An Access Control Service for Dynamic and Hierarchical Resources: Declarative Model and Implementation on top of XACML

Giuseppe Psaila[(1)] Fulvio Biondi[(2)]

[(1)]Università degli Studi di Bergamo
Facoltà di Ingegneria
Viale Marconi 5
I-24044 Dalmine (BG), Italy

[(2)]Ingenium Technologyn s.r.l.
Via Philips 3
I-20052 Monza (MI), Italy

psaila@unibg.it fulvio.biondi@ingeniumtech.it

Abstract: The increasing complexity of (distributed) information systems requires new solutions for dealing with access control problems. In particular, information systems are based on a large number of resources, with very complex structure, that must be accessed by a large variety of users. Traditional and instance based solutions are not adequate.

In this paper, we propose a new approach to the problem. First of all, we define an access control model which is declarative, modular, hierarchical and instance independent, so that it is suitable for highly dynamic contexts. Then, we reports about the implementation of a Profile Service, which effectively exploits the XACML technology to simplify and shorten the development.

1 Introduction

The increasing complexity of (distributed) information systems requires new solutions for dealing with access control problems. In particular, information systems are based on a large number of resources, with very complex structure, that must be accessed by a large variety of users. Traditional and instance based solutions are not adequate.

This is the case of the *I-Service* system, developed by *Ingenium Technology* [Ing], which provides Service Level Management support: any kind of service can be monitored; agreements over the quality of service are modeled in the system and reports based on the agreements are generated. Thus, this is a complex system that can be used by an incredible variety of users. system administrators, managers of the company providing the service, managers of the customer company, etc.. Furthermore, several services can be monitored at the same time and resources involved in these services can be a very large number and can vary in a very dynamic way.

In the context of access control solutions (we can cite just a few works [DPS03, BDS02,

JSS+01, GM, DSDP02], since this area is extremely dynamic; however, [DPS03] is an interesting and recent survey), an interesting technology is XACML (XML Access Control Modeling Language), a proposal of the OASIS group: this is a very powerful and declarative specification language, provided with a library, that constitute a framework on top of which it is possible to define more abstract access control models. Thus, it is not suitable as a practical solution, but allows to build simple and powerful solutions.

In this paper, we try to solve the above mentioned problem by proposing a new approach to the problem. First of all, we define an access control model which is declarative, modular, hierarchical and instance independent, so that it is suitable for highly dynamic contexts; the model is introduced and explained by means of a running example, inspired to the *I-Service* system. Then, we reports about the implementation of the Profile Service based on the proposed model within the *I-Service* system; in particular, we discuss the architecture of the Profile Service, whose implementation exploits the XACML technology, showing how this solution provides significant advantages, in that it simplified and shortened the development of the Profile Service.

The paper is organized as follows. Section 2 describes the problem, by moving from features of the *I-Service* system. Section 3 introduces the access control model, and explains it by means of several examples. Section 4 discusses the main features concerning the implementation of the Profile Service within the *I-Service* system; in particular, Section 4.1 briefly introduces the main features of XACML, while Section 4.2 describes the architecture in details. Finally, Section 6 draws the relevant conclusions.

2 The Problem

We introduce the problem by talking about the context from which it generated, i.e. the *I-Service* system developed by *Ingenium Technology* [Ing]. *I-Service* is a *Service Level Management* (SLM) system. it can be exploited by organizations to monitor the quality of a service (network service, back office service, etc., i.e. not necessarily an ICT service). By means of *Inspectors*, the system gathers a large volume of raw data that are later used to evaluate the quality of the service, for example by generating reports or by defining complex evaluation formulas that compare the actual service level with the desired service level, for example, as specified by a *Service Level Agreement* (SLA). Thus, the system is complex, usually deals with a large number of resources and with a large number of users; nevertheless, the structure of resources might be very complex, and not every user is allowed to access and/or perform actions on every resource or resource component, due to privacy concerns.

