Using OWL Reasoner to Control the Data Accessibility

Zinnia Sultana, and Mohammed Mahmudur Rahman

Department of Computer Science & Engineering, International Islamic University Chittagong, Bangladesh.

Corresponding Author: Mohammed Mahmudur Rahman

Abstract
In this work we have presented firstly a translation of policy documents those written in extended XACML into OWL-DL and then described how those translations can be used for policy analyzing and matching by exploiting OWL reasoner functionalities. As a reasoner we have exploit Hermit reasoner capabilities. We have provided some background knowledge about XACML, extended XACML and OWL to make clear the problem and suggested solution. We believe that, this approach can be used to help developing security frameworks for dynamic environments which require an agreement before sharing data. Policy Matching application would be useful in several fields in which access control is main issue. Its flexible structure enables to adapt to the application domain easily and cover all necessary access control problems. We have demonstrated that representing policies with OWL has several advantages: Firstly most policy languages define constraints over classes of targets, objects, actions, time etc. which can be defined easily with OWL. Secondly OWL’s logic facilities enable to translate of policies expressed in OWL to other formalisms either for analysis or for execution. Moreover OWL reasoning capacity allows users manage and solve policy conflicts in an automated, error free way particularly in hierarchical issues and in multi-organizational environments.

Keywords: semantic web, owl reasoner, policy ontology, access data, loader, matcher

INTRODUCTION
A significant feature of any information system is to provide an access control mechanism that means protecting data and resources from unauthorized access while at the same time guaranteeing their availability to the authorized users. Access control is the process of determining an access decision, permit or deny, to each request which is made to access resources. The access decision is imposed by a security policy that involves regulations.

Policy matching application basically composed of two parts: first component called “Loader” which allows user to upload a policy file to the system. System will parse uploaded policy document and extract the related information and save it into Policy Ontology. Second component called “Matcher” allows user to select two policies from existing ontology and match them. System will recursively query policies' components and compare them one by one. Finally result will be presented to the user.

We use the “match” concept to compare two policies in order to see if one is less/more permissive than the other. To be more clear consider following situation: suppose a subject, let us call provider, defines a policy in which he/she states which data can be accessed by whom and how this data can be used etc. Now suppose another subject, let us call consumer, wants to access the provider’s data. Consumer should define a policy as well. In order to decide allowing access or not we have to match these two policies. If data consumer has required authentications and agrees on defined obligations then he/she shall access data, otherwise not. The authentications and obligations in consumer’s policy should be more restrictive, less permissive than in the provider’s policy.

SYSTEM DESIGN
Policy matching application composed of two modules: Loader and Matcher. Loader allows users to upload policy documents, those specified in extended XACML. Loader parses uploaded documents and extracts related information by examining DataHandlingPreference or DataHandlingPolicy element and its children. Then extracted information will be stored in Policy Ontology. Matcher module allows user to select two policies from Policy Ontology in order to match them. Matcher will conclude data consumer may access requested data if and only if data consumer has required authentications and agrees on the specified obligations. To determine these features, Matcher will send queries to OWL reasoner continuously. Figure 1 indicates system model.

If data consumer’s policy is equally or less permissive than data provider’s policy then it can be concluded that data consumer has required authentications and agrees on specified obligations. Data consumer’s policy is equally or less permissive than data provider’s policy if and only if the set of authorizations specified in the data consumer’s policy
is equally or less permissive than the set of authorizations specified in the data provider’s policy and the set of obligations specified in the data consumer’s policy is equally or less permissive than the set of obligations specified in the data provider’s policy.

The meaning of less permissive for a set of authorizations and obligations is defined as follows: for each authorization in the data consumer policy, there should be a more permissive authorization in the data provider policy. On the other hand matching function for obligations is different: for each obligation in the data provider policy, there should be a less permissive obligation in the data consumer policy. Each obligation composed of three parts:

- TriggerSet: List of triggers resulting in the execution of action
- Action: Performed operation
- Validity: Validity period of the obligation.

If data provider and consumer both have “ActionNotifyUser” then we will compare usernames.

C: ActionNotifyUser \(\preceq\) P: ActionNotifyUser

(C.userName == P.userName)

If data provider and consumer both have “ActionNotifyByEmail” then we will compare usernames and addresses.

C:ActionNotifyByEmail \(\preceq\) P: ActionNotifyByEmail

((C.userName == P.userName) \&\& (C.address == P.address))

If data consumer has “ActionNotifyUser” and data provider has “ActionNotifyByEmail” then we will compare usernames.

