organized as follow. I define the goals of my work in Section 2. An overview of related work is presented in Section 3. A more particular overview of my proposal is laid down in Section 4. The explanation of how such proposal is going to be evaluated is presented in Section 5. Finally, in Section 6, the final conclusions are presented.

2 Goals

The main goal of my work is to continue the research and development of the solution described in Dumienne [5]. Specifically, I first intend to review and improve its three components: the model; the DMAPL; and the run-time engine (these are described in more detail in Section 3). I propose to review the concepts introduced (authorization, amplification of privileges, and delegation of rights) and study if a unification of these concepts is possible.

Afterwards, to validate and better assess the solution's performance, I plan to implement it in a real, large, and complex RDM Web application currently used by several Universities in Portugal - the FénixEdu Web application [6]. The information obtained regarding the actual solution's performance should then guide the next steps in its development.

Some issues that were not entirely addressed by Dumienne [5] are also part of my working goals. One is to have a framework that can be easily used by a team of Java developers (such as the one behind FénixEdu) when specifying access control policies. Therefore, the development of the framework has to take in consideration the Java programming language and the OOP paradigm, and how its developers usually work and develop solutions. Another issue is the delegation of rights. Despite the solution's current support for delegation, it is not clear how this mechanism will be used. For example, through which interface will the users delegate?

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

3 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 two 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).

3.1 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 tradition Mandatory access control² (MAC) and Discretionary access control³ (DAC) 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 support delegation upon RBAC models, and I shall now present three of these proposals that are relevant regarding access control in RDM Web applications.

User-to-User Delegation

In [7], 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. In order to delegate a permission, a user has to have not only direct access to the specific permission, but also the right to delegate it. And when a user delegates the delegation of a permission, the grantee (user who receives the delegation) automatically gains direct access to the permission in order to be able to further delegate it.

When delegating, the user can specify if the delegation is **restricted** or **unrestricted**. For this, [7] provides a rich set of controls regarding **delegation chains**. A user can control how a delegation will be further delegated (for example specify its maximum depth), and also set **generic constraints** upon the delegation. This way, a user may hold **total delegation** or **conditional delegation** rights. And with generic constraints, it is possible to preserve certain

² http://en.wikipedia.org/wiki/Mandatory_access_control

³ http://en.wikipedia.org/wiki/Discretionary_Access_Control

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 (user who delegates the right) 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.

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

Hybrid Hierarchies

Several RBAC-based delegation models have been developed but most of them considers a general hierarchy type. In the context of [8], multiple types of hierarchies have been proposed and considered desirable in order 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 [8] 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. [9] addresses role-based delegation schemes in the presence of hybrid hierarchies.

Two new concepts are introduced by [9]: **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 1 for an example).

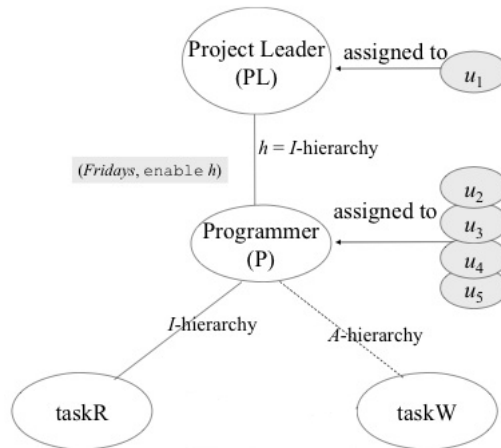


Fig. 1. Example of an upward delegation in a hybrid hierarchy. The project leader (grantee) mainly supervises the programming tasks. Only the programmers (grantors) do the coding. The project leader can only look at the tasks of the programmers each Friday. Role TaskR contains the read-only permissions 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.

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 supported by lter 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, [9] 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 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.

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 [10] 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), [10] 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 refer how [10] combines delegations and roles.

In [10], policy statements take the form of *role definitions*. An example of a simple role definition is

$$K_a.R \leftarrow K_b$$

$K_a.R$ 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

$$K_a.R \Leftarrow K_b : K_c.R_2$$

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 [9], 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 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, unlike GTRBAC, 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.