Consider, for instance, the case of reports. They are a family of resources, where each report is a resource instance. A report contains several sub-resources, such as *Charts* and *Matrices*, as well as resources corresponding to *Data Sources* and *Aggregation Functions* (which compute aggregate values for large sets of values). If we consider a *Matrix*, its specification is based on (thus contains) a pool of resources, such as *Style Parameters*, *Cell Definitions*, and so on; consequently, instances of these resources are contained in

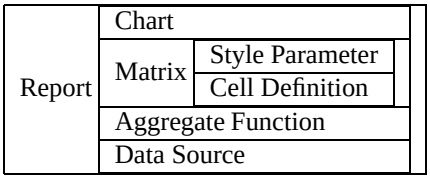

Table 1: Containment Relationship among resources for Reports.

an instance of *Matrix*, which is contained in an instance of *Report*. Table 1 shows this relationship.

This structure usually becomes rather complex, and may contain hundreds of resource instances in practical cases; furthermore, several users may access these resources with different roles (for example, report manager, executive, customer, etc.); to further complicate the situation, these resource instances may change dynamically. The main result of this situation is that an instance oriented access control model is not suitable for this context.

The problem can be summarized as follows.

Consider a set of resources, where each resource has a set of parameters and contains a set of sub-resources; sub-resources are recursively defined in the same way. Given a group of users G, it must be possible to assign access rights to users in G such that they concern a sub-set of resources, are properly propagated to sub-resources, and are not tied to specific instances of resources.

3 The Model

The access control model defined to cope with the problem introduced in Section 2 is based on several different concepts. They can be divided into two main categories: *Resources* and*Profiles*. We now introduce the concepts, by describing the syntax to specify them.

In the following, we write non-terminal symbols in italic; the derivation rule for a non-terminal symbol is in the right hand side of symbol $:=$; multiple repetitions of symbols or expressions are denoted as **(** *expr* **)**$*$ (i.e. the repeated expression is enclosed in bold face parenthesis followed by $*$); the symbol $|$ denotes alternatives.

3.1 Resources

Resource Schema. The basic concept in our model is the *Resource Schema*. It defines the structure of typologies of resources, and defines the containment relatiosnhips between resource typologies. The syntax of a resource schema definition (*ResourceSchemaDef*) is the following.

ResourceSchemaDef :=
`ResourceSchema(`*Family*`:`*Typology*`->`*ContainedIn*`)=`
`[(`*ParameterDef*`)*] {(`*ActionDef*`)*}`

where *Family* is the family which the defined resource typology belongs to, *Typology* is the name of the resource typology under definition, *ContainedIn* is the name of the resource typology that contains the resource typology under definition (the family of both resource typologies must be the same). Between square brackets it is possible to define a list of parameters (*ParameterDef*) which characterize the resource typology; between curly brackets it is possible to define the set of actions (*ActionDef*) allowed in resource instances belonging to the typology under definition.

The syntax of *ParameterDef* is

ParameterDef := `(`*ParamName*`,` *ParamType*`, [(`*ComparisonOp*`)*])`

where, for each parameter, it is necessary to specify the name, the data type and a list of comparison operators allowed for matching against the parameter.

Similarly, the syntax of *ActionDef* is

ActionDef := `(`*ActionName*`,` *ActionScope*`)`

where, for each action, it is necessary to specify the name and the scope, that can be either `common` (i.e. the action is propagated to contained resource typologies) or `custom` (the action is proper of the typology under definition only).

If a resource typology is not contained in any other resource typology, the feature *ContainedIn* can be omitted. However, if the feature *ContainedIn* is specified, the following constraint must hold: *given a resource typology T_1 that contains another resource typology T_2, if we denote with $Par(T_1)$ and $Par(T_2)$ the set of parameters for T_1 and T_2, respectively, it must be $Par(T_1) \subseteq Par(T_2)$*. The meaning of this condition is that T_2 inherits all the parameters of T_1, adding to these specific parameters; for the sake of simplicity in the specification, inherited parameters must not be redefined. The same is for common actions, that are inherited by contained resources.

