POLITECNICO DI MILANO

FACOLTÀ DI INGENERIA DELL' INFORMAZIONE

MASTER OF SCIENCE IN COMPUTER ENGINEERING

# CONTROLLING ACCESS TO DATA VIA OWL REASONING

*Supervisor:*

*Prof. Marco COLOMBETTI*

*Author:*

*Ebru UZUNDERE*

*737084*

2009-2010

# Table of Contents

# List of Figures

# List of Tables

# ABSTRACT

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.

With the advent of the Semantic Web, Semantic Web techniques, especially ontologies, allow to describe Web services with more machine understandable semantics, thus enabling to use logic and formal reasoning techniques for automatic analysis and verification of policies. Access control should have the ability to work synergistically across different organizations or services to provide security. There has been a great amount of attention to using Semantic Web techniques in above mentioned issues due to the fact that Semantic Web techniques fit very well to the nature of access control and policies.

In this work, firstly, we have presented 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. We believe that, this approach can be used to help developing security frameworks for dynamic environments which require an agreement before sharing data i.e. social networks.

## SOMMARIO

Poiché la quantità di dati preziosi a disposizione sul web cresce, il controllo degli accessi diventa estremamente importante per i fornitori di dati e per gli utenti. Il controllo degli accessi, ossia condizioni che gli utenti devono soddisfare per poter accedere a determinate funzionalità, svolge un ruolo importante nei sistemi basati sulla sicurezza.

Con l'avvento del Semantic Web, delle tecniche del Semantic Web, e in particolare le ontologie, é possibile descrivere i servizi Web con più macchina semantica comprensibile, permettendo così di usare la logica e le tecniche di ragionamento formale per l'analisi automatica e per la verifica delle regole. Il controllo degli accessi deve avere la capacità di lavorare in sinergia tra diverse organizzazioni o servizi per fornire sicurezza. C'è stata una grande attenzione riguardo all'utilizzo di tecniche di Semantic Web nei casi discussi precedentemente per il fatto che le tecniche di Web semantico si adattano molto bene alla natura del controllo e delle regole di accesso.

In questo lavoro, prima, abbiamo presentato una traduzione dei documenti di politiche di accesso, quelle scritte in esteso XACML in  OWL-DL e qunidi abbiamo descritto come quelle traduzioni possono essere utilizzate per l'analisi di politiche e il confronto, sfruttando le funzionalità di ragionatore di OWL. Crediamo che questo approccio può essere usato per aiutare lo sviluppo di strutture di sicurezza per ambienti dinamici, che richiedono un accordo prima di condividere i dati, cioe` reti sociali.

# Chapter 1

## INTRODUCTION

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. There have been several studies to develop access control models such that role based access control, attribute based access control and credential based access control models. Here is a summary of these models: [1]

- Attribute-based Access Control (ABAC) grants access, not based on the rights of user authentication, but based on attributes of the user. In ABAC, authorizations are associated with a set of rules expressed on measurable parameters, attributes, and permissions are granted to users who can prove compliance with these rules.

- Discretionary Access Control (DAC) is an access control model in which owner of an object decides who is allowed to access the object and what privileges they have. Limiting access to a file is an example for DAC model; it is the owner of the file who controls other users' accesses to the file. Only those users specified by the owner may have permissions to access to the file. DAC policy tends to be very flexible and is widely used in the commercial applications and government sectors.

- Mandatory Access Control (MAC) is another access control model in which access control is determined by the system, not the owner. For instance in military security, a data owner cannot change the classification of a document from *Top Secret* to *Secret.* MAC is the most popular NDAC (Non- Discretionary Access Control) policy.

- Role-based Access Control (RBAC) is another NDAC policy in which access control is determined by the system.In RBAC, individuals are assigned to some roles and access decisions are based on those roles. Access rights are grouped by role name, and the use of resources is limited to authorized individuals of the associated role. For instance, within a high school system, the permissions for professor role and for student role are different; each group can access specified resources those associated with their roles. RBAC can be effective for developing and enforcing enterprise-specific security policies.

- Credential-based Access Control (CBAC) is an authorization based access control rather than authentication based. A credential is used for proving qualification, such as proof of identity and/or proof of authority. For example, national identity cards are proofs of identity, plane tickets are proofs of authorization to have a particular flight in a particular seat, and driver's licenses are proofs of identity and of authorization to drive motor vehicles of a certain category.

In general these models use declarative, rule based languages which allow users to specify complicated policies. Current access control researches follow two paths: some studies focus on developing a powerful and expressive language for policies while others are focus on what can be done by using existing languages. In this scope, policy languages have been compared according to the functionalities such that:

- Policy Comparison: Check two policies, if both give the same decision result, access permit or deny, when they have the same input conditions.
- Policy Verification: Check if two policies satisfy a particular policy property.
- Policy Analysis: Check if a policy consistent or not.
- Policy Matching: By comparing two policies decide which one is less/more permissive than the other.

With the advent of the Semantic Web, Semantic Web techniques, especially ontologies, allow to describe Web services with more machine understandable semantics, thus enabling to use logic and formal reasoning techniques for automatic analysis and verification of policies. Access control should have the ability to work synergistically across different organizations or services to provide security. An access control policy can be taught as a class description which denotes a set of “acceptable” things by describing them. There has been a great amount of attention to using Semantic Web techniques in abovementioned issues due to the fact that Semantic Web techniques fit very well to the nature of access control and policies.

PrimeLife project [2], whose goal is managing privacy and identity in Europe, is an example of recent studies for access control issue. Within scope of the PrimeLife project, a new policy language has been developed. In our work, we have created an ontology called “Policy Ontology” to represent policies described in newly implemented XML alike language of the PrimeLife. Structure of that ontology will be discussed in chapter 4. Then we designed a “Policy Matching” application whose aim is matching two policies by using the Policy Ontology. Main features of the Policy matching application will be discusses in chapter 5.

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 will discuss Loader and Matcher components deeply in chapter 3 and chapter 5.

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.

The remainder of report is organized as follows: In chapter 2 we provide relevant background information and in chapter 3 we explain system design. In chapter 4 we describe structure of Policy Ontology and provide an example. In chapter 5 we specify Policy Matching application features with an example. In chapter 6 we provide some examples to show usage of implemented system and conclusions in chapter 7.

# Chapter 2

# BACKGROUND KNOWLEDGE

## 2.1 Access Control Issues

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.

The development of an access control system requires the definition of the regulations according to which access is to be controlled. The development process is usually carried out with a multi-phase approach based on the following concepts [3]:

- Security Policy: The set of rules that define the conditions under which an access may take place.
- Security Model: Formalization of the access control security policy and it's working.
- Security Mechanism: Definition of the low level (software and hardware) functions that implement the controls imposed by the policy and formally stated in the model.

To develop an access control system, following issues should be considered: security principles, policies and policy languages, and models proposed in the literature. For existing access control models please check chapter 1.

In principle each access control model implements a single specified policy. However a single policy cannot capture all the security requirements. To solve this problem, policies can be implemented as a part of the application code. But this solution is not optimum because it makes the verification, modification and enforcement of policy difficult.

Recent solutions have proposed policy languages those provide a single mechanism able to enforce multiple policies. Logic-based languages, for their expressive power and formal foundations, and also their flexibility and extensibility features represent a good candidate. XML based languages are another candidate since they have ability to interchange data between different applications, systems or organizations.

## 2.2 Overview of XACML

Policy matching application takes policy documents as an input and produces a decision either access is allowed or denied as an output. Input policy documents are described with a policy language which has been designed for the PrimeLife project. This language extends the XACML 3.0 [4] with a number of privacy enhancing and credential based features.

XACML, extensible access control markup language, released by OASIS is a XML-based language for describing and interchanging access control policies. It defines policy strategies through attribute matching mechanism and it can be used to define general access control requirements. XACML allows expressing both positive and negative authorization and the hierarchical role based access control model (RBAC). Moreover it enables the use of arbitrary attributes in policies. XACML defines both an architecture for evaluation of policies and a communication protocol for message interchange.

The root of all XACML policies is a Policy or PolicySet element. A PolicySet is a container that can hold other Policy or PolicySet elements. A Policy element represents a single access control policy, expressed through a set of Rules. Rules are the most basic element of XACML those actually take an access request as an input and yield a decision; Permit, Deny or Not-Applicable. Target element is used to determine if a Rule is applicable to an access request or not.  A Target is a set of simplified conditions for the Subject, Resource and Action. Figure 2.2.1 shows general structure of an XACML policy.

Some of the novel functionalities offered by XACML can be listed as below:

- Policy Combination: Different entities can define their policies on the same resource. When an access request on that source is submitted, the system takes into consideration all the applicable policies.

- Combining Algorithms: Policy or PolicySet may contain multiple policies or rules, each of which may evaluate to different access control decision. XACML supports different combining algorithms, Policy Combining Algorithms or Rule Combining Algorithms, to result multiple decisions into a single decision.

- Attribute-based Restrictions: XACML supports policies based on attributes associated with subjects and resources. XACML includes some built-in operators for comparing attribute values.

- Policy Distribution: XACML allows to a policy contain or refer to another policy.

- Obligations: XACML provides obligation actions those shall be enforced after the access decision has been taken.