3.2 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 and an example of such language is Sun Microsystems Java. 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, and others) which makes the language highly portable. Therefore, and also due to great community support, Java becomes a great language of choice to develop applications which 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 and Zás, which are two Java-based access control mechanisms that try to cover these features.

JAAS

Java Authentication and Authorization Service (JAAS) [11] 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.⁴ So, 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 and for each action to be access controlled, changes in the action code are required.

Listing 1.1. Example of a permission definition in JAAS.

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

An example of a simple permission definition is shown in Listing 1.1, where *user1* has *read* access to file "foo.txt" while executing the code in *SampleAction.jar*.

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, not to mention that basic desirable access control features, such as delegations, are not addressed at all.

Zás

Zás [12] [13] 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

⁴ http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules

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 [12] 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. 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 requirements can be made in two ways: deep and shallow. When an access controlled resource is annotated with the attribute deep, its access control will not extend to other resources that may be invoked by the originally annotated resource. On the other hand, when in the presence of the attribute shallow, all the resources invoked by the annotated resource will inherit its access control.

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 1.2 is an example of an access control specification over the method *foo()*. This 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).

Listing 1.2. Example of an access control specification in Zás with permissions and trust.

```
public class A {
    @AccessControlled(
        requires = "aPermission",
        trusts = { B.class }
    )
    public void foo() {}
}

public class B {
    public void bar() {
        new A().foo();
    }
}
```

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.

3.3 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 (DSLs) 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 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 [14].

In this architecture (see Figure 1) 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

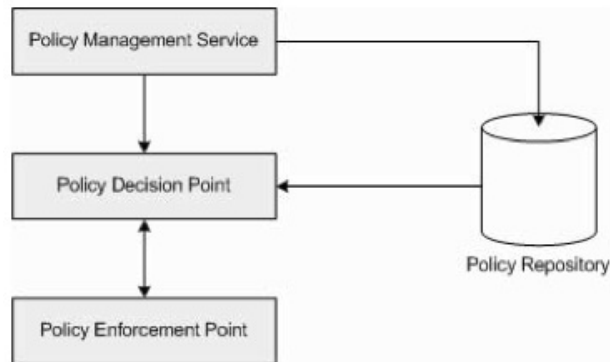


Fig. 2. IETF's proposed architecture for policy-based management

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 run-time 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 in order 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: Ponder; Ponder2; SPL; and DMAPL.

Ponder

Ponder [15] is a policy specification language originally developed at the Im-

perial 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 [16] (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 1.3 for an example). Conflicts between positive and negative authorizations can be detected through static analysis of the policy specification.

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 inputted 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 run-time 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 are used to separate 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 1.3. 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.

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

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

Ponder2

Ponder2 [17] is a policy specification language created upon the original Ponder and it is strongly based in Ponder's concepts. It is currently being developed

at the Imperial College and the main idea behind Ponder2 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 [14].

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. It can only be applied to the root domain to avoid possible conflicts since its value propagates to all the sub-domains.

Conflict Resolution

Similar to Ponder, Ponder2 also makes static analysis of the policy set to identify and resolve conflicts between policies. Unfortunately, conflicts of policies that depend on run-time state can be detected only in run-time. Ponder2 introduces a conflict-resolution strategy for such conflicts [18].

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.

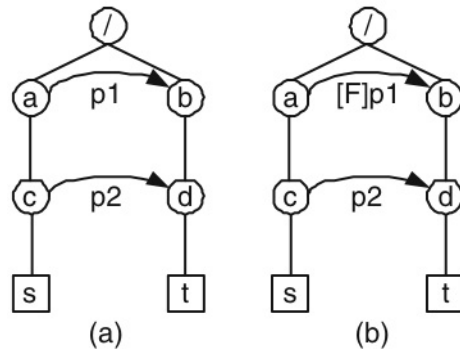


Fig. 3. Example of priority based on domain nesting (a) and final status (b)

In the example of Figure 3-(a), policy $p2$ takes precedence over policy $p1$ being $p2$ more specific than $p1$. In Figure 3-(b) 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.