Example 1: Consider the problem of defining reports. We might define the schema for reports, matrices and style parameters as follows..

```
ResourceSchema(Reports:Report)=
  [(Name, String)(Customer: String)(Service: String)]
  { (Create, common)(Modify, common)(Show, common) }
ResourceSchema(Reports:Matrix->Report)=
  [(MatrixId, String) (Importance, Int, [< = >])]
  { (Approve, custom) }
ResourceSchema(Reports:StyleParameter->Matrix)=
  [(StyleName, String)(] { }
```

`Report` is the main typology for family `Reports` (it is not contained in any other typology); its parameters are the report `Name`, the identifier of the `Customer` that receives the service, and the `Service` type; furthermore, three actions are possible, i.e. `Create`, `Modify` and `Show`, that can be inherited by contained resources. A `Matrix` is contained

Typology	*Parameters*	*Actions*
`Report`	`Name Customer Service`	`Create Modify Show`
`Matrix -> Report`	`Name Customer Service MatrixId Importance`	`Create Modify Show Approve`
`StyleParameter -> Matrix`	`Name Customer Service MatrixId Importance StyleName`	`Create Modify Show`

Table 2: Resource Typologies defined in Example 1.

in a `Report`, has a name (`MatrixName`) and a numerical value (`Importance`), which denotes the importance of the matrix in the report; notice that three comparison operators `< = >` are allowed.; an action `Approve` is added to the inherited ones (it is `custom`, thus not inherited below). Finally, resources `StyleParameters` are contained in matrices: for this typology, only parameter `StyleName` is added to the ones inherited from `Matrix`, while no actions are added. Table 2 shows the complete schemas for the three defined resource typologies. □

Resource Group. Defined schemas for resources, it is possible to define resource groups, i.e. groups of resource instances with some common features (notice that instances may be a priori infinite and can dynamically change). The syntax of a resource schema definition (*ResourceGroupDef*) is the following.

ResourceGroupDef :=
`ResourceGroup(`*ResourceGroupName*`) = (`*Family*`:`*Typology*`)`
`[((`*ParamName* , *ParamValue* , *ComparisonOp*`))`$*$`]`

where *ResourceGroupName* is the name of the resource group under definition, *Family* : *Typology* is the resource typology which resource instances belong to. Between square brackets it is possible to define the list of values for parameters: it is necessary to specify *ParamName*, i.e. the name of the resource parameter (defined in the schema of the resource typology) and the parameter value *ParamValue*; furthermore, *ComparisonOp* is the comparison operator used to match resource instances: a resource instance r is matched if, for each parameter specification $\mathit{ParamName}_i$, it holds

$$r.ValueOf(\mathit{ParamName}_i)\ \mathit{ComparisonOp}_i\ \mathit{ParamValue}_i$$

where $r.ValueOf(\mathit{ParamName}_i)$ denotes the value of specified parameter for instance r, while $\mathit{ParamValue}_i$ is the parameter value. In case only one comparison operator is defined in the schema of the resource typology, this is the default comparison operator and it can be omitted where the parameter appears in the resource group definition.

The left hand side of Figure 1 shows the relationship between resource schema and resource groups. In particular, the solid line arrow means that resource groups are sets of instances of resource typologies.

Example 2: We define three resource groups.

```
ResourceGroup(Rep_s1_c1)=
   (Reports:Report)[(Service, "s1")(Customer, "c1")]
```

```
ResourceGroup(Rep_c1)=
   (Reports:Report)[(Customer, "c1")]
ResourceGroup(ImportantMatrix_c1)=
   (Reports:Matrix)[(Customer, "c1")(Importance, 3 <)]
```

The first one contains all reports concerning service of type `"s1"` and customer `"c1"`; the second one all reports concerning customer `"c1"`; the third one all matrices in reports concerning services for customer `"c1"` with `Importance` less than `3`, thus very important (observe that parameter `Customer` is inherited from `Report`). □

3.2 Profiles

Over resource schemas and resource groups, it is possible to define profiles. In particular, we considered three levels of profiles: *Roles* (defined for resource schemas), *Responsibilities* (defined for resource groups) and *User Profiles* (defined for single users and related with roles, responsibilities and specific resource groups).

