Automated Management and Analysis of security policies using Eclipse

Marco Guarnieri, Eros Magri, Simone Mutti [⋆]

Dipartimento di Ingegneria dell'Informazione e Metodi Matematici,
Università degli Studi di Bergamo, Italy
{marco.guarnieri,eros.magri,simone.mutti}@unibg.it

Abstract. The design of efficient and effective techniques for security policy analysis and management is a crucial open problem in modern information systems. The increasing complexity of current IT systems requires new techniques for designing access control policies.
Thus, in order to ease the definition and management of access control policies, a tool chain that lets developers defining and managing security policies is needed. This tool chain can be used to support a model-driven approach to the definition and implementation of access control policies, in which the policies are refined in several steps in order to produce concrete security configurations.
In this paper we present an extension of the *PoSecCo Eclipse Policy Plug-in* (PEPP), which provides to the users three different reasoning services for detecting anomalies in security policies. The reasoning services are based on Semantic Web and ontology management technologies, which offer an adequate basis for the realization of techniques able to support conflict analysis in security policies. The three services are: (a) *Policy Incompatibility*, (b) *Redundancy Detection*, and (c) *Separation of Duty Conflicts Detection*.

1 Introduction

The widespread diffusion of information systems in the last decade has led to a considerable growth in the capabilities and in the width of the range of services offered. However this improvement in terms of flexibility was followed by a huge increase both in terms of complexity and management of the systems itself. This increase in the systems' complexity has had an effect also on the management of the security of the system for two reasons. Firstly, the complexity of the system and the number of interconnections between elements of the system has increased the attack surface of the system itself. Secondly, it has increased the complexity of the security management process by system administrators. Recent analysis of information security breaches has found empirical evidences which highlight the fact that security management is a difficult and error prone task [7], [5], [4].

[⋆] The research leading to these results has received funding from the European Commission Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-257129 "PoSecCo".

The management of security requirements is, thus, a critical task that has the goal of avoiding possible information disclosures and to show compliance with respect to the many regulations promulgated by governments. A particular area of security requirement management is access control management, which focuses on defining a set of rules, called policies, that define for each user which actions he/she can do on the resources of the information system. Thus, in order to ease the definition and management of access control policies, a tool chain that lets developers defining and managing security policies is needed. This tool chain can be used to support a model-driven approach to the definition and implementation of access control policies. The model-driven approach leads to the specification of security requirements at an abstract level, then the refinement of these requirements in several iteration and in the end to a concrete implementation. This approach guarantees that given a correct set of requirements, the actual configuration of the system derived satisfies the requirements defined at the beginning of the process.

In order to help security administrators in the definition of access control policies we have developed the *PoSecCo Eclipse Policy Plugin* [9] that can be used to define access control policies, to check the absence of structural errors and to reason about the model in order to identify conflicts and inconsistencies. Semantic Web is an extension of the current idea of Web that supports the sharing of data between computers, in a way that let computers to easily understand and reason about data. The World Wide Web Consortium (W3C) defined a set of standards languages, protocols and technologies that can be used to partially realize this vision, and thus let users to enable exploration and experimentation and to support the evolution of the concepts and technology. We use the Web Ontology Language (OWL), which is a family of knowledge representation languages based on Description Logic (DL) with a representation in RDF, to define our model and to describe access control policies. We then use Semantic Web technology, namely *OWL-DL* [1], *SWRL* [2] and *SPARQL* [3], to provide security administrators with the following reasoning services:

1. **Policy Incompatibility** service (given a set of authorizations A, check whether exist pairs of authorizations (a_1,a_2) such that a_1 and a_2 apply to the same request and have opposite sign),
2. **Redundancy Detection** service (given a set of authorizations A, check whether a subset of those authorizations R exists that is dominated by other authorizations),
3. **Separation Of Duty Satisfiability** service (given a set of authorizations A, check whether they satisfy a set of Separation of Duty (SoD) constraints).

Section 2 presents the base model that we use to describe access control policies. Section 3 presents in detail the three reasoning services, whereas in Section 4 we present the architecture of the tool. Section 5 reports on experimental results, using as representative policies a model built over bibliographical databases. Finally, Section 6 draws a few concluding remarks and presents future work.