SPL

SPL [19] is a policy-oriented constraint-based language for expressing security specifications developed at INESC-ID⁵ by Professor Carlos Ribeiro and Professor Paulo Ferreira. It has a specific algebra with 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 5):

- Elements
- Groups
- Rules
- Policies

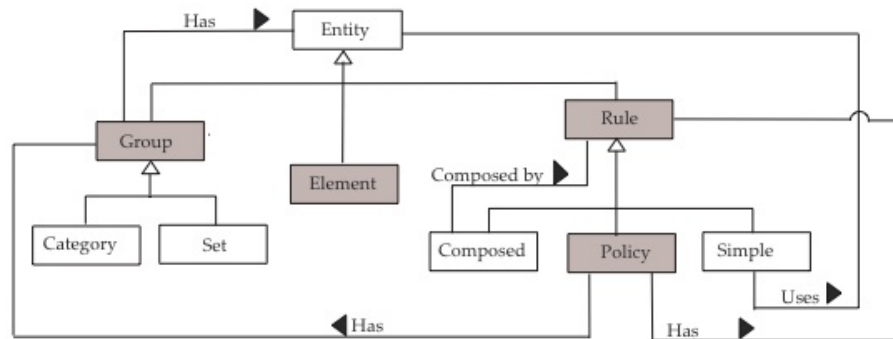


Fig. 4. SPL's structure and basic blocks

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

⁵ <http://www.inesc-id.pt/>

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 is composed of an optional label and two logic expressions (see Listing 1.4): 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 1.5).

Listing 1.4. Syntax of a simple rule

```
[label :] domain-expression :: decide-expression
```

Listing 1.5. Example of a simple rule where it is stated that payment order approvals cannot be done by the owner of payment order.

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

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.

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 ACID⁶ 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.

DMAPL

In [5], it is 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:

⁶ <http://en.wikipedia.org/wiki/ACID>

- Model
- Domain Model Authorization Policy Language (DMAPL)
- Run-time 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 1.6.

Listing 1.6. Example of a positive authorization rule in DMAPL.

```
GiveGradeAccess :
  allow role Teacher
  to studentrecord.Student
    .giveGrade(Course course, int grade)
  where { user.getPerson().getTeacher()
    .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 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 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 for a specific movie at the cinema, the ticket amplifies the privileges of the person which allows her to go to the cinema to see the specified movie. After entering the cinema or after the movie's starting time, the ticket is revoked from the person and the amplification of privileges ended. To specify when a user should receive a ticket, DMAPL uses a special type of policy rules called amplification rules. Once again, negative authorizations rules have priority over amplification rules.

Listing 1.7. Example a ticket and amplification rule in DMAPL.

```
ticket gradeTicket(Student student)
to studentrecord.Grade.getGrade()
  where { receiver.getStudent() == student }

OfficerAverageAccess:
  on studentrecord.Student.getAverage()
  give role officer ticket gradeTicket(receiver)
```

In Listing 1.7 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 run-time, the rights granted to the grantee (the user who receives the delegation) by the grantor (the user who delegates). 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 1.8).

Listing 1.8. Example of a delegation rule with constraint and validity constraint in DMAPL.