Role. A role is a set of permissions, which specify which actions can be performed on resources by users associated to the role. A role considers only the schema of resources. The syntax of a role definition (*RoleDef*) is the following.

RoleDef := `Role(`*RoleName*`)=[` **(** *TypologyGrant***)**$*$`]`

where *RoleName* is the name given to the role. Between square brackets, it is possible to define the list of grants concerning typologies (*TypologyGrant*). Their syntax is the following

TypologyGrant := `Grant_for_Typology(`*Family*`:`*Typology*`,`
`{` **(** *ActionName* **)**$*$`}`, *PropagationMode*`)`

where `Grant_for_Typology` specifies the resource typology for which the grant is defined and the set of actions allowed (within curly brackets). The *PropagationMode* can be of two types: if it is `local`, the grant is limited to the typology for which it is defined; if it is `propagate`, the grant is propagated to all contained resource typologies, for all actions defined as `common` (thus, inherited by contained resource typologies).

Example 3: Based on the schema for reports, we can define the standard role for report administrators; notice that the grant is propagated, so that automatically all contained resources can be managed though this role (it implicitly grants actions `Create`, `Modify` and `Show` for `Matrix` and `StyleParameter`).

```
Role(ReportAdmin)=
  [Grant_for_Typology(Reports:Report,
    {(Create)(Modify)(Show)}, propagate)]
```

□

Responsibility. A role is independent of specific resource instances. To set up grants concerning resource instances, but still independent of specific users, the concept of *Responsibility* must be used. The syntax of a responsibility definition (*ResponsibilityDef*) is

the following.

ResponsibilityDef :=
`Responsibility(`*ResponsibilityName*`) = [`**(***RoleGrant* | *ActionGrant* **)***`* `]`

where *ResponsibilityName* is the name of the defined responsibility. Within square brackets, it is possible to define a list of grants, that can be either of type *RoleGrant* or of type *ActionGrant*. *RoleGrant* grants permissions about roles; *ActionGrant* grants actions on resource groups, independently of the role. The syntax for *RoleGrant* is

RoleGrant := `Grant_Roles( {` **(** `(`*RoleName*`)` **)***`*`}, {` **(** `(`*GroupName*`)` *)***`*`} )`

where between the first curly brackets it is possible to define a set of role names, while in the second curly brackets it is possible to define a set of resource group names for which the previous set of roles is granted.

The syntax for *ActionGrant* is

ActionGrant := `Grant_Actions( {` **(** `(`*ActionName*`)` **)***`*`}, {` **(** `(`*GroupName*`)` *)***`*`},`
PropagationMode `)`

where between the first curly brackets it is possible to define a set of granted action names, while in the second curly brackets it is possible to define a set of resource group names for which the previous set of actions is granted. Finally, if *PropagationMode* is `local`, the grant is limited to resource instances in the resource group for which it is defined; if it is `propagate`, the grant is propagated to all contained resource instances, for all actions defined as `common` (thus, inherited by contained resource typologies).

Example 4: Suppose we want to define some responsibilities for reports.

```
Responsibility(Manager_c1) =
    [Grant_Roles({(ReportAdmin)}, {(Rep_c1)})]
Responsibility(Standard_User_c1) =
    [Grant_Actions({(Show)}, {(Rep_c1)}, propagate)
```

Responsibility `Manager_c1` grants the role `ReportAdmin` for resources in the resource group `Rep_c1`; consequently, this responsibility allows actions `Create`, `Modify` and `Show` on resource instances of type `Report`, `Matrix` and `StyleParameters` (since the propagation mode was set to `propagate` in the role). Responsibility `Standard_User_c1` grants action `Show` on all reports and sub-resources in the resource group `Rep_c1`. □

User Profiles. Finally, it is necessary to grant permissions to single users, i.e. defining *User Profile*s. The syntax of a user profile definition (*UserProfileDef*) is the following.