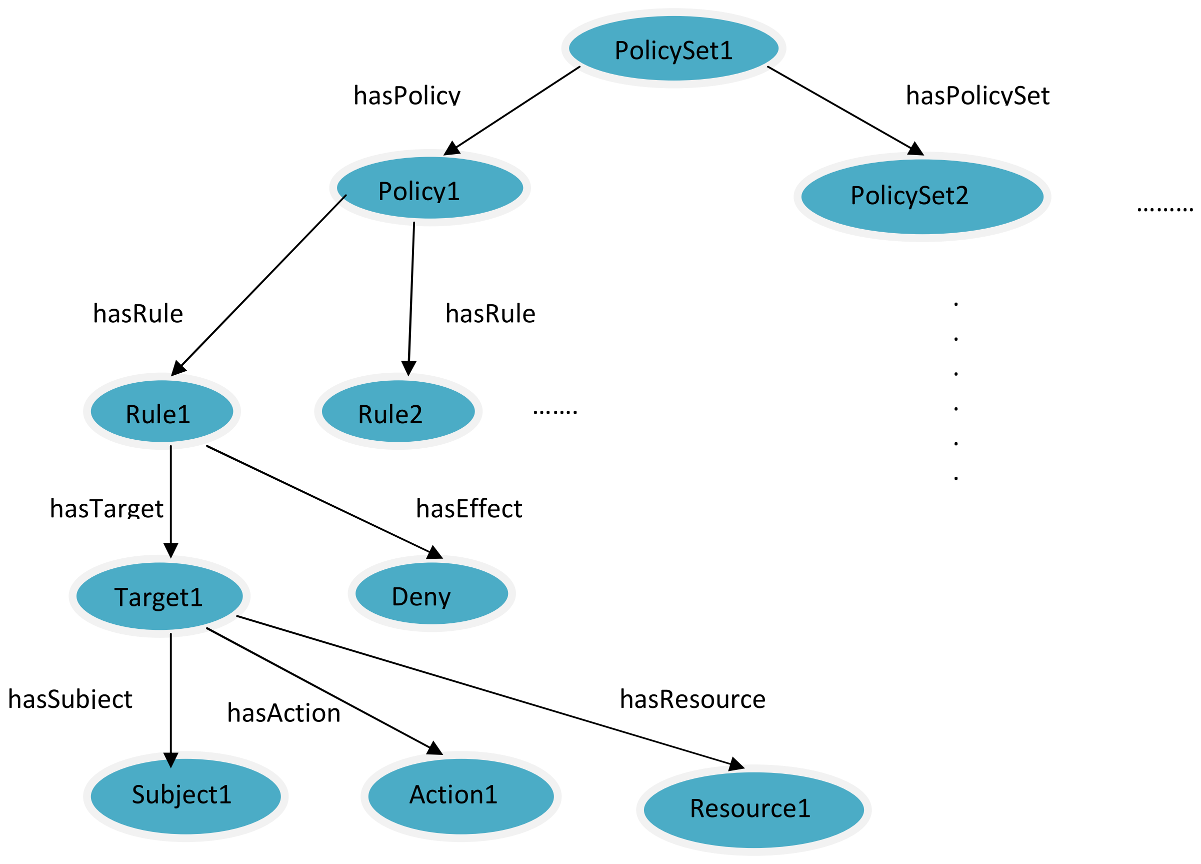


*Figure 2.2.1: Structure of an XACML Policy*

As an example of XACML policy, suppose that a hospital defines a high-level policy stating that “any user with role head physician can read the patient record for which he/she is designated as head physician” [2]. Figure 2.2.2 illustrates the XACML policy of this example. The policy has one rule with a target that requires a read action, a subject with role head physician and a condition that applies only if the subject is the head physician of the requested patient.

```
<Policy PolicyId="Pol1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0: rule-
combining-   algorithm:permit-overrides" . . . >
<Rule RuleId="ReadRule" Effect="Permit">
  <Target>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              head physician
          </AttributeValue>
          <AttributeDesignator
              Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
              AttributeId= "urn:oasis:names:tc:xacml:2.0:example:attribute:role"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </Match>
      </AllOf>
    </AnyOf>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              read
          </AttributeValue>
          <AttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#string"
              Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
              AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Condition>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
            Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:head-physicianID"/>
      <AttributeSelector RequestContextPath="/ctx:Request/ctx:Resource/ctx:
            ResourceContent/hospital:record/hospital:patient/hospital:
            patient-head-physicianID/text()"
            DataType="http://www.w3.org/2001/XMLSchema#string"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"/ >
  </Condition>
</Rule>
</Policy>
```

*Figure 2.2.2: Example of an XACML Policy*

## 2.3 Overview of Extended XACML Policy Language

Within scope of the PrimeLife project, XACML is extended. Figure 2.3.1 illustrates extended XACML model which has several newly added components. Extended XACML policy language is intended to be used:

- by a subject to specify the access restrictions to his/her personal information and how he/she wants this information to be treated afterwards
- by a subject to specify the access restrictions to the resources that he/she requests and how he/she is going to handle accessed data afterwards

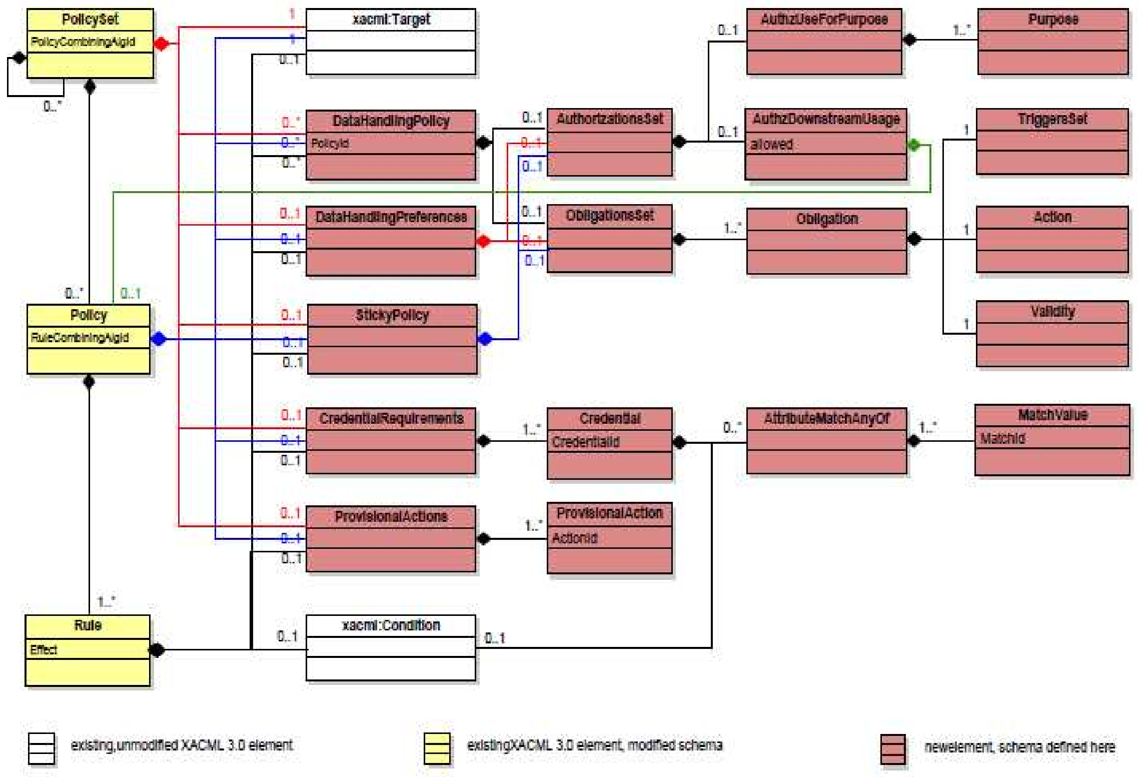


*Figure 2.3.1: Model of extended XACML*

As we see from Figure 2.3.1, extended XACML policy language modified PolicySet, Policy and Rule elements those already implemented in XACML 3.0. Similarly to XACML, in extended policy language each rule has an effect, either “Permit” or “Deny”. Rules are grouped together in policies and when a policy is evaluated, the rule combining algorithm of the policy determines the effect of the policy by examining the effects of the applicable rules. Policies are grouped together in policy sets like XACML 3.0 and the effect of a policy set is determined by the effects of the contained policies and the stated policy combining algorithm. Lastly, different policy sets can be further grouped together in parent policy sets.

The main components of a rule are: a target, a condition, credential requirements, provisional actions, data handling policies, and data handling preferences. Target and Condition elements are used in extended XACML without any modification. CredentialRequirements element describes the credentials that are required in order to access to the resource. ProvisionalActions element describes the actions that have to be performed in order to access to the requested resource.

If we name the individual whose personal data are collected as Data Provider, DataHandlingPreference element would involve preferences of data provider about how this data should be handled. Similarly if we name the individual who asks for an access to a specific personal data as Data Consumer, DataHandlingPolicy element would define intends of data consumer about how he/she is going to handle the data. Each rule, policy, or policy set can contain a number of DataHandlingPolicy or DataHandlingPreference element.

Each DataHandlingPreference and DataHandlingPolicy element may involve a set of authorizations and a set of obligations. For the time being, authorizations focus on two basic concepts: usage and downstream sharing. Usage restricts what data consumer can do with accessed data while downstream sharing defines under which conditions data can be shared with another data consumer. Obligations state what the data consumer shall do after accessing specified data. An obligation can be taught as action on an event if a specific condition is satisfied.