```
TeacherGiveGradeOnVacationDeleg:
  allow delegation of GiveGradeAccess
  grantor role Teacher
  grantee role Officer
  to studentrecord.Student
    .giveGrade(Course course, int grade)
  where { grantor.getPerson().getTeacher()
    .isOnVacation() }
  valid { grantor.getPerson().getTeacher()
    .isOnVacation() }
```

In the example in Listing 1.8, 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 its 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. [5] states that the DMAPL provides constructs that fit the way developers think when reasoning about security policies. The goals of DMAPL are essentially: *expressiveness*; *compositionality*; *human readable*; and *enable automation*.

The run-time 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. And, testing in a real, large and complex application such as FénixEdu [6] is also highly desired to assess its real performance.

Interesting future work in 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 [3] to increase its efficiency; study if the constructs provided by DMAPL are the most adequate for its purpose; and study how it would be possible to support cascade delegation of rights.

Comparison

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

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 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. Also, Ponder and Ponder2 are oriented towards directory services instead of objects in a rich domain, like DMAPL.

SPL makes use of a different and more powerful construction of rules in comparison with Ponder, Ponder2, and the DMAPL. However, its syntax as 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. And 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 four PSLs presented in this Subsection, DMAPL appears to be the most capable and well adapted to specify access control policies in RDM Web applications. DMAPL is also a PSL which clearly states to aim at the problems encountered when trying to specify access control policies in RDM Web applications. Therefore, I choose DMAPL to support my following work.

3.4 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. And 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.

As I mentioned before, the application I intend to use to validate and further develop the solution described in Dumienne [5] is the the FénixEdu Web application.

Research about FénixEdu's current access control mechanisms has been made by Malheiro [20]. 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 [20] 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 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. And 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.

4 Proposal Overview

After describing the goals of my work and the solution presented in Dumienne [5], in this section I will outline the possible steps of my work to be done upon the framework in [5].

There are three main paths which can be explored to significantly improve the DMAPL's solution:

Unification of concepts consists in trying to converge the concepts supported by the model: authorization, amplification of privileges, and delegation. The purpose behind this idea is to simplify the solution's implementation and maintenance by making its model simpler, and the use of the framework easier whether for a programmer or a security expert. Regarding this idea, I have made an early attempt of expression delegations solely with tickets from the amplification of privileges concept. In listings 1.10 and 1.11, I present an attempt to express de delegation specified in listings 1.8 and 1.9 using a ticket and an amplification of privileges rule.

Listing 1.9. Teachers' right to give grades specified in DMAPL.

```
GiveGradeAccess :
  allow role Teacher
  to studentrecord.Student
  .giveGrade(Course course, int grade)
  where { user.getPerson().getTeacher()
  .teachesCourse(course) }
```

Listing 1.10. Expression of the delegation right specified in Listing 1.9 using a ticket in DMAPL.

```
ticket giveGradeAccess(Student student)
  to studentrecord.Student
  .giveGrade(Course course, int grade)
  where { receiver.getStudent() == student &&
  course.getTeacher().isOnVacation() }
```

However, some problems arise. The loss of the grantor semantics which is important information regarding access control policies. And if the delegations were to be fully replaced by the tickets and amplification of privileges rules, the negative delegations would have to be replaced with the explicit introduction of corresponding negative authorizations, since it is not possible

Listing 1.11. Expression of the delegation rule specified in Listing 1.8 using an amplification rule in DMAPL.