UserProfileDef := `UserProfile(`*UserProfileName*`) =`
`{` **(** `(`*UserName*`)` **)***`* `} [(` *RoleGrant* | *ActionGrant* | *RespGrant* `)` **)***`* `]`

where *UserProfileName* is the name given to the profile. The list of users, to which this profile is granted, is reported within the first parenthesis. Within square brackets, it is possible to specify a list of grants: a grant can be a *RoleGrant* (previously defined), or an *ActionGrant* (previously defined) or a *RespGrant* that grants responsibilities. The syntax of *RespGrant* is

RespGrant := `Grant_Responsibilities( {` **(** (*ResponsibilityName*) **)**$*$ `} )`

where, within curly brackets, it is possible to specify the list of granted responsibilities. Notice that the propagation mode is specified in the definition of responsibilities.

Example 5: Finally, we can define user profiles.

```
UserProfile(c1_users)={(John)(Jim)}
  [Grant_Responsibilities({(Standard_User_c1})]
UserProfile(c1_manager)={(Jeff)}
  [Grant_Actions({(Approve})), {(ImportantMatrix_c1)},
                local)]
UserProfile(manager_for_c1_and_s1)={(Jack)}
  [Grant_Role({(ReportAdmin)}, {(Rep_s1_c1)})]
```

User profile `c1_users` grants responsibility `Standard_User_c1` to users `John` and `Jim`: they can show all reports and sub-resources in the resource group `Rep_c1`. User profile `c1_manager` grants user `Jeff` (which should belong to enterprise `c1`) action `Approve` for resource instances in the resource group `ImportantMatrix_c1`: thus, `Jeff` can approve matrices in the report. Finally, user profile `manager_for_c1_and_s1` grants `Jack` (which belongs to the enterprise providing the service) the role `ReportAdmin` for resources in the resource group `Rep_s1_c1`: thus, `Jack` can perform actions `Create`, `Modify` and `Show` on all reports and sub-resources for which customer is `"c1"` and the service type is `"s1"`. □

The right hand side of Figure 1 shows dependencies between concepts about profiles, while the overall figure shows dependencies between all defined concepts. In particular, dotted line arrows means that a generic profile (either role or responsibility) is granted, while dashed line arrows denotes that actions are granted for resources (either resource typologies or resource instances).

4 Implementation

The profile service based on the model presented in Section 3 has been implemented in the *I-Service* system. Fully implemented in the Java Language, it exploits the XACML technology.

4.1 XACML

XML Access Control Modeling Language (XACML, [XACa]) is an initiative coordinated by the OASIS group. XACML presents several interesting features, making it a modern solution for access control.

General architecture. XACML envisions an architecture for access control where separate components are responsible for the different roles of the access control mechanism.

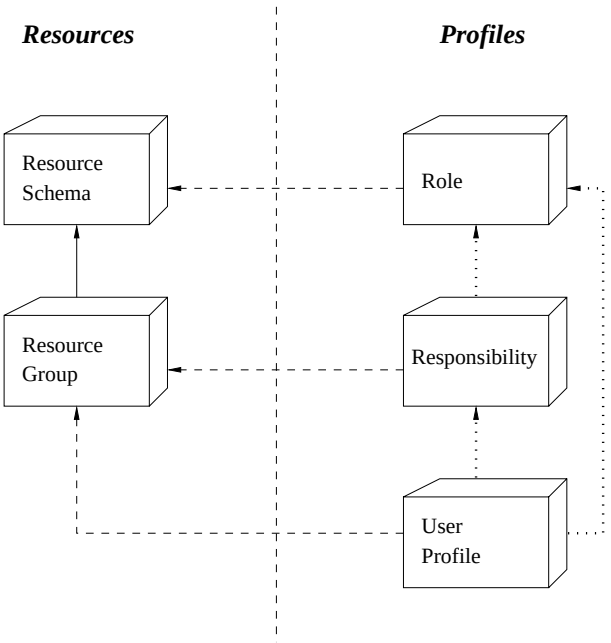

Figure 1: Structure of the Model.