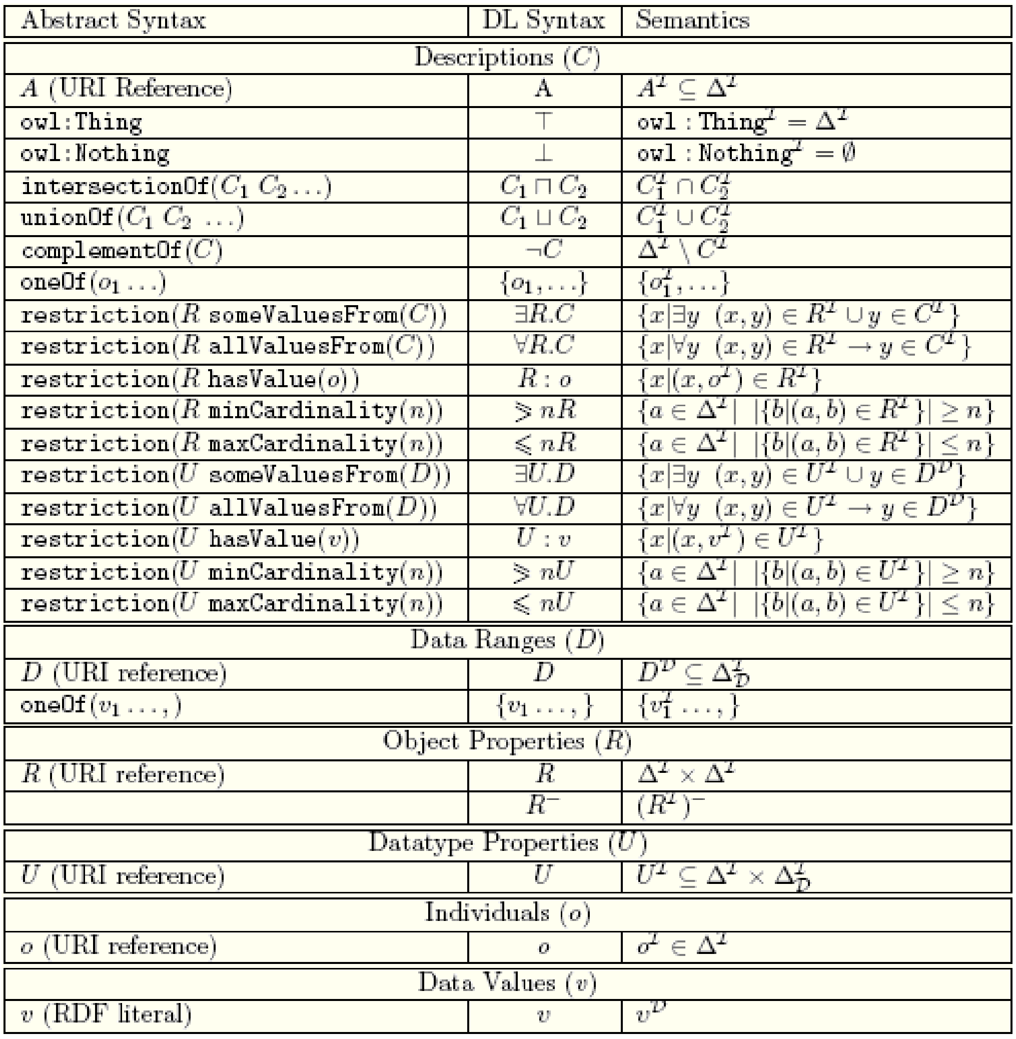
## 2.4 The Web Ontology Language OWL

The OWL, Web Ontology Language [5] is a family of knowledge representation languages based on Description Logic (DL) for authoring ontologies with a representation in RDF.

OWL supports the specification and usage of ontologies those provide a domain of interest in terms of concepts and relations according to which these concepts are related. Ontologies uses a DL knowledge base composed by individuals relevant to the domain of interest, classes of individuals, relations, properties and axioms that assert constraints over them.

OWL has three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL Lite is intended to support users primarily needing a classification hierarchy and simple constraints over ontologies. OWL DL was designed to provide the maximum expressiveness possible while retaining computational completeness, decidability, and the availability of practical reasoning algorithms. OWL Full is based on a different semantics from OWL Lite or OWL DL, and was designed to preserve some compatibility with RDF Schema.

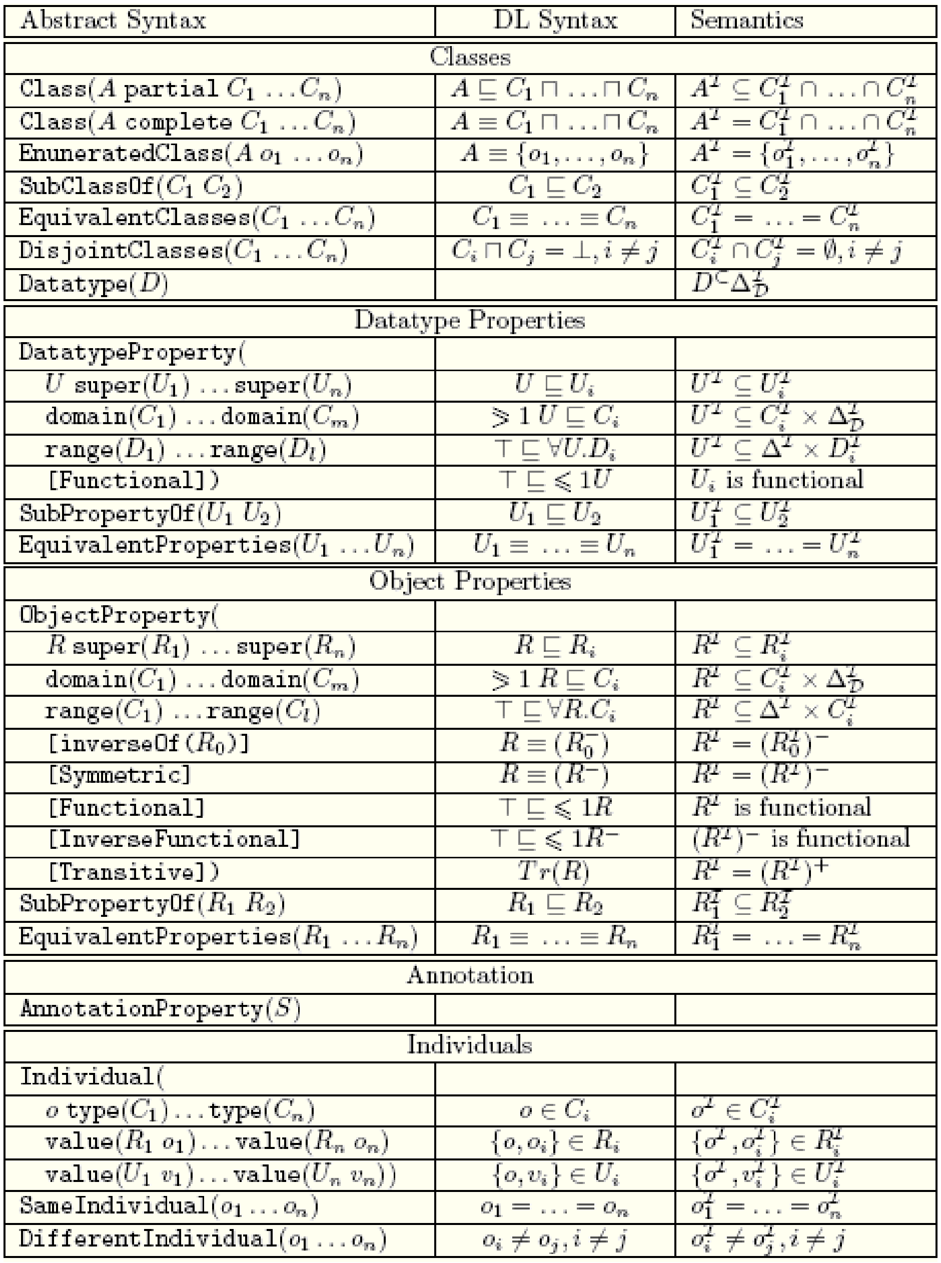
The OWL DL descriptions, data ranges, properties, individuals and data values syntax and semantics are summarized in the first table below, OWL DL axioms and facts are summarized in the second table below [6].



| Abstract Syntax | DL Syntax | Semantics |
|---|---|---|
| Descriptions ($C$) | | |
| $A$ (URI Reference) | A | $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ |
| `owl:Thing` | $\top$ | $\mathtt{owl:Thing}^{\mathcal{I}} = \Delta^{\mathcal{I}}$ |
| `owl:Nothing` | $\bot$ | $\mathtt{owl:Nothing}^{\mathcal{I}} = \emptyset$ |
| `intersectionOf`($C_1\ C_2 \ldots$) | $C_1 \sqcap C_2$ | $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ |
| `unionOf`($C_1\ C_2\ \ldots$) | $C_1 \sqcup C_2$ | $C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ |
| `complementOf`($C$) | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| `oneOf`($o_1 \ldots$) | $\{o_1, \ldots\}$ | $\{o_1^{\mathcal{I}}, \ldots\}$ |
| `restriction`($R$ `someValuesFrom`($C$)) | $\exists R.C$ | $\{x \mid \exists y\ \ (x,y) \in R^{\mathcal{I}} \cup y \in C^{\mathcal{I}}\}$ |
| `restriction`($R$ `allValuesFrom`($C$)) | $\forall R.C$ | $\{x \mid \forall y\ \ (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$ |
| `restriction`($R$ `hasValue`($o$)) | $R : o$ | $\{x \mid (x, o^{\mathcal{I}}) \in R^{\mathcal{I}}\}$ |
| `restriction`($R$ `minCardinality`($n$)) | $\geqslant nR$ | $\{a \in \Delta^{\mathcal{I}} \mid\ \ \lvert\{b \mid (a,b) \in R^{\mathcal{I}}\}\rvert \geq n\}$ |
| `restriction`($R$ `maxCardinality`($n$)) | $\leqslant nR$ | $\{a \in \Delta^{\mathcal{I}} \mid\ \ \lvert\{b \mid (a,b) \in R^{\mathcal{I}}\}\rvert \leq n\}$ |
| `restriction`($U$ `someValuesFrom`($D$)) | $\exists U.D$ | $\{x \mid \exists y\ \ (x,y) \in U^{\mathcal{I}} \cup y \in D^{\mathcal{D}}\}$ |
| `restriction`($U$ `allValuesFrom`($D$)) | $\forall U.D$ | $\{x \mid \forall y\ \ (x,y) \in U^{\mathcal{I}} \rightarrow y \in D^{\mathcal{D}}\}$ |
| `restriction`($U$ `hasValue`($v$)) | $U : v$ | $\{x \mid (x, v^{\mathcal{I}}) \in U^{\mathcal{I}}\}$ |
| `restriction`($U$ `minCardinality`($n$)) | $\geqslant nU$ | $\{a \in \Delta^{\mathcal{I}} \mid\ \ \lvert\{b \mid (a,b) \in U^{\mathcal{I}}\}\rvert \geq n\}$ |
| `restriction`($U$ `maxCardinality`($n$)) | $\leqslant nU$ | $\{a \in \Delta^{\mathcal{I}} \mid\ \ \lvert\{b \mid (a,b) \in U^{\mathcal{I}}\}\rvert \leq n\}$ |
| Data Ranges ($D$) | | |
| $D$ (URI reference) | $D$ | $D^{\mathcal{D}} \subseteq \Delta_{\mathcal{D}}^{\mathcal{I}}$ |
| `oneOf`($v_1 \ldots,$) | $\{v_1 \ldots,\}$ | $\{v_1^{\mathcal{I}} \ldots,\}$ |
| Object Properties ($R$) | | |
| $R$ (URI reference) | $R$ | $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ |
| | $R^{-}$ | $(R^{\mathcal{I}})^{-}$ |
| Datatype Properties ($U$) | | |
| $U$ (URI reference) | $U$ | $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}}$ |
| Individuals ($o$) | | |
| $o$ (URI reference) | $o$ | $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ |
| Data Values ($v$) | | |
| $v$ (RDF literal) | $v$ | $v^{\mathcal{D}}$ |