C:ActionNotifyByEmail \(\preceq\) P: ActionNotifyUser

(C.userName == P.userName)

The reasoning process is going on similarly until every single part matched. Assuming C is data consumer policy while P is data provider policy and \(\preceq\) symbol stand for equally or less permissive, logical formulas about reasoning process can be formalized as follows:

- C: Policy \(\preceq\) P: Policy

\(( ( C.authorizations \preceq P.authorizations ) \&\& (C.obligations \preceq P.obligations ) )\)

Data consumer’s policy is equally or less permissive than data provider’s policy if and only if the set of authorizations specified in the data consumer’s policy is equally or less permissive than the set of authorizations specified in the data provider’s policy and the set of obligations specified in the data consumer’s policy is equally or less permissive than the set of obligations specified in the data provider’s policy.

- C: ListAuthorizations \(\preceq\) P: ListAuthorizations

\( ( i \in C ) : \(\preceq\) ( j \in P ) \) where (i \(\preceq\) j)

Data consumer’s authorization set is equally or less permissive than data provider’s if and only if for each authorization in the data consumer’s authorization set, there is a more permissive authorization in the data provider’s authorization set.

- C: ListObligations \(\preceq\) P: ListObligations

\( ( i \in C ) : \(\preceq\) ( j \in P ) \) where (i \(\preceq\) j)

Data consumer’s obligation set is equally or less permissive than data provider’s if and only if for each obligation in the data consumer’s obligation set, there is a less permissive obligation in the data consumer’s obligation set.

- C: Obligation \(\preceq\) P: Obligation

\(( ( C.action \preceq P.action ) \&\& ( C.triggers \preceq P.triggers ) ) \&\& ( C.validity \preceq P.validity )\)

An obligation in data consumer’s policy is equally or less permissive than an obligation in data provider’s policy if and only if triggers, action and validity in data
consumer side are equally or less permissive.

• C:ListTriggers ⊆ P:ListTriggers ⊆ (b ∈ P) : ∅ (a ∈ C) where (a ⊆ b)

A trigger set in data consumer’s policy is equally or less permissive than a trigger set in data provider’s policy if and only if for each trigger in the data provider’s trigger set, there is a less permissive trigger in the data consumer’s trigger set.

\[ C:\text{Validity} \sqsubseteq P:\text{Validity} \ (\text{(C.start} = \text{P.start}) \land \text{(C.end}} = \text{P.end}) \]

A validity in data consumer’s policy is equally or less permissive than a validity in data provider’s policy if and only if start date of validity in data consumer side is less or equal than validity in data provider side while end date is great or equal.

**STRUCTURE OF POLICY ONTOLOGY**

Policy Ontology is created to represent data provider and consumers’ access control policies in OWL. Policy matching application uses Policy Ontology and exploits OWL reasoner capabilities to recursively query each single component in policies. The main class of Policy Ontology is “Policy”. Each policy must have exactly one “ObligationSet” and also one “AuthorizationSet”.

AuthorizationSet involves Authorizations and likewise ObligationSet involves Obligations. It is possible to associate an obligation or an authorization directly to the policy element. However processes over these classes are different thus it is easy to manage and differentiate them by defining a set concept. Once we get access a set then we will behave same to the all elements within that set. Figure 2 and Figure 3 shows class hierarchy in Policy Ontology.

Figure 2: Class Hierarchy

Authorizations specify actions those are allowed to perform over accessed data. Authorization class can be divided into two. “DownStreamUsageAuthorization” class states if the specified data can be forwarded to the third parties or not. Permission decision is determined by “hasPermission” data property. “UsagePurposeAuthorization” class states what the purpose of using the data is and the purpose is specified by “hasPurpose” data property.

Obligations specify actions to be performed by the data consumer after accessing requested data. An obligation is an action on an event if a specific condition is satisfied. Each obligation consists of three parts: TriggerSet, Action, and Validity. In our structure triggers can be seen as the set of events that result in actions and validity can be seen as obligation’s validity period.

TriggerSet contains “Triggers” whose action and validity are same. Trigger class can be divided into subclasses. “TriggerAtTime” is a time-based trigger that occurs only once between specified time slots. This class has two data properties: “hasStart” specifies start date of trigger while “hasMaxDelay” specifies maximum delay before execution.

“TriggerPeriodic” is a time-based trigger that occurs multiple times on a periodic basis between start and end. Data properties of this class are: “hasStart” and “hasEnd” state start and end dates. “hasMaxDelay” specifies maximum delay while “hasPeriod” specifies periodicity of trigger.

“TriggerPersonalDataAccessed” is an event-based trigger. This trigger occurs each time the personal data associated with the obligation is accessed for one of the specified purposes. Data properties of this class are: “hasData” which is reference to the personal data concerned by the obligation. “hasPurpose” states purpose that trigger the obligation and “hasMaxDelay” specifies maximum delay before execution.