2 Policy model

Modeling security requirements requires a rich metamodel, in order to represent the large number and variety of entities involved in actual enterprise scenarios, and a rich and flexible language to express relationships between the entities. We have chosen to represent our model by applying techniques from the *Semantic Web* and knowledge representation area, which provide us with flexible and expressive tools to specify the entities and the relationships between them. We have represented our model in terms of OWL classes, which represent sets of entities, and OWL properties, which represent relations between a class called domain of the property and a class called the range of the property.

The use of OWL to define policies has several advantages that become critical in distributed environments involving coordination across multiple organizations. First, most policy languages define constraints over classes of targets, objects, actions and other constraints (e.g., location). A substantial part of the development of a policy is often devoted to the precise specification of these classes. This is especially important if the policy is shared among multiple organizations that must adhere to or enforce the policy even though they have their own native schemas or data models for the domain in question. The second advantage is that OWL's grounding in logic facilitates the translation of policies expressed in OWL to other formalisms, either for analysis or for execution.

Our model, which is based on Role Based Access Control (RBAC) model [10], contains the following OWL classes:

Principal: it represents *Identities* and *Roles*. The former are composed of *SingleIdentity*, i.e. single users, and *Group*, which represents sets of users. The latter represent specific enterprise function. In RBAC each role represents the binding between users and permissions associated to them. In our model users can activate roles dynamically. The property *containedIn*: *Identity* $\rightarrow$ *Group* represents the hierarchy of groups and the property $roleHierarchy$: *Role* $\rightarrow$ *Role* represents the hierarchy of roles. The transitive closures of the two properties can be used to explicitly represent the expanded hierarchy.

Resource: it represents the entitities in the information system whose access must be regulated through security policies. We modeled a taxonomy of resources and thus we can represent several kinds of resources, from files to databases, from folders to network elements. Our model allows the representation of a resource hierarchy by means of the property *resourceHierarchy*: *Resource* $\rightarrow$ *Resource*, which represents the structural containment between two resources (e.g., a folder that contains files).

Action: it represents the types of operations over resources available to principals. The structure of the action is strictly related to the characteristics of the underlying access control system.

Authorization: it represents the allowed and forbidden behaviors for a principal. Each authorization is characterized by its sign, represented by the property $sign$: $Authorization \rightarrow \{+, -\}$, and the principal (identity or role) it is granted to, represented by the property $grantedTo$: $Authorization \rightarrow$ *Principal*. Each authorization has only one sign (authorizations with pos-

itive sign are called *PositiveAuthorization*, whereas authorizations with negative sign are called *NegativeAuthorization*) and it is granted to only one principal.
We have defined two kinds of authorizations: (1) *SystemAuthorization* represents the fact that a principal p is allowed/disallowed to do a certain action a on a certain resource r, depending on the sign of the authorization. (2) *RoleAuthorization* represents the fact that a principal p can activate a certain role r.

3 Reasoning Services

We have implemented the reasoning services over policies expressed using our OWL model by means of several *Semantic Web* technologies. Several approaches and techniques exist, each one specifically optimized with a focus on the trade off between expressivity and performance on specific application scenarios. The main reasoning techniques are:

– *Description Logic* (DL) reasoning is usually based on tableaux techniques. It is highly optimized to perform consistency checks, concept classification, instance classification and information retrieval. However, it introduces some limitations on the types of axioms that can be used to model policies and constraints.
– *SWRL* rules are a particular kind of implications, which can be expressed using a subset of the OWL language. SWRL rules are written as antecedent consequent pairs. In SWRL terminology, the antecedent is referred to as the rule body and the consequent is referred to as the head. The head and body consist of a conjunction of one or more atoms. An atom is (a) a class, (b) a property, (c) one of the predefined built-in expressions (e.g., *DifferentFrom* or *SameAs*), each atom can refer to individuals using variables, denoted by a question mark as a prefix. The intended meaning is consistent with the classical semantics that characterizes most rule paradigms: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. Reasoning techniques based on *SWRL* rules compute a closure of the original model and provide less restrictive graph expressions to describe model fragments, enriched by the firing of SWRL rules.
– *SPARQL* is a query language that can be used to extract data from RDF data, which is a directed and labeled graph data format for representing information in the Web, by defining required and optional graph patterns that the selected data should satisfies.