*Table 2.4.1: OWL DL Syntax and Semantics*

A concept from DL is referred to as a class in OWL and a role from DL is referred to as a property in OWL. In OWL "Classes" denote set of objects while "Individuals" denote single objects. "Object properties" denote binary relationships between objects and "Datatype properties" denote binary relationships between objects and data.



| Abstract Syntax | DL Syntax | Semantics |
|---|---|---|
| Classes | | |
| Class($A$ partial $C_1 \ldots C_n$) | $A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$ | $A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$ |
| Class($A$ complete $C_1 \ldots C_n$) | $A \equiv C_1 \sqcap \ldots \sqcap C_n$ | $A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$ |
| EnuneratedClass($A\ o_1 \ldots o_n$) | $A \equiv \{o_1, \ldots, o_n\}$ | $A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$ |
| SubClassOf($C_1\ C_2$) | $C_1 \sqsubseteq C_2$ | $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ |
| EquivalentClasses($C_1 \ldots C_n$) | $C_1 \equiv \ldots \equiv C_n$ | $C_1^{\mathcal{I}} = \ldots = C_n^{\mathcal{I}}$ |
| DisjointClasses($C_1 \ldots C_n$) | $C_i \sqcap C_j = \bot, i \neq j$ | $C_i^{\mathcal{I}} \cap C_j^{\mathcal{I}} = \emptyset, i \neq j$ |
| Datatype($D$) | | $D^{\subseteq}\Delta_{\mathcal{D}}^{\mathcal{I}}$ |
| Datatype Properties | | |
| DatatypeProperty( | | |
| $U$ super($U_1$) $\ldots$ super($U_n$) | $U \sqsubseteq U_i$ | $U^{\mathcal{I}} \subseteq U_i^{\mathcal{I}}$ |
| domain($C_1$) $\ldots$ domain($C_m$) | $\geqslant 1\, U \sqsubseteq C_i$ | $U^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}}$ |
| range($D_1$) $\ldots$ range($D_l$) | $\top \sqsubseteq \forall U.D_i$ | $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times D_i^{\mathcal{I}}$ |
| [Functional]) | $\top \sqsubseteq \leqslant 1U$ | $U_i$ is functional |
| SubPropertyOf($U_1\ U_2$) | $U_1 \sqsubseteq U_2$ | $U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$ |
| EquivalentProperties($U_1 \ldots U_n$) | $U_1 \equiv \ldots \equiv U_n$ | $U_1^{\mathcal{I}} = \ldots = U_n^{\mathcal{I}}$ |
| Object Properties | | |
| ObjectProperty( | | |
| $R$ super($R_1$) $\ldots$ super($R_n$) | $R \sqsubseteq R_i$ | $R^{\mathcal{I}} \subseteq R_i^{\mathcal{I}}$ |
| domain($C_1$) $\ldots$ domain($C_m$) | $\geqslant 1\, R \sqsubseteq C_i$ | $R^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}}$ |
| range($C_1$) $\ldots$ range($C_l$) | $\top \sqsubseteq \forall R.C_i$ | $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C_i^{\mathcal{I}}$ |
| [inverseOf($R_0$)] | $R \equiv (R_0^-)$ | $R^{\mathcal{I}} = (R_0^{\mathcal{I}})^-$ |
| [Symmetric] | $R \equiv (R^-)$ | $R^{\mathcal{I}} = (R^{\mathcal{I}})^-$ |
| [Functional] | $\top \sqsubseteq \leqslant 1R$ | $R^{\mathcal{I}}$ is functional |
| [InverseFunctional] | $\top \sqsubseteq \leqslant 1R^-$ | $(R^{\mathcal{I}})^-$ is functional |
| [Transitive]) | $Tr(R)$ | $R^{\mathcal{I}} = (R^{\mathcal{I}})^+$ |
| SubPropertyOf($R_1\ R_2$) | $R_1 \sqsubseteq R_2$ | $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ |
| EquivalentProperties($R_1 \ldots R_n$) | $R_1 \equiv \ldots \equiv R_n$ | $R_1^{\mathcal{I}} = \ldots = R_n^{\mathcal{I}}$ |
| Annotation | | |
| AnnotationProperty($S$) | | |
| Individuals | | |
| Individual( | | |
| $o$ type($C_1$) $\ldots$ type($C_n$) | $o \in C_i$ | $o^{\mathcal{I}} \in C_i^{\mathcal{I}}$ |
| value($R_1\ o_1$) $\ldots$ value($R_n\ o_n$) | $\{o, o_i\} \in R_i$ | $\{o^{\mathcal{I}}, o_i^{\mathcal{I}}\} \in R_i^{\mathcal{I}}$ |
| value($U_1\ v_1$) $\ldots$ value($U_n\ v_n$)) | $\{o, v_i\} \in U_i$ | $\{o^{\mathcal{I}}, v_i^{\mathcal{I}}\} \in U_i^{\mathcal{I}}$ |
| SameIndividual($o_1 \ldots o_n$) | $o_1 = \ldots = o_n$ | $o_1^{\mathcal{I}} = \ldots = o_n^{\mathcal{I}}$ |
| DifferentIndividual($o_1 \ldots o_n$) | $o_i \neq o_j, i \neq j$ | $o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, i \neq j$ |

*Table 2.4.2: OWL DL Axioms and Facts*

Part of Pizza Ontology [7] may be presented in OWL as shown in Figure 2.4.1. This example can be expressed in natural language as following: "Pizza has PizzaBase as its base; Pizza is disjoint with PizzaBase; NonVegetarianPizza is exactly Pizza that is not VegetarianPizza; isIngredientOf is a transitive property; isIngredientOf is inverse of hasIngredient".

The example can be expressed in the description logic syntax as follows:

Pizza $\subseteq \exists$ hasBase. PizzaBase   (Pizza has PizzaBase as its base)

Pizza $\cap$ PizzaBase $\equiv \perp$ (Pizza is disjoint with PizzaBase)

NonVegeterianPizza $\equiv$ Pizza $\cap \neg$ VegeterianPizza (NonVegetarianPizza is exactly Pizza that is not VegetarianPizza)

Tr (isIngredientOf) (isIngredientOf is a transitive property)

isIngredientOf $\equiv$ hasIngredient $^{-}$ (isIngredientOf is inverse of hasIngredient)

```
Namespace (p = <http://example.com/pizzas.owl#>)
Ontology (<http://example.com/pizzas.owl#>
  Class (p: Pizza partial
     Restriction (p: hasBase someValuesFrom(p:PizzaBase)))
  DisjointClasses (p: Pizza p: PizzaBase)
  Class (p: NonVegetarianPizza complete
     intersectionOf (p:Pizza complementOf(p:VegetarianPizza)))
  ObjectProperty (p: isIngredientOf Transitive
     inverseOf (p:hasIngredient))
)
```

*Figure 2.4.1: Part of Pizza Ontology*

# Chapter 3

## SYSTEM DESIGN

“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. 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 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 provider’s policy.

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 3.1 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.

As we shall see in following chapters, 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.

Thus obligations are compared as follows: An obligation in data consumer policy is equally or less permissive than an obligation in data provider policy if and only if triggers, action and validity in data consumer side are equally or less permissive.

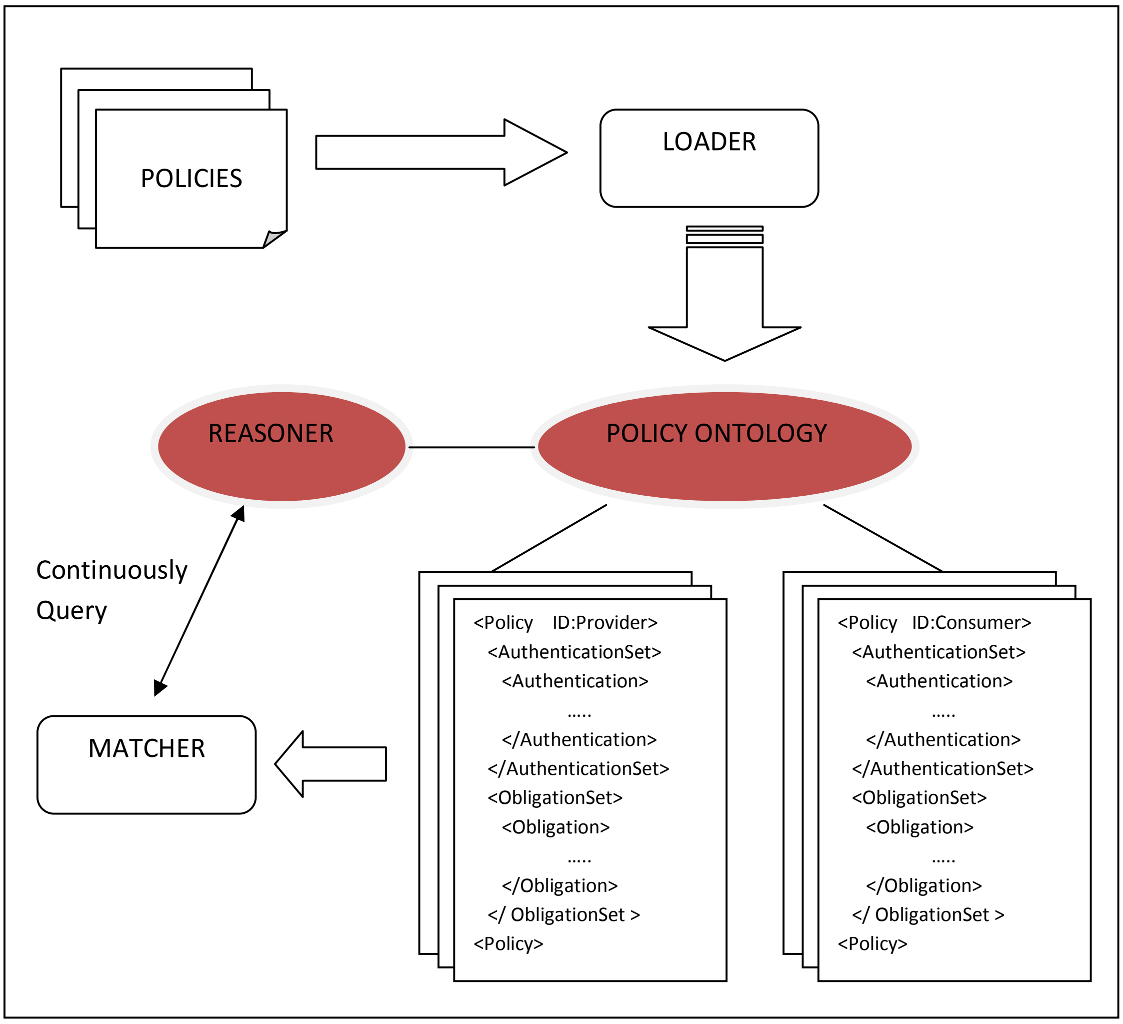


*Figure 3.1: System Model*

Validities are basically compared based on their start and end date. Start date of validity in data consumer side should be less or equal than validity in data provider side while end date should be great or equal.