The architecture envisions four distinct actors.

- *Policy Administration Point (PAP)*: it represents the component that manages the access control policies;
- *Policy Execution Point (PEP)*: it represents the component that controls the resource to which access is required;
- *Policy Decision Point (PDP)*: it represents the component that evaluates an access request, typically sent by a PEP, and determines if it is consistent with the defined access control policy;
- *Policy Information Point (PIP)*: it represents the component that supports the construction of access requests.

Access policies are created on the PAP. When a subject requires access to a resource managed by the PEP, the PEP creates a request that is sent to the PDP. The request will contain a description of the subject requesting access, of the resource asked, and of the action that the subject requires on the resource. The request will contain additional information describing the time and in general the environment where the request originated; the PIP is designed to support the construction of this portion of the request. The PDP receives the request and evaluates it with respect to the policy provided by the PAP; it then returns a positive or negative answer.

In this architecture, XACML specifies the XML format in which the policy is described. It also specifies an XML format for the request.

One of the main advantages of XACML is its compatibility with novel Web and network applications, that require a distributed construction of access control services. The design of XACML tried to reach a balance between the comprehensiveness of the specification, which increases the degree of cooperation that systems built around XACML will be able to offer, and its flexibility, which makes an application able to extend it to suit its specific needs.

We can resume the advantages of using XACML. First of all, XACML is an open format, based on XML and independent of the platform. Second, it is totally declarative, in that resources do not have to be specified by their identities; it overcomes the classical distinction between the representation of the access control matrix in terms of ACLs or capabilities. Finally, XACML is scalable; in fact, it allows the integration among different policies, defined in separate contexts.

Nevertheless, XACML is difficult to use. In fact, it is a very powerful instrument, but both the XML syntax, and its generality (i.e. it is not tied to any particular type of resources) make it difficult to be directly used to express complex access rights structures, such as the ones proposed in this paper. In contrast, it is a good tool, provided that a higher level model has been devised: concepts and access rights expressed in this higher level model can be automatically mapped into complex XACML PDP specifications; then, an XACML engine can effectively validate requests to access resources.

4.2 Architecture

The architecture of the profile service is reported in Figure 2. Its definition has been conditioned by the overall architecture of the *I-Service* system, where the server side provides core functionality, such as instantiation of services, data storage, etc.. On the client side heterogeneous applications can be executed, such as configuration and management consoles, report generators, etc... Thus, the profile service can be exploited by any client application in the system. Let us describe the architecture, by discussing how the components interact.

1. The client application provides username and password to the *Local Profile Manager*, a client side object that is responsible to communicate with the server side.

2. The *Local Profile Manager* transmits username and password to the *Server profile Manager*, an always running service that authenticates the user.

3. The *Server Profile Manager* exploits its repositories for user authentication and, in case of success, for deriving a *Profile Description* for the logged on user; this description is obtained by analyzing all direct and indirect grants specified for the user.