Semantic Web technology has made available an extensive collection of tools. Several approaches can be adopted, relying on the tools presented above, with different profiles in terms of abstractness and efficiency. In general, tools and approaches are available that offer a simple and direct representation of the policy, e.g. *SWRL* and *SPARQL*, but may present scalability problems when applied to complex analysis tasks, and other tools may require greater effort in the representation of the problem, but offer scalability and support the analysis of complex properties in large-scale scenarios, e.g. *OWL-DL*. This paper will present experimental results that will confirm that the use Semantic Web tools for policy analysis permits the adoption of a variety of approaches, with a choiche that depends on the complexity of the analyzed property and the size of the policy. A *one-size-fits-all* strategy cannot be considered adequate.

Using the techniques presented above we have defined the following reasoning services:

– The *Policy Incompatibility* reasoning service aims at checking whether a set of System Authorizations contains pairs of authorizations $a_p \in \mathit{PositiveAuthorization}$ and $a_n \in \mathit{NegativeAuthorization}$ such that a_p and a_n apply to the same principal, action and resource. If such a pair exists, this means that the policy contains an inconsistency.
  This kind of conflicts is usually called *Modality conflict* and arises when principals are authorized to do an action a on a resource r by a positive authorization, and forbidden to do the same action a on the resource r by a negative authorization. In this case the two authorizations are said to be *incompatible.*
  A simple criteria that can be used to solve these kind of conflicts is the "*Denials take precedence*", which states that, in case of conflicts, the negative authorization always wins. A more flexible criteria is the "*Most specific Wins*", which states that, when one authorization dominates the other, the more specific wins.
  We have implemented this reasoning service using SWRL rules. For instance one of the rules that enforce the "*Most specific Wins*" criterion is the following:

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

– The *Redundancy Detection* service aims at identifying and removing redundancies in a given access control policy. Given two authorizations a_1 and a_2 (contained in the same policy) with same action and sign, and called p_i (respectively r_i) the principals (respectively resources) associated, directly or indirectly, with a_i. If $p_2 \subseteq p_1$ and $r_2 \subseteq r_1$ and a_2 is not involved in any conflict with other authorizations, then a_2 is redundant with respect to a_1, and can be safely removed from the policy without modifying the behaviour of the system. As highlighted by [8], actual policies are usually unnecessarily complicated and contains redundancies that increase the costs of managing and updating the policy. We have implemented the service by using both *SWRL* rules and *SPARQL* queries. For instance the *SPARQL* query that can be used to extract redundant authorizations is the following:

```
PREFIX rbac:
    <http://www.posecco.eu/ontologies/accesscontrol#>
```

```
SELECT ?auth2 WHERE { ?auth1 rbac:implies ?auth2 .
        NOT EXISTS { ?auth2 rbac:unRemovable ?auth3 . } }
```

– The *Separation of Duty Satisfiability* service aims at checking whether a policy satisfies a set of Separation of Duty (SoD) constraints. SoD constraints follow the common best practice for which sensitive combinations of permissions should not be held by the same individual in order to avoid the violation of business rules. These constraints can be implemented in RBAC by expressing that given two roles r_1 and r_2, it must not exist a user u who can activate both r_1 and r_2. We have implemented this service using only *DL* logic. In this way the performance of this service are higher than the performance of the other two. We express SoD constraints on *Identities* by adding to the ontology an axiom in the form:

$$SoDConflictOnId \equiv \exists canHaveRole+.\{role_1\} \sqcap \exists canHaveRole+.\{role_2\}$$

for each *RoleAuthorization auth* such that $sign(auth, -)$, $grantedTo(auth, role_1)$, $enabledRole(auth, role_2)$. In order to enforce the SoD constraints we simply have to add to the ontology the axiom $SoDConflictOnId \sqsubseteq \bot$.

4 Architecture

Model driven approaches for the definition and management of the security configurations of IT systems may be an effective way of handling the increasing complexity of those systems and may also reduce the impact of security misconfigurations. In these approaches, as a first step the administrator defines a high-level security policy which is close to the business requirements. Then he/she refines the policy in several steps, by adding details about the system. During this refinement process the administrator can execute automated analysis tools over the policy, in order to detect conflicts and anomalies in the policy itself. The results of the analysis can lead to further modifications to the policy, which require to adapt the current policy to the changes. Finally, the administrator can derive and deploy, in a semi-automated way, the concrete security configuration of the system.