Actions are compared based on their parameters. However hierarchical actions, such as ActionNotifyByEmail ⊆ ActionNotifyUser, are compared based on following rules: (Assume C is data consumer policy while P is data provider preferences and ⊴ symbol stands for equally or less permissive)

- If data provider and consumer both have “ActionNotifyUser” then we will compare usernames.
  C: ActionNotifyUser ⊴ P: ActionNotifyUser ⇔ (C.userName ==P.userName)

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

  C: ActionNotifyByEmail ⊴ 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⊴R: ActionNotifyUser ⇔ (C.userName == P.userName)

Triggers are compared based on their parameters as well. However hierarchical triggers, such as TriggerPersonalDataAccessedForPurpose ⊆ TriggerPersonalDataAccessed, are compared based on following rules:

- If data provider and consumer both have “TriggerPersonalDataAccessed” then compare data.
  C: TriggerPersonalDataAccessed ⊴ P: TriggerPersonalDataAccessed ⇔
  (C.dataRef ==P.dataRef )

- If data provider and consumer both have “TriggerPersonalDataAccessedForPurpose” then compare data and purposes.
  C: TriggerPersonalDataAccessedForPurpose ⊴
  P: TriggerPersonalDataAccessedForPurpose ⇔
  ((C.dataRef == P.dataRef ) ∧ (C.purpose == P.purpose ) )

- If data consumer has “TriggerPersonalDataAccessed” and data provider has “TriggerPersonalDataAccessedForPurpose” then compare data.

C: TriggerPersonalDataAccessed ⊴ P: TriggerPersonalDataAccessedForPurpose ⇔ (C.dataRef == P.dataRef )

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 ⊴ symbol stand for equally or less permissive, logical formulas about reasoning process can be formalized as follows:

- C:Policy ⊴ P:Policy ⇔ ( ( C.authorizations ⊴ P.authorizations ) ∧ (C.obligations ⊴ 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 ⊴ P:ListAuthorizations ⇔ ∀ (i ε C) : ∃ (j ε P) where (i ⊴ 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 ⊴ P:ListObligations ⇔ ∀ (j ε P) : ∃ (i ε C) where (i ⊴ 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 provider's obligation set, there is a less permissive obligation in the data consumer's obligation set.

- C:Obligation ⊴ P:Obligation ⇔ ( ( (C.action ⊴ P.action) ∧ (C.triggers ⊴ P.triggers) ) ∧ (C.validity ⊴ 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 ε P) : ∃ (a ε 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.

- $C{:}Validity \trianglelefteq P{:}Validity \Leftrightarrow ( (C.start \leq P.start) \wedge (C.end \geq P.end))$

  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.

# Chapter 4

## POLICY ONTOLOGY

Semantic Web [8] vision was first articulated by World Wide Web Consortium (W3C) director Tim Berners Lee as an extension of existing web. It describes methods and technologies to understand and reason about knowledge and data on the World Wide Web. Under the supervisory of the W3C, a set of languages, protocols and technologies have been developed to realize this vision and to support the evolution of the concepts and technology those enable to carry out this vision.

The current set of W3C standards are based on RDF [9], a language that originally designed as a metadata data model and has become used to provide a basic capability of specifying graphs with a simple interpretation as a semantic network and serializing them in XML and some other Web systems.  The OWL, Web Ontology Language is a family of knowledge representation languages based on Description Logic (DL) for authoring ontologies with a representation in RDF.

OWL has three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL Lite is intended to support users primarily needing a classification hierarchy and simple constraints over ontologies. OWL DL was designed to provide the maximum expressiveness possible while retaining computational completeness, decidability, and the availability of practical reasoning algorithms. OWL Full is based on a different semantics from OWL Lite or OWL DL, and was designed to preserve some compatibility with RDF Schema.

OWL supports the specification and usage of ontologies those provide a domain of interest in terms of concepts and relations according to which these concepts are related. Ontologies uses a DL knowledge base composed by individuals relevant to the domain of interest, classes of individuals, relations, properties and axioms that assert constraints over them.

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.

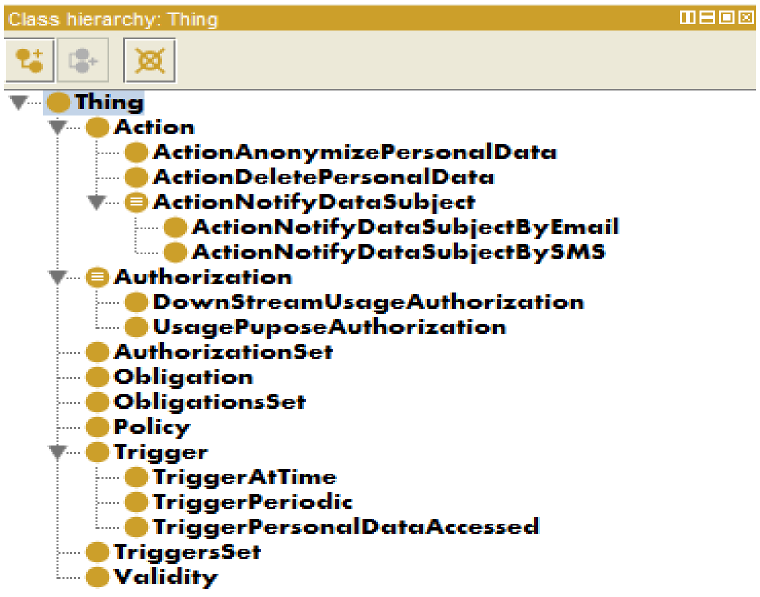
For reasoning process Hermit [10] has been used. Given an OWL file, Hermit can determine whether or not the ontology is consistent, identify subsumption relationships between classes, and much more. Hermit implements a novel hyper tableau reasoning algorithm which provides more efficient reasoning than any previously-known algorithm.

## 4.1 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 4.1.1 and Figure 4.1.2 shows class hierarchy in Policy Ontology.

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. "UsagePuposeAuthorization" class states what the purpose of using the data is and the purpose is specified by "hasPurpose" data property.



- Thing
  - Action
    - ActionAnonymizePersonalData
    - ActionDeletePersonalData
    - ActionNotifyDataSubject
      - ActionNotifyDataSubjectByEmail
      - ActionNotifyDataSubjectBySMS
  - Authorization
    - DownStreamUsageAuthorization
    - UsagePuposeAuthorization
  - AuthorizationSet
  - Obligation
  - ObligationsSet
  - Policy
  - Trigger
    - TriggerAtTime
    - TriggerPeriodic
    - TriggerPersonalDataAccessed
  - TriggersSet
  - Validity

*Figure 4.1.1: Class Hierarchy*

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.

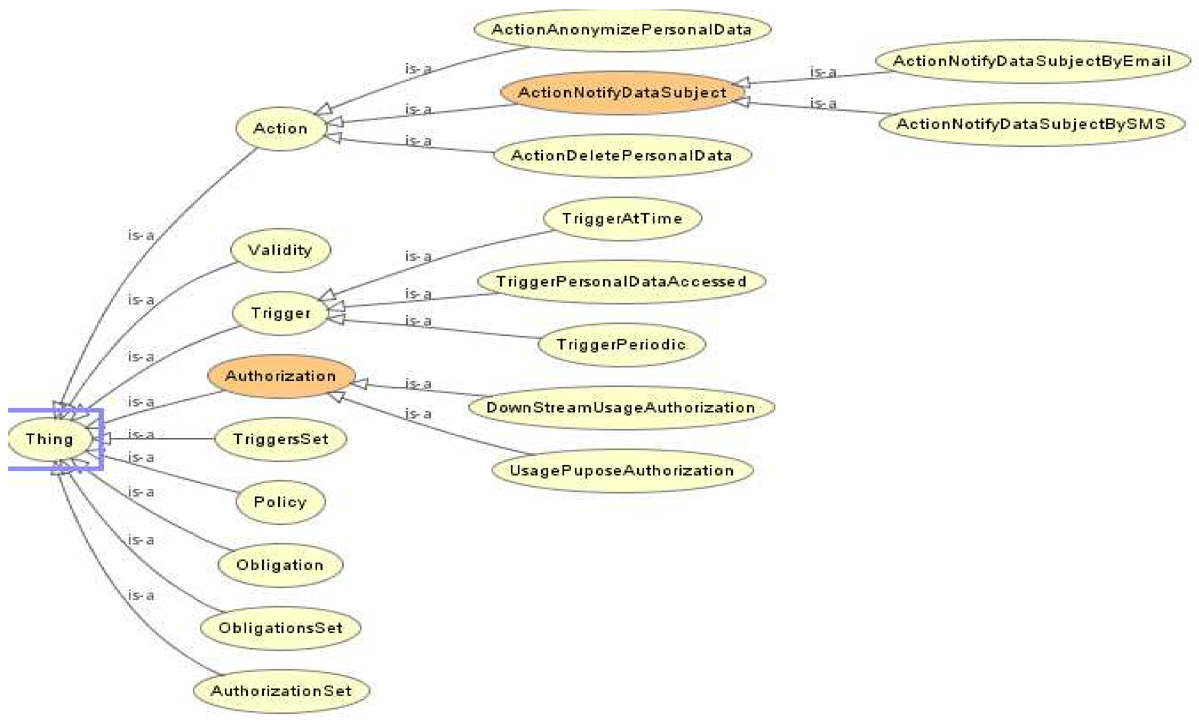


*Figure 4.1.2: Class Hierarchy in Graphical Representation*

Action class is divided into subclasses as well. “ActionDeletePersonalData” deletes a specific piece of information. It has data property “hasData” which is reference to the personal data to delete. “ActionAnonymizePersonalData” anonymizes a specific piece of information and owns “hasData” data property which is reference to the personal data to anonymize.