```
OfficerGiveGradeAccess :
  on studentrecord.Student
    .giveGrade(Course course, int grade)
    give role officer ticket giveGradeAccess(receiver)
```

to express negative delegations with tickets. Further experiments with the unification of concepts shall be made.

Implementation in a real application is also needed in order to successfully validate a solution, and evaluate its performance with some accuracy. This is a crucial step in any peace of software being developed currently since it makes possible to get reliable feedback about the software's performance. After doing so, I plan to analyze the information gathered and guide the next steps of work according to any conclusions it may be drawn. For example, if it the current solution causes a big overhead in the system by enforcing access control policies, optimizations the run-time engine are probably required. Or if the DMAPL's expressiveness is insufficient or misleading when specifying or managing access control policies, changes in its syntax are probably needed.

The application chosen to install and test the solution is the FénixEdu Web application and its underlying framework, the Fénix framework. Fénix framework is the underlying framework of the FénixEdu Web application that supports the development of RDM applications.

Delegation user interface is also missing in this solution. Although there is a conceptual support and the respective mechanisms to enable delegations between users, there is no interface for them to order the action of delegation. This issue I believe, must be further studied in order to understand which is the most appropriate place in the system for the order of delegation to happen, how that order should be made.

I believe, that to first start with the implementation in a real system and the unification of concepts is the best approach, since this will lead to the next natural steps in the solution's development.

5 Methodology for Evaluating the Work

As stated before, testing and evaluating any peace 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 intend to use to implement and test the solution already mentioned before is FénixEdu [6]. 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, being this way a very good candidate to test our solution and provide us reliable feedback with real data.

The granularity level of the access control solution in Dumiense [5] is the Java interface method, corresponding directly to one of the current access control mechanisms present in the objects of the FénixEdu Web application [20].

Based on such conclusions, after the implementation of the solution presented in Dumiense [5], I will try to assess if it is functioning correctly and solving the desired problems mentioned earlier.

Other very important information to collect about this solution, is its performance evaluation. Often, the enforcement of access control policies causes very big overheads in the system, and it is important to try to minimize them in favor of the system's overall performance. As previously stated, part of my work is to integrate Dumiense [5] access control mechanisms in the Fénix framework. And since benchmarks to measure its performance are available, I will try to use them to assess the solution's performance.

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 in [5] present the most flexible and adapted framework to deal with access control policies in RDM Web applications.

However, [5] still needs further study and development. Specifically, it needs a real implementation to assess its real performance and highlight possible problems. An excellent candidate to test this framework is the FénixEdu Web application which is a real, large, and complex system used by several Universities for campus support and administration.

With the goals of my work well defined, I look forward to improve the solution described in [5] and see what the outcome might be. Hopefully, a usable and extensible access control mechanism specially crafted for Rich Domain Model Web applications, successfully implemented in FénixEdu.

References

1. Ziemer, S.: An architecture for web applications essay in dif 8914 distributed information systems (2002)

2. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2003)
3. Cachopo, J.: Development of Rich Domain Models with Atomic Actions. Instituto Superior Técnico (2007)
4. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
5. Dumienne, G.M.: Enforcing Complex Access Control Policies in Rich Domain Applications. Unpublished Master Thesis (2008)
6. FénixEdu: Fénixedu. homepage: <http://fenixedu.sourceforge.net>. (2005)
7. Wainer, J.: A fine-grained, controllable, user-to-user delegation method in rbac. In: Proc. of the Symp. on Access Control Models and Technologies (SACMAT), ACM Press (2005) 59–66
8. Joshi, J.B.D., Bertino, E., Latif, U., Ghafoor, A.: A generalized temporal role-based access control model. In: IEEE Transactions on Knowledge and Data Engineering, Volume 17., Pittsburgh University, PA, USA, Springer-Verlag (January 2005) 4–23
9. Joshi, J.B.D.: 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)
10. Li, N., Mitchell, J.C.: RT: A Role-based Trust-management Framework. In: Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX'03). (2003)
11. Lai, C., Gong, L.: User authentication and authorization in the java(tm) platform. In: In ACSAC 99: Proceedings of the 15th Annual Computer Security Applications Conference, IEEE Computer Society (1999) 285
12. Zenida, P., de Sequeira, M.M., Henriques, D., Serrao, C.: Zás - Aspect-oriented Authorization Services. In: ICSOFT 2006. (2006)
13. Zenida, P., de Sequeira, M.M., Domingos, D.: Zás Aspect-Oriented Authorization Services (second take)
14. Pendarakis, D., Guerin, R.: A framework for policy-based admission control. (2000)
15. Damianou, N., Dulay, N.: The ponder policy specification language. In: Lecture Notes in Computer Science, Springer-Verlag (2001) 18–38
16. OMG: Object Constraint Language. Homepage: <http://www.omg.org/docs/formal/06-05-01.pdf>
17. Ponder2: The ponder2 project. homepage: <http://ponder2.net/>
18. Russello, G., Dong, C., Dulay, N.: Authorization and conflict resolution for hierarchical domains
19. Ribeiro, C., Ferreira, P.: A policy-oriented language for expressing security specifications (2005)
20. Malheiro, H.G.: Controlo de Acesso no Sistema Fénix. Unpublished Master Thesis (2007)