Our tool aims at allowing security administrators to define, manage and analyze security policies through several refinement steps. We have chosen to implement it on the basis of the Eclipse framework for four main reasons:

1. Eclipse is by now one of the de-facto standard in terms of IDEs and has several plugins related to model driven engineering and thus, in our opinion, the Eclipse framework is flexible enough to support our needs,
2. it provides several useful characteristics that ease the definition of the GUI of the tool,
3. it can be easily integrated with Semantic Web tools by using the *OWLAPI* Java library,
4. by defining a new extension point it lets us defining an extensible and flexible way to handle the integration and customization of the reasoning services in the architecture.

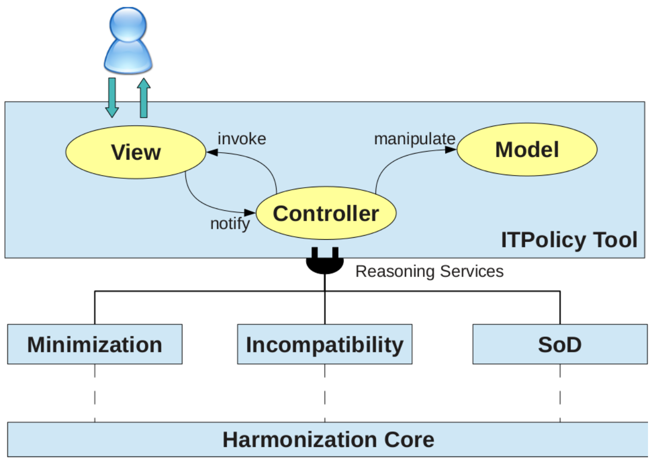


Fig. 1. Architecture of the Tool

The architecture of the tool is shown in Figure 1, it consists of four main components.

PoSecCo Eclipse Policy Plugin: It is the main component of the architecture, and it is presented more in detail in [9]. It contains the GUI and represents the component that manages all the interaction with the user. Its main task is to provide to the user a form-based interface that can be used to define a high level representation of the security policy and to refine this policy through several steps. The policies can be stored in XMI format and in XACML format. It is implemented by following the Model-View-Controller (MVC) pattern:

– Controller module. The Controller module represents the core of the entire plugin. It instantiates each module at start time, receives the command from the user (through the View module) and executes the related action on the data. It also invokes the reasoning services when needed.
– Model module. The Model module permits to maintain a dynamic representation of the IT Policy. The model is a collection of Java classes that keep updated the information about the IT Policy.
– View module. The View module provides the functionalities to show on the screen the information.

The *PoSecCo Eclipse Policy Plugin* defines an extension point, called *ReasoningService*, that allows other plugins to contribute to the global architecture with dedicated reasoning services. In this way we can decouple the definition of the reasoning services from the tool. This solution ease the definition of new reasoning services without modifications to the *PoSecCo Eclipse Policy Plugin* component.

The extension point, which definition is presented in Listing 1.1, has two attributes: (1) *name* represents the name of the reasoning service, (2) *class* represents the class that implements the reasoning service, which has to extend the *IReasoningService* interface.

```
<element name="ReasoningService">
  <complexType>
    <attribute name="class" type="string" use="required">
        <meta.attribute kind="java" basedOn=":IReasoningService"/>
    </attribute>
    <attribute name="name" type="string" use="required"/>
  </complexType>
</element>
```

Listing 1.1. Definition of ReasoningService Extension Point

The *IReasoningService* interface has three methods:

- `boolean isConsistent(Policy p)`: it takes as input a policy and, by using reasoning, checks whether it is consistent or if it contains one or more conflicts, and returns an adequate boolean value.
- `List <Explanation> getExplanation(Policy p)`: it takes as input the policy and, in case p is not consistent, returns a list of explanations that tries to identify the inconsistencies.
- `List <Fix> getRepair(Policy p)`: it takes as input the policy and, in case p is not consistent, it returns a list of fixes that can remove the identified inconsistencies.