“ActionNotifyDataSubject” notifies data provider subject when triggered. It has “hasMedia” the media used to notify the user (e-mail, SMS, etc.) and “hasAddress” the corresponding address (e-mail address, phone number, etc.) as data properties. “ActionNotifyDataSubjectByEmail” and “ActionNotifyDataSubjectBySMS” are special case of this class.

Validity class represents obligation’s validity time. It owns “hasStart” and “hasEnd” data properties which state start and end date of validity. Class relationships, object properties, can be seen from bellowed figure.

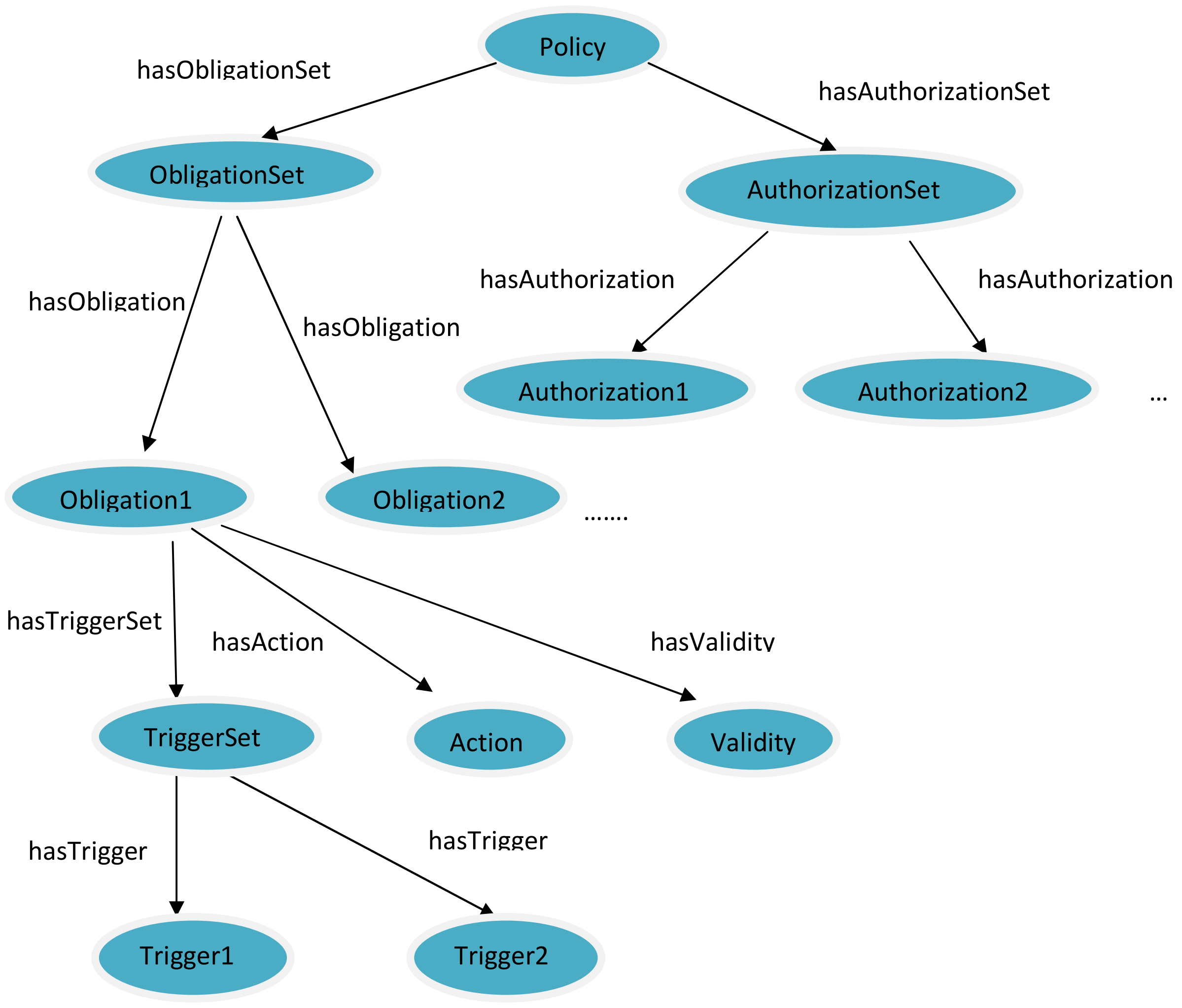


*Figure 4.1.3: Object Properties*

## 4.2 Example

Let us consider a policy example shown in Figure 4.2.1 and formalize it with abovementioned ontology.

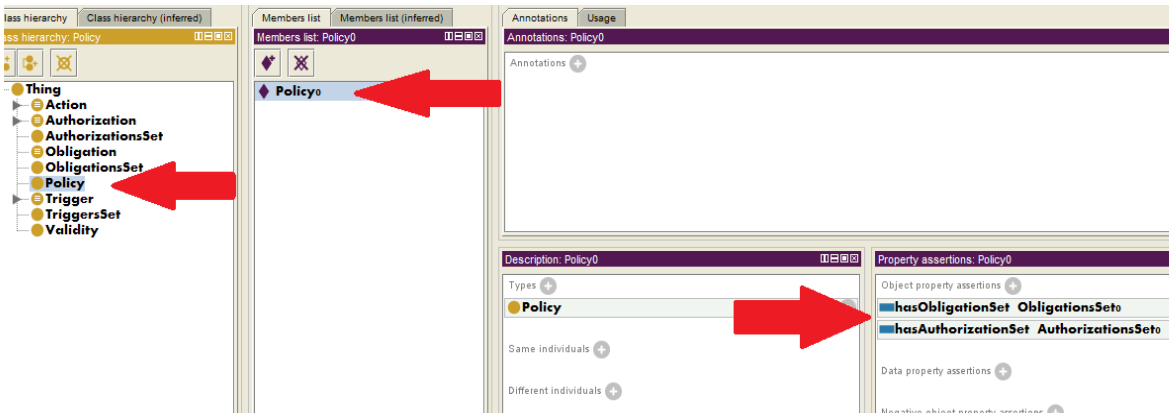
```
<? xml version="1.0" encoding="utf-8"?>
<Policy>
        <AuthorizationsSet>
                <UsagePuposeAuthorization>
                        <purpose> admin</purpose>
                 </UsagePuposeAuthorization>
        </AuthorizationsSet>
        <ObligationsSet>
                <Obligation Id="1">
                        <TriggersSet>
                                <TriggerAtTime>
                                        <start> 2010-01-01T00:00:00 </start>
                                        <maxDelay> P365D </maxDelay>
                                </TriggerAtTime>
                        </TriggersSet>
                        <Action>
                                <ActionDeletePersonalData>
                                        <personalData> ref to personal data </personalData>
                                </ActionDeletePersonalData>
                        </Action>
                        <Validity>
                                <start> 2009-10-14T00:00:00 </start>
                                <end> 2010-10-14T00:00:00 </end>
                        </Validity>
                </Obligation>
        </ObligationsSet>
</Policy>
```

*Figure 4.2.1: Policy Example*

To represent policy example in our ontology firstly we have to create an instance of Policy class, Policy0. Policy0 has two object properties called “hasAuthorizationSet” which states that

Policy0 has an authorization set called AuthorizationsSet0 and hasObligationSet” which states that Policy0 has an obligation set called ObligationsSet0.

> Policy (Policy0)
> AuthorizationSet (AuthorizationSet0)
> hasAuthorizationSet (Policy0, AuthorizationsSet0)
> ObligationSet (ObligationSet0)
> hasObligationSet (Policy0,ObligationsSet0)



*4.2.2: Policy Instance*

AuthorizationSet0 is an instance of AuthorizationSet class. This is consistent with the fact that “hasAuthorizationSet” property’s domain is Policy class and range is AuthorizationsSet class.

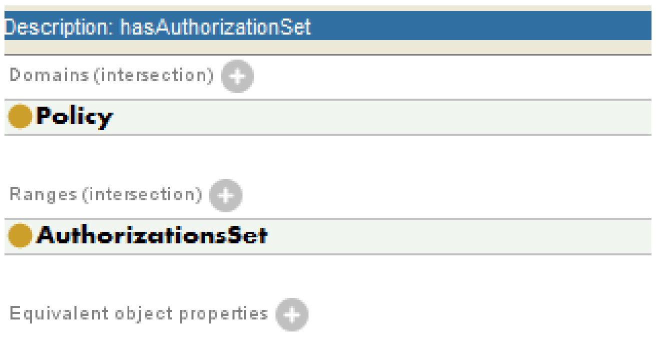
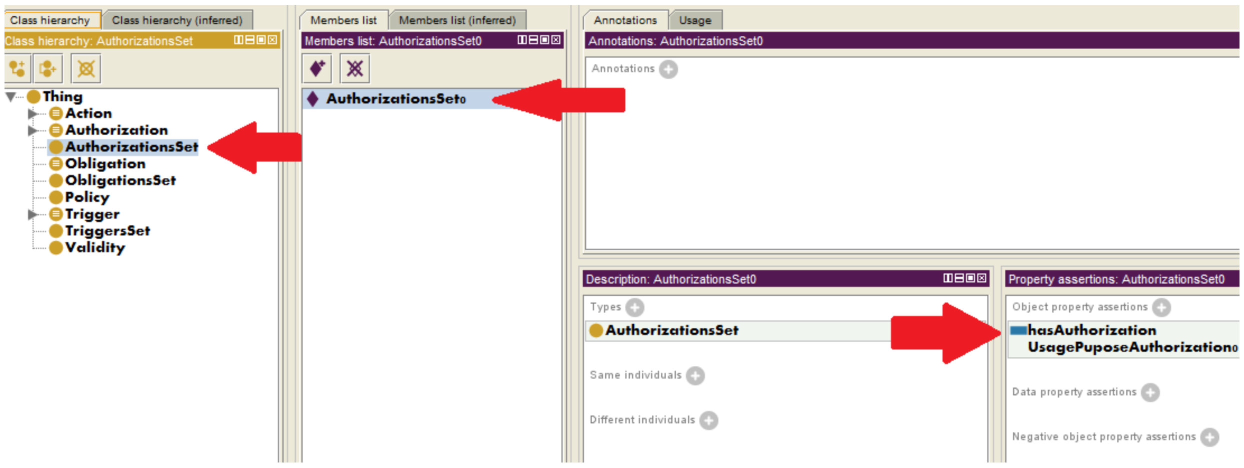


*Figure 4.2.3: hasAuthorizationSet Object Property Domains and Ranges*

AuthorizationSet0 has an object property called “hasAuthorization” whose domain is AuthorizationSet and range is Authorization. From Figure 4.2.4 it can be observed that AuthorizationSet0 has one authorization called UsagePurposeAuthorization0.