4. The *Profile Description* is received by the *Local Profile Manager*, which generates a *Profile* object. This newly generated object is actually able to handle authorizations over

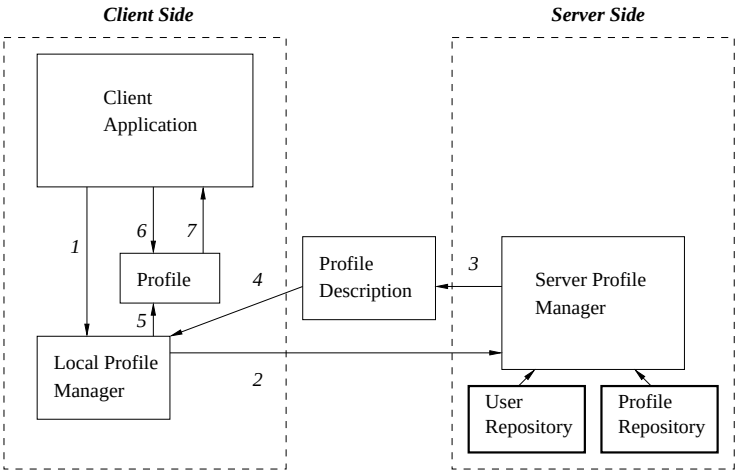

Figure 2: Architecture of the Profile Service.

specific resource instances.

5. The *Profile* object is passed to the *Client Application*.

6. When the *Client Application* has to perform a specific action on a specific resource instance, it asks the *Profile* object to know if the action is allowed for the user.

7. The *Profile* object gives or denies the required authorization, by exploiting the XACML engine to formulate the answer.

The Profile Object. The *Profile Description* object is simply a serializable object that describes the portion of model concerning the logged on user. In practice, the *Server Profile Manager* explores all grants, and extracts all of them that concern the user; these are described by the *Profile Description* object.

From this, the *Local Profile Manager* builds a *Profile* object. By exploiting the description provided by the *Profile Description* object, the corresponding XACML formulation is derived and an XACML PDP is built.

At this point, when the *Client Application* requires the authorization to perform an action on a given resource instance, it passes the required action and the full description of the resource instance (i.e. the values of parameters, as in the resource schema) to the *CheclPermission* method of the *Profile* object. The Method generates an XACML request, that is submitted to the XACML engine provided the PDP. The XACML engine resolves the request and its answer is sent back to the *Client Application*.

We exploited Sun's XACML engine, freely available on the Internet [XACb]. This engine is fully implemented in Java, and provides a rather complete implementation of XACML standard. In practice, this engine provides two features: at first, it loads a PDP, which specifies access policies to resources; then, it evaluates requests to access specific resources, allowing or denying the access, based on the previously loaded PDP.

Advantages Provided by XACML. Although the access control model we defined is not based on XACML (in fact, it does not exploit any specific concept provided by XACML), however the exploitation of XACML provided several advantages in the implementation. The first we can notice is that the *Profile* class was built by working at a declarative level: in fact, the XACML engine is driven by deriving a declarative specification (the XACML specification for PDP) from a data structure. A second advantage is the modularity of an XACML specification: in fact, for each component in the access control model, i.e. resource schemas, resource groups, roles, responsibilities and user profiles, an XACML fragment is generated in isolation, ignoring the other components. All these XACML fragments are composed together in a unique XACML PDP specification in a very modular way. Them, it is the XACML engine that exploits all dependencies to answer the request. Consequently, a third advantage was provided by the exploitation of XACML: the time of development was rather short, since the programmers were able to build the *Profile* class in a few days, because they exploited the powerful features provided by the XACML engine (we can figure out that the development, without using XACML, of a Java class able to evaluate the proposed access control model would have required at least three months, due to the semantic richness of the proposed model).

5 Related Work

The problem of access control has been recognized as a crucial problem in information systems since thirty years ago. The problem can be easily stated as follows: an information system has to protect data and resources against unauthorized disclosure (secrecy) and unauthorized or improper modifications (integrity), while at the same time ensuring their availability to legitimate users (no denials-of-service); enforcing protection thus requires that every access to a system and its resources be controlled and that all and only authorized accesses can take place [SD01].

The problem can be considered at different levels, i.e. at the level of *security policy*, at the level of *security model* and at the level of *security mechanism*. This paper is concerned with the *security model* level.