Policy Incompatibility Service: it is the component that implements the *Policy Incompatibility* reasoning service. It defines the *PolicyIncompatibility* extension for the *ReasoningService* extension point. An example of the extension is shown in Listing 1.2.

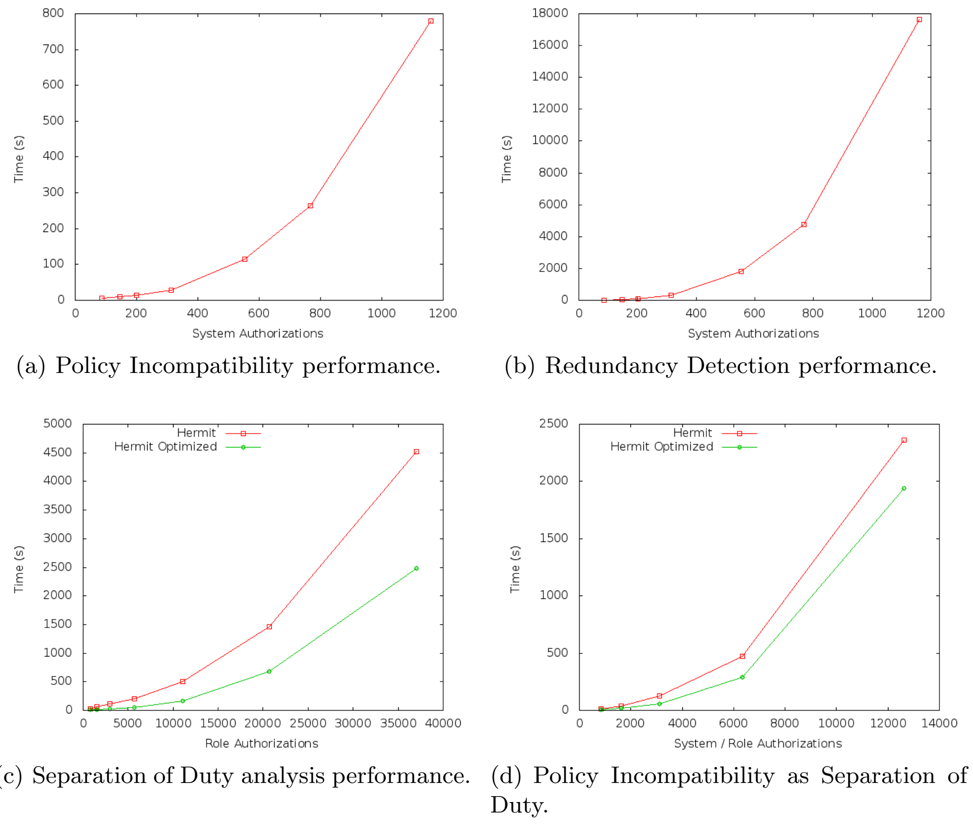
```
   <extension id="services" point="ReasoningService">
      <service class="PolicyIncompatibilityService"
            name="Policy Incompatibility Reasoning Service" />
  </extension>
```

Listing 1.2. Definition of PolicyIncompatibility reasoning service Extension

Redundancy Detection Service: it is the component that implements the *Redundancy Detection* reasoning service. It defines the *RedundancyDetection* extension for the *ReasoningService* extension point.
SoD Conflict Detection Service: it is the component that implements the *SoD Conflict Detection* reasoning service. It defines the *SoDConflictDetection* extension for the *ReasoningService* extension point.
Harmonization Core: it is the component that contains all the functionalities shared by the reasoning services. It contains the classes that manage the ontology by using the *OWL-API* Java library (enriched by the use of SWRL and SPARQL). It allows the definition of an ontological representation of the IT Policy. *OWL-API* through *OWLDataFactory* objects permits to instantiate

all classes, properties and axioms of the ontology. This component is a dependency of the reasoning service components presented above, because all these components use core functionalities. The dedicated functionalities of a specific reasoning service, e.g. a particular reasoner or a set of SWRL rules, are included only in the specific component.

5 Case Study



(a) Policy Incompatibility performance. (b) Redundancy Detection performance.

(c) Separation of Duty analysis performance. (d) Policy Incompatibility as Separation of Duty.