Validity class represents obligation’s validity time. It owns “hasStart” and “hasEnd” data properties which state start and end date of validity.
LOGIC AXIOMS FOR POLICY ONTOLOGY

Now let us formalize Policy ontology with logic axioms by starting with Policy class: Policy class objects can be defined as ones those have exactly one obligationSet and one AuthorizationSet.

Policy $\equiv \forall 1 \text{hasAuthorizationSet} \cap \forall 1 \text{hasObligationSet}$

AuthorizationSet class has a restriction states that each AuthorizationSet should contain at least one Authorization. And ObligationSet should contain at least one Obligation.

AuthorizationSet $\equiv \exists 1 \text{hasAuthorization} \text{ObligationSet} \equiv \exists 1 \text{hasObligation}$

Authorization class is union of its two subclasses named DownStreamUsageAuthorization and UsagePuposeAuthorization. These two subclasses are disjoint (△ symbol refers to empty).

Authorization $\equiv$ DownStreamUsageAuthorization $\cup$ UsagePuposeAuthorization

Authorization $\equiv$ DownStreamUsageAuthorization $\subseteq$ UsagePuposeAuthorization

DownStreamUsageAuthorization $\cap$ UsagePuposeAuthorization $\subseteq$ UsagePuposeAuthorization

Obligation class objects should have exactly one TriggerSet, one Action and one Validity.

Obligation $\equiv \forall 1 \text{hasTriggerSet} \cap \forall 1 \text{hasAction} \cap \forall 1 \text{hasValidity}$

TriggerSet class objects should have at least one Trigger.

TriggerSet $\equiv \exists 1 \text{hasTrigger}$

Trigger class has three subclasses: TriggerAtTime, TriggerPeriodic and TriggerPersonalDataAccessed.
These subclasses are disjoint from each other.
Trigger ⊑ TriggerAtTime ∪ TriggerPeriodic ∪ TriggerPersonalDataAccessed

Trigger ⊑ TriggerAtTime
Trigger ⊑ TriggerPeriodic
Trigger ⊑ TriggerPersonalDataAccessed
TriggerAtTime ∩ TriggerPeriodic ∩ TriggerPersonalDataAccessed ⊑

Action class consists of three subclasses:
ActionAnonymizePersonalData, ActionDeletePersonalData and ActionNotifyDataSubject. These subclasses are disjoint from each other. Moreover ActionNotifyDataSubject has two subclasses as well:
ActionNotifyDataSubjectByEmail and ActionNotifyDataSubjectBySMS.

Action ⊑ ActionAnonymizePersonalData ∪
ActionDeletePersonalData ∪ ActionNotifyDataSubject
Action ⊑ ActionAnonymizePersonalData
Action ⊑ ActionDeletePersonalData
Action ⊑ ActionNotifyDataSubject
ActionNotifyDataSubject ⊑
ActionNotifyDataSubjectByEmail ∪
ActionNotifyDataSubjectBySMS
ActionNotifyDataSubject ⊑
ActionNotifyDataSubjectByEmail
ActionNotifyDataSubject ⊑
ActionNotifyDataSubjectBySMS

**POLICY MATCHING APPLICATION AND USER INTERFACES**

“Policy Matching” application aims to match two policies. We use the “match” concept to compare two policies in order to see if one is less/more permissive than the other. Policy matching application takes policy documents as an input and produces a decision either access is allowed or denied as an output.

Policy matching application composed of two modules: Loader and Matcher. Loader allows users to upload policy documents, those specified in extended XACML. Loader parses uploaded documents and extracts related information by examining DataHandlingPreference or DataHandlingPolicy element and its children. Then extracted information will be stored in Policy Ontology.

Matcher module allows user to select two policies from Policy Ontology in order to match them. Matcher will conclude data consumer may access requested data if and only if data consumer has required authentications and agrees on specified obligations. To determine these features Matcher will send queries to OWL reasoner continuously.

Whenever Policy Matching application starts to run, application will request location of the Policy Ontology from the user. After locating Policy Ontology, the application will allow the user either upload a new policy to the system or match two policies those already exist. After locating the Policy Ontology, let us assume user selects uploading a new policy. To do so, he or she should specify the directory for the policy to be uploaded. Once policy document is specified the system will parse the document, extract related information and store it in Policy Ontology. If an error occurs user will be notified by the system otherwise extracted policy information will be presented to the user.

![Figure 5: Present Extracted Policy Information](image)

On the other hand, if user selects matching two policies, system will provide a list of existing policies to the user. After selecting two policies, one for data provider and one for data consumer, the system will send queries for each element in the policies to the reasoner to determine the access decision. If there is a mismatch, process will halt and system will notify user where is the mismatch located. Otherwise “Access permitted” result will return to the user.