UsagePurposeAuthorization0 (UsagePurposeAuthorization0)
hasAuthorization (AuthorizationSet0, UsagePurposeAuthorization0)


Class hierarchy
Class hierarchy (inferred)
Class hierarchy: AuthorizationsSet
Members list
Members list (inferred)
Members list: AuthorizationsSet0
Annotations
Usage
Annotations: AuthorizationsSet0
Annotations
Description: AuthorizationsSet0
Types
Same individuals
Different individuals
Property assertions: AuthorizationsSet0
Object property assertions
Data property assertions
Negative object property assertions

Thing
Action
Authorization
AuthorizationsSet
Obligation
ObligationsSet
Policy
Trigger
TriggersSet
Validity

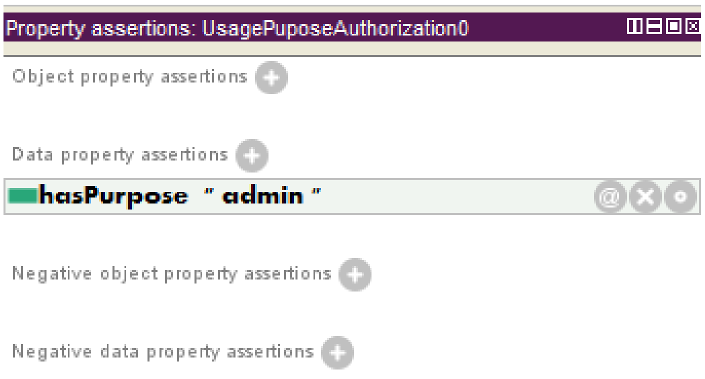
AuthorizationsSet0

AuthorizationsSet

hasAuthorization UsagePuposeAuthorization0

Figure 4.2.4: AuthorizationSet Instance

When we look UsagePurposeAuthorization0 instance, it can be observed that it has a data property called hasPurpose whose value is admin.


Property assertions: UsagePuposeAuthorization0

Object property assertions

Data property assertions

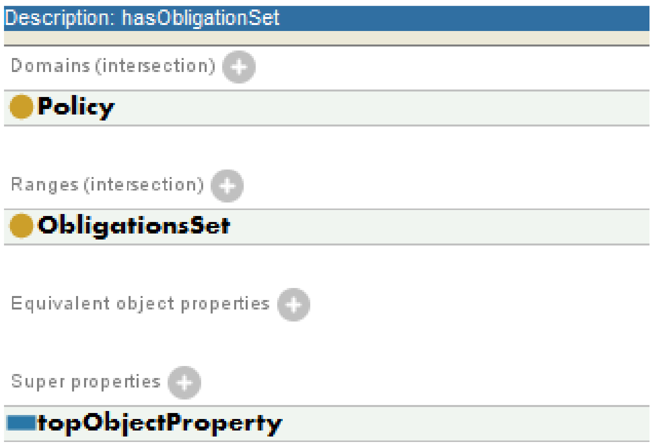
hasPurpose "admin"

Negative object property assertions

Negative data property assertions

Figure 4.2.5: Data Property of UsagePurposeAuthorization0 Instance

ObligationSet0 is an instance of ObligationSet class. This is consistent with the fact that “hasObligationSet” property domain is Policy class and range is ObligationSet class.


Description: hasObligationSet

Domains (intersection)

Policy

Ranges (intersection)

ObligationsSet

Equivalent object properties

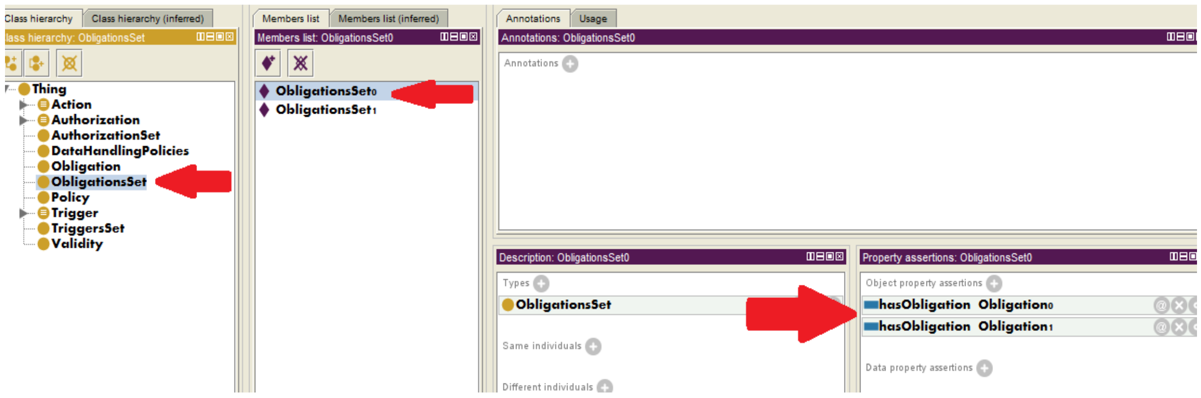
Super properties

topObjectProperty

*Figure 4.2.6: hasObligationSet Object Property Domains and Ranges*

ObligationSet0 has an object property called "*hasObligation*" whose domain is ObligationsSet and range is Obligation. From Figure 4.2.7 it can be observed that ObligationSet0 has two obligations: Obligation0 and Obligation1.

Obligation (Obligation0)
Obligation (Obligation1)
hasObligation (ObligationSet0,Obligation0)
hasObligation (ObligationSet0,Obligation1)


Class hierarchy | Class hierarchy (inferred)

lass hierarchy: ObligationsSet

- Thing
  - Action
  - Authorization
  - AuthorizationSet
  - DataHandlingPolicies
  - Obligation
  - ObligationsSet
  - Policy
  - Trigger
  - TriggersSet
  - Validity

Members list | Members list (inferred)

Members list: ObligationsSet0

- ObligationsSet0
- ObligationsSet1

Annotations | Usage

Annotations: ObligationsSet0

Annotations

Description: ObligationsSet0

Types

ObligationsSet

Same individuals

Different individuals

Property assertions: ObligationsSet0

Object property assertions

hasObligation Obligation0

hasObligation Obligation1

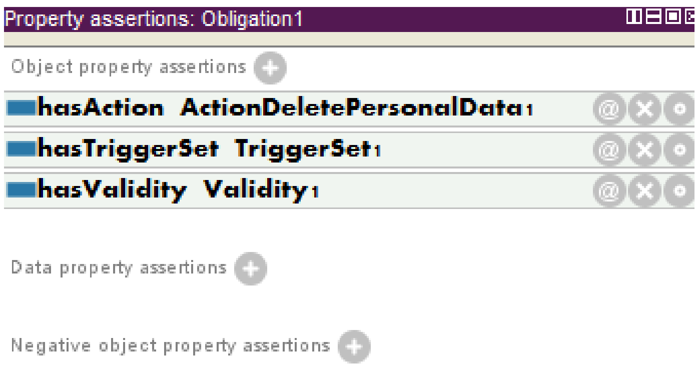
Data property assertions

*Figure 4.2.7: ObligationSet Instance*

Let have a look at Obligation1 instance. It belongs to Obligation class and it has three object properties: "hasTriggerSet", "hasAction" and "hasValidity".

ActionDeletePersonalData (ActionDeletePersonalData1)
TriggerSet (TriggerSet1)

Validity (Validity1)
hasAction (Obligation1, ActionDeletePersonalData1)
hasTriggerSet (Obligation1,TriggerSet1)
hasValidity (Obligation1,Validity1)



Property assertions: Obligation1

Object property assertions

hasAction ActionDeletePersonalData1
hasTriggerSet TriggerSet1
hasValidity Validity1

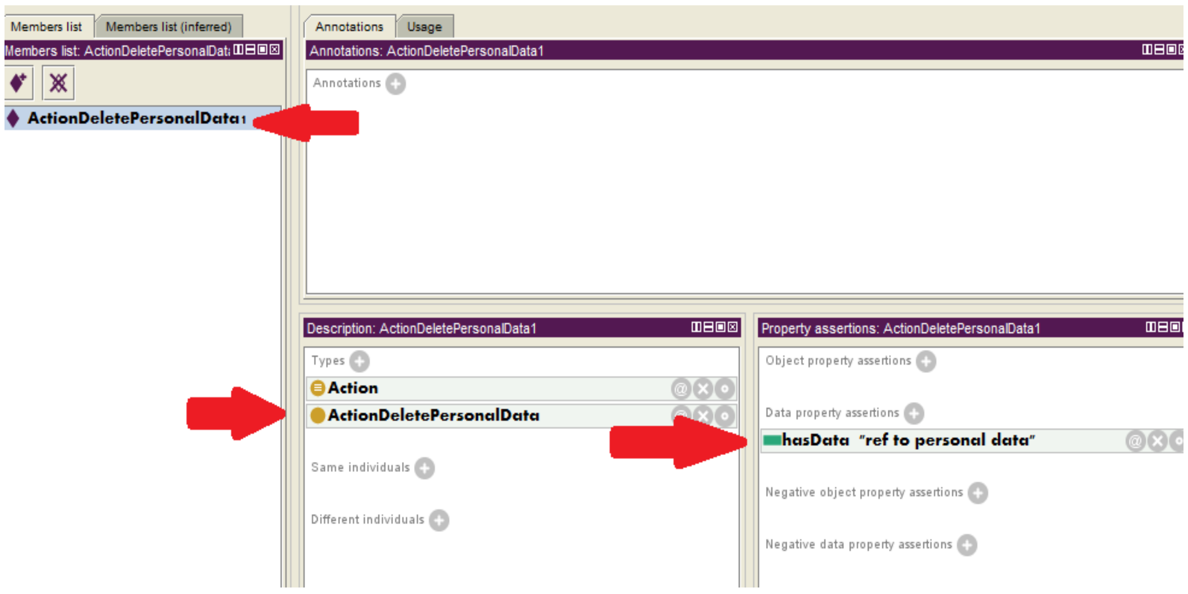
Data property assertions

Negative object property assertions

*Figure 4.2.8: Object Properties of Obligation1 Instance*

"hasAction" object property's domain is Obligation class and range is Action class. When we check ActionDeletePersonalData1 instance it can be observed that it has a data property called "hasData" whose value is "ref to personal data".