Fig. 2. Experimental results

For the purpose of evaluating our approach on use cases, since there are no freely available large datasets of real security policies we chose to test our prototype against policies built according to an interpretation of the data in bibliographic databases. We used randomly selected subsets of *PubMed Central*[1] (PMC), well known in the medical sciences, and DBLP[2], well known in the computer science community. Each of them provides a rich set of attributes and relationships that represent real and extensive social networks. PMC has rich information about journals, with a description of editorships and the funding of

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

papers. DBLP has a rich description of conferences. The two databases support different experiments.

The *Policy Incompatibility* service solves the problem about the incompatibility between policies, presented in Section 3. It receives as input a policy and produces as output a list of authorizations involved in conflicts that are not solvable using the "*Most specific wins*" or "*Denials takes precedence*" criteria. For this experiment we use randomly generated subsets of the bibliographic database *PubMed Central*.

We created instances of principals in the following way: (a) for each *author* or *editor* of a paper, we add a *SingleIdentity*, (b) for each group of authors that have written a paper together, we create a *Group* containing the *SingleIdentity*, (c) for each group of editors of a conference or a journal, we create a *Group* containing the *SingleIdentity*, (d) for each journal issue (and conference for the DBLP case) , we create an "editor" *Role* and an "author" *Role*. We then created the following authorizations:

- for each paper author we create the positive authorizations to *read* and *write* the paper and the negative authorization to *review* the paper;
- for each editor of the issue of the journal containing the paper we add the positive authorizations to *read* and *review* the paper and the negative authorization to *write* the paper;
- for each author that receives funding from the same grant that funded the paper, we add a negative authorization to *review* the paper and a positive authorization to *read* the paper;
- for each group containing all the authors belonging to the institution to which the author is affiliated, we add the authorization to *read* the paper;
- for each member of the editorial board of the journal that published the paper we add the positive authorizations to *read* and *review* the paper and the negative authorization to *write* the paper;
- for the group of authors of the paper, we add the positive authorizations to *read* and *write* the paper and the negative authorization to *review* the paper (these are redundant authorizations);
- for the group of editors of the paper we add the positive authorizations to *read* and *review* the paper and the negative authorization to *write* the paper (these are also redundant authorizations).

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

The *Redundancy Detection* service identifies redundant authorizations in a policy. The service receives as input a policy and produces as output a list of redundant authorizations. The approach proposed in Section 3 shows how SWRL and SPARQL can be used to find redundant policies. We tested our service with authorizations produced starting from randomly selected subsets of the bibliographic database *PubMed Central*, with the same model used for Policy Incompatibility.

The *Separation of Duty Conflicts Detection* (*SoD*) service solves the problem presented in Section 3. In this case, the service receives as input a policy containing negative role authorizations and produces as output a list of users that break the *SoD* constraints. We tested our service using a model that is built from a randomly selected subset of conferences in the DBLP bibliographic database. We enforced the SoD constraints on author and editor roles for the same conference/journal issue.

We have run *Policy Incompatibility* and *Redundancy Detection* services on random samples of the PMC dataset, while we have run the *Separation of Duty* service on random samples of the DBLP dataset. Experiments have been run on a PC with two Intel Xeon 2.0GHz/L3-4MB processors, 12GB RAM, four 1-Tbyte disks and Linux operating system. The results of the experiments are reported in the charts in Figure 2. Each observation is the average of the execution of ten runs.

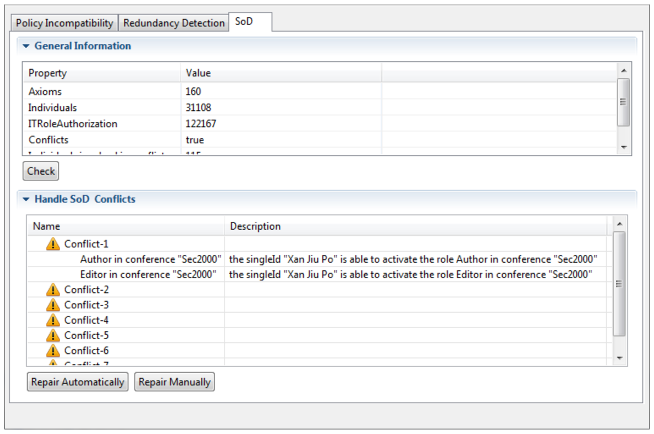
The first set of experiments aimed at evaluating how the performance of the *Policy Incompatibility* service evolves with the increase in the number of authorizations. Figure 2(a) reports the observed performance. It is clear that this solution is applicable for policies with a relatively small number of authorizations. For large policies, the simplicity in the definition of the verification rules is associated with a large computational cost.