**EXAMPLE**

Let us consider two policy document examples and observe the execution path. First step is representing these policies with OWL. Logical axioms for the first policy are the following:

```
Policy (Policy1)
AuthorizationSet (AuthorizationSet1)
hasAuthorizationSet (Policy1, AuthorizationSet1)
DownStreamUsageAuthorization (DownStreamUsageAuthorization1)
hasAuthorization (AuthorizationSet1, DownStreamUsageAuthorization1)
hasPermission (DownStreamUsageAuthorization1, false)
ObligationSet (ObligationSet1)
hasObligationSet (Policy1, ObligationSet1)
Obligation (Obligation1)
Obligation (Obligation2)
hasObligation (ObligationSet1, Obligation1)
hasObligation (ObligationSet1, Obligation2)
TriggerSet (TriggerSet1)
hasTriggerSet (TriggerSet1, TriggerSet1)
TriggerAtTime (TriggerAtTime1)
hasTrigger (TriggerSet1, TriggerAtTime1)
```

390
We believe that the Policy Matching application aims to provide access control in the data consumer’s policy. In principle instead of sending queries step by step, policy matching application could combine all queries into one and get the answer either yes or no. However when there is a mismatch in order to locate it, queries should be send partially.

We have to state the fact that system will have low performance when queries are being sent partially. Because reasoner will go through whole ontology for each partial query and this will cause system overload thus low performance.

**APPLICATION USAGE FIELDS**

As the amount of valuable data available on the Web grows, access control becomes extremely important to data providers and users. Access control, which means the users must fulfill certain conditions in order to access certain functionality, plays an important role in security based systems. Policy matching application aims to provide access control by using OWL.

Expressing policies by using OWL has several important advantages. Firstly most policy languages define constraints over classes of targets, objects, actions, time etc. which can be defined easily with OWL. Secondly OWL’s logic facilities enable to translate of policies expressed in OWL to other formalisms either for analysis or for execution. Moreover OWL reasoning capacity allows users manage and solve policy conflicts in an automated, error free way particularly in hierarchical issues and in multi-organizational environments.

We believe that the Policy Matching application would be useful in several fields such that social network and commercial applications etc. Next, we
will provide an example to indicate Policy Matching application usage.

**COMMERCIAL APPLICATION EXAMPLE**

Let us have a look at the website of CardScan in which users can create an online address book to access their contacts' information. In CardScan users can upload their personal information and contact information to the system and define who can access these data.

To be more clear, consider the following example: Suppose a data provider uses CardScan to upload his/her personal information and contact information. Moreover, he/she states that the user1 cannot access data while user2 may access to the provided data. So, user2 can access the data provider’s information and update his/her address book. In this scenario, we can assume that the actual data consumer is the CardScan system and according to the policy between the data provider and the system, the system may forward data to the third parties or not. Figure 6 illustrates this situation.

Moreover, the data provider may define some obligations over his/her data to limit the data consumer. Part of the data provider’s policy is shown in Figure 7.

From these two application examples, we can observe that Policy Matching application would be useful in several fields in which access control is a main issue. Its flexible structure enables it to adapt to the application domain easily and cover all necessary access control problems.

**CONCLUSION**

Semantic Web technologies open new opportunities to deal with a great number of problems. However, they do not concern about providing high performance for the time being. Also, to combine them with existing technologies can be a little difficult, because of poor documentation and incompatibility issues. We believe that these problems will be solved in near future since usage of Semantic Web technologies is increasing.

Although our work is complete enough to be used in simple policy specifications, there are still some issues need to be considered in more detail to enable cover complex policies. As a part of future work, firstly we are considering to extend Policy Ontology structure. Since it is domain dependent, a specific domain of interest has to be chosen. Thus new kind of triggers and/or actions shall be added. Furthermore, a subject and/or resource hierarchy can be defined in order to exploit OWL reasoner capacity more. On the other hand, our long term goal is to continue to investigate the Semantic Web services in relation with access control models and declarative policy languages.

**REFERENCES**


CardScan: [www.cardscan.net](http://www.cardscan.net)


PizzaOntology: http://owl.cs.manchester.ac.uk/tutorial/s/protegeowltutorial/

Project: http://www.primelife.eu/images/stories/h5.3.2-seconddesign/h5.3.2.html

RDF: www.w3.org/RDF/

SemanticWeb: http://semanticweb.org/wiki/Main_Page


Vladimir Kolovski, James Hendler and Bijan Parsia, Formalizing XACML Using Defeasible Description Logics: www.mindswap.org/~kolovski/xacml_tr.pdf