There has been a long series of articles and works on the topic of access control models and mechanisms. It would be impossible to discuss this history in the limited space available for the paper. Anyway, we want to relate our work with some key situations.

First of all, our model is substantially a *role based model* [OSM00]; this is a necessary choice, since it has to deal with a large number of resources and a large number of users. Second, our model cannot be an Access Control List-based (ACL-based) model, since the number of resources is large, and resources are hierarchically structured and dynamically change. Furthermore, solutions developed for databases (see, for instance, [Da95, PB95]) are not suitable as well, since the nature of data (i.e. resources in the database context) does not correspond to the nature of resources (based on a hierarchical relationship denoting containment) typical of our application context. Finally, not even a view of resources such the one typical of directory systems is suitable [GM]: this is due to the fact that in our

context, a resource contains other resources, and contained resources inherit properties from the container resource.

The reader can refer to [DPS03, BDS02, JSS+01, DSDP02] for recent works and surveys on the topic.

6 Conclusions

In this paper, we present an innovative approach to solve access control problems in complex information systems. In particular, the approach was devised to deal with sets of complex and hierarchical resources, in order to express access rights in a very declarative way, and independently of the specific set of resource instances. This latter requirement was fundamental to deal with highly dynamical contexts. The practical application context that generated the problem was the *I-Service* system, a *Service Level Management* (SLM) system produced by *Ingenium Technology*.

We afforded the problem in two steps. First of all, we devised the access control model; the model is based on a hierarchical structure, in order to capture complex situations (concerning roles and responsibilities for users) declaratively and with a very reduced number of rules, being at the same time independent of the specific resource instance. Then (second step), we implemented the profile service in the *I-Service* system, by exploiting XACML, the OASIS's proposals for access control problems, and the SUN's implementation [XACb]. The exploitation of XACML was very effective, in that allowed us to work in a declarative and modular way, thus simplifying the development and dramatically reducing the time of development (a few days against the estimated three months without using XACML).

References

[BDS02] Bonatti, P., De Capitani di Vimercati, S., and Samarati, P.: An algebra for composing access control policies. *ACM Transactions on Information and System Security*. 5(1):1–35. February 2002.

[Da95] Date, C. J.: *An Introduction to Database Systems*. Addison Wesley. sixth edition, 1995.

[DPS03] De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P.: Access control: Principles and solutions. *Software - Practice and Experience*. 33(5):397–421. April 2003.

[DSDP02] Damiani, E., Samarati, P., De Capitani di Vimercati, S., and Paraboschi, S.: Xml access control systems: a component-based approach. *Informatica (Slovenia)*. 26(2). 2002.

[GM] Goodwin, M. and McDonell, K. J.: Access control for network directory systems. In: *Proceedings of the ACM SIGCOMM conference on Communications architectures and protocols, 1986, Stowe, Vermont, United States August 5-7, 1986*.

[Ing] Ingenium Technology Web Site. *http://www.igenium-tech.it*.

[JSS+01] Jajodia, S., Samarati, P., Sapino, M., , and Subrahmanian, V.: Flexible support for multiple access control policies. *ACM Transactions on Database Systems*. 26(2):214–260. June 2001.

[OSM00] Osborn, S., Sandhu, R., and Munawer, Q.: Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on INformation and System Security*. 3(2). 2000.

[PB95] Polk, W. R. and Bassham, L. E.: Security issues in the database language SQL. Technical Report NIST Special Publication 800-8. Institute of Standards and Technology. 1995.

[SD01] Samarati, P. and De Capitani di Vimercati, S.: Access control: Policies, models, and mechanisms. *Riccardo Focardi, Roberto Gorrieri (Eds.): Foundations of Security Analysis and Design*. Lecture Notes in Computer Science 2171. 2001.

[XACa] OASIS XACML Web Site. *http://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=xacml*.

[XACb] Sun's XACML Implementation. *http://sunxacml.sourceforge.net*.