The second set of experiments aimed at evaluating the response time of the *Redundancy Detection* service with the increase of the number of authorizations. Figure 2(b) reports the results of these experiments, which exhibit response times for the same policies that are significantly larger than those observed in the previous experiments. The specific test case is characterized by a large number of redundant authorizations and in general the analysis required for the identification of redundant rules requires to produce a large number of derivations. As presented in [8], policies are often unnecessarily complicated due to redundancies and for the cases typically considered in these analyses the number of authorizations is relatively small and this makes our approach acceptable. However, for the largest policies considered in our experiments, a significant benefit can be obtained from the use of tools, like the reasoners able to efficiently process DL structures, which require a different and more complicated representation of the problem.

The third experiment aimed at evaluating the performance of the *Separation of Duty* service. Furthermore, Figure 3 depicts the results produced by the *Separation of Duty* service.

In Figure 2(c) we can see how the performance evolves with the increase of the number of role authorizations. Even for extremely large policies (more than 35,000 role authorizations), the response times remain adequate for the profile of the security design activity, which operates in sessions that have long duration.

The pruning algorithm presented in [6] can provide a significant contribution to performance improvement. As demonstrated in [6], the approach used for the management of *Separation of Duty* constraints can be adapted to solve the *Policy Incompatibility* problem. Figure 2(d) reports the performance observed



Property	Value
Axioms	160
Individuals	31108
ITRoleAuthorization	122167
Conflicts	true

Name	Description
⚠ Conflict-1	
Author in conference "Sec2000"	the singleId "Xan Jiu Po" is able to activate the role Author in conference "Sec2000"
Editor in conference "Sec2000"	the singleId "Xan Jiu Po" is able to activate the role Editor in conference "Sec2000"
⚠ Conflict-2	
⚠ Conflict-3	
⚠ Conflict-4	
⚠ Conflict-5	
⚠ Conflict-6	

Fig. 3. An example of the *Separation of Duty* service output.

using this approach, offering a remarkable performance increase compared to the normal approach depicted in Figure 2(a).

6 Conclusions

Information systems are becoming more difficult to manage, with the intention of resources of different owners, and access offered to a larger variety of users. Service oriented architectures facilitate this evolution. We presented approaches that enhance the management of security policies in these scenarios. We described the relationship between the PoSecCo security model and OWL and showed how the Eclipse framework provides a powerful environment for tool integration and already makes available a large number of functions that are commonly required in the construction of such systems.

The work described in this paper testifies that Eclipse can also have an important role in the construction of modern environments for the management of security policies in large and integrated information systems.

References

1. OWL Web Ontology Language Reference. Technical report, W3C, http://www.w3.org/TR/owl-ref/, 2004.
2. SWRL: A semantic web rule language combining OWL and ruleML. Technical report, W3C, http://www.w3.org/Submission/SWRL/, 2004.
3. SPARQL Query Language for RDF. Technical report, W3C, http://www.w3.org/TR/rdf-sparql-query/, 2008.
4. Data breach investigations report. Technical report, Verizon Business, 2009.
5. 7safe. Uk security breach investigations report. Technical report, 2010.
6. M. Guarnieri and E. Magri. Techniques for conflict detection and minimization for access control policies, Master Thesis, Università degli Studi di Bergamo, 2012.
7. J. Langevin, M. McCaul, S. Charney, and H. Raduege. Securing cyberspace for the 44th presidency. Technical report, DTIC Document, 2008.
8. I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo. Mining roles with multiple objectives. *ACM Trans. Inf. Syst. Secur.*, Dec. 2010.
9. S. Mutti, M. A. Neri, and S. Paraboschi. An eclipse plug-in for specifying security policies in modern information systems. In *Proc. of ECLIPSE-IT 2011*, 2011.
10. R. Sandhu. Role-based access control. *Advances in computers*, 46, 1998.