ActionDeletePersonalData1: hasData: "ref to personal data"^^string



Members list | Members list (inferred)

Members list: ActionDeletePersonalData1

ActionDeletePersonalData1

Annotations | Usage

Annotations: ActionDeletePersonalData1

Annotations

Description: ActionDeletePersonalData1

Types
Action
ActionDeletePersonalData

Same individuals

Different individuals

Property assertions: ActionDeletePersonalData1

Object property assertions

Data property assertions
hasData "ref to personal data"

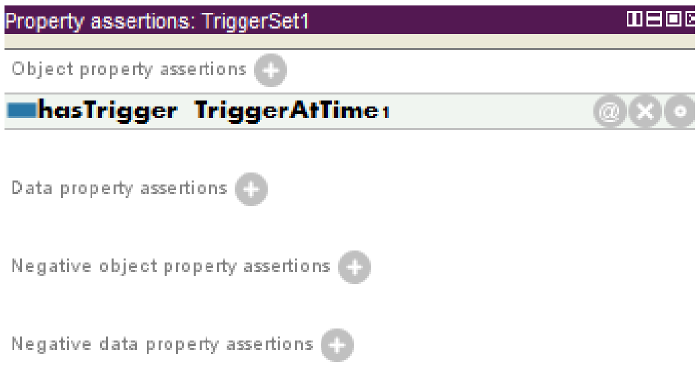
Negative object property assertions

Negative data property assertions

*Figure 4.2.9: ActionDeletePersonalData Instance*

“hasTriggerSet” object property’s domain is Obligation class and range is TriggerSet class. When we check TriggerSet1 instance we see that it has an object property called “hasTrigger”.

TriggerAtTime (TriggerAtTime1)
hasTrigger (TriggerSet1, TriggerAtTime1)


Property assertions: TriggerSet1

Object property assertions

hasTrigger TriggerAtTime1

Data property assertions

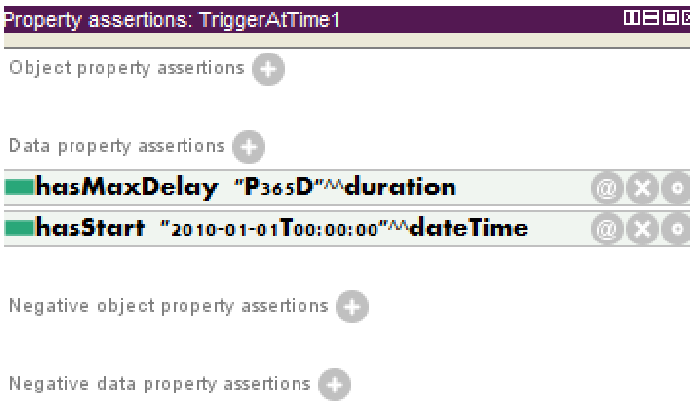
Negative object property assertions

Negative data property assertions

*Figure 4.2.10: Object Properties of TriggerSet1 Instance*

“hasTrigger” object property’s domain is TriggerSet class and range is Trigger class. When we check TriggerAtTime1 instance it can be observed that it has two data properties: “*hasMaxDelay*” and “*hasStart*”.

TriggerAtTime1: hasMaxDelay: “P365D”^^duration
TriggerAtTime1: hasStart: “2010-01-01T00:00:00”^^dateTime


Property assertions: TriggerAtTime1

Object property assertions

Data property assertions

hasMaxDelay "P365D"^^duration

hasStart "2010-01-01T00:00:00"^^dateTime

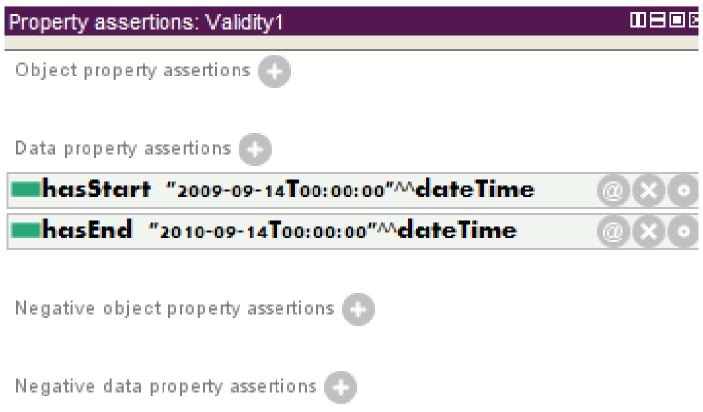
Negative object property assertions

Negative data property assertions

*Figure 4.2.11: Data Properties of TriggerAtTime1 Instance*

And finally “hasValidiy” object property domain is Obligation and range is Validity class. When we check Validity1 instance we see that it has two data properties called “*hasStart*” and “*hasEnd*”.

Validiy1: hasStart: “2009-09-14T00:00:00”^^dateTime
Validiy1: hasEnd: “2010-09-14T00:00:00”^^dateTime


Property assertions: Validity1

Object property assertions

Data property assertions

hasStart "2009-09-14T00:00:00"^^dateTime

hasEnd "2010-09-14T00:00:00"^^dateTime

Negative object property assertions

Negative data property assertions

*Figure 4.2.12: Data Properties of Validiy1 Instance*

## 4.3 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.

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

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

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

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

$$\text{Authorization} \equiv \text{DownStreamUsageAuthorization} \cup \text{UsagePuposeAuthorization}$$
$$\text{Authorization} \subseteq \text{DownStreamUsageAuthorization}$$
$$\text{Authorization} \subseteq \text{UsagePuposeAuthorization}$$
$$\text{DownStreamUsageAuthorization} \cap \text{UsagePuposeAuthorization} \subseteq \bot$$

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

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

TriggerSet class objects should have at least one Trigger.

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

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

$$\text{Trigger} \equiv \text{TriggerAtTime} \cup \text{TriggerPeriodic} \cup \text{TriggerPersonalDataAccessed}$$
$$\text{Trigger} \subseteq \text{TriggerAtTime}$$
$$\text{Trigger} \subseteq \text{TriggerPeriodic}$$
$$\text{Trigger} \subseteq \text{TriggerPersonalDataAccessed}$$
$$\text{TriggerAtTime} \cap \text{TriggerPeriodic} \cap \text{TriggerPersonalDataAccessed} \subseteq \bot$$

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.

$$\text{Action} \equiv \text{ActionAnonymizePersonalData} \cup \text{ActionDeletePersonalData}$$
$$\cup\ \text{ActionNotifyDataSubject}$$
$$\text{Action} \subseteq \text{ActionAnonymizePersonalData}$$
$$\text{Action} \subseteq \text{ActionDeletePersonalData}$$
$$\text{Action} \subseteq \text{ActionNotifyDataSubject}$$
$$\text{ActionNotifyDataSubject} \equiv \text{ActionNotifyDataSubjectByEmail}$$
$$\cup\ \text{ActionNotifyDataSubjectBySMS}$$
$$\text{ActionNotifyDataSubject} \subseteq \text{ActionNotifyDataSubjectByEmail}$$
$$\text{ActionNotifyDataSubject} \subseteq \text{ActionNotifyDataSubjectBySMS}$$

# Chapter 5

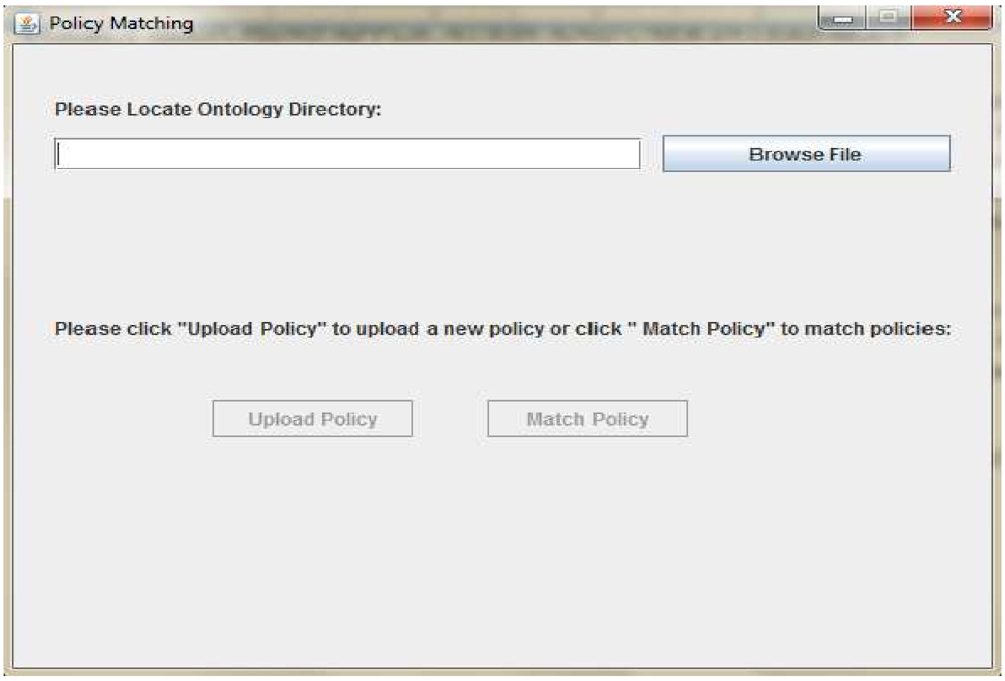
## POLICY MATCHING APPLICATION AND USER INTERFACES

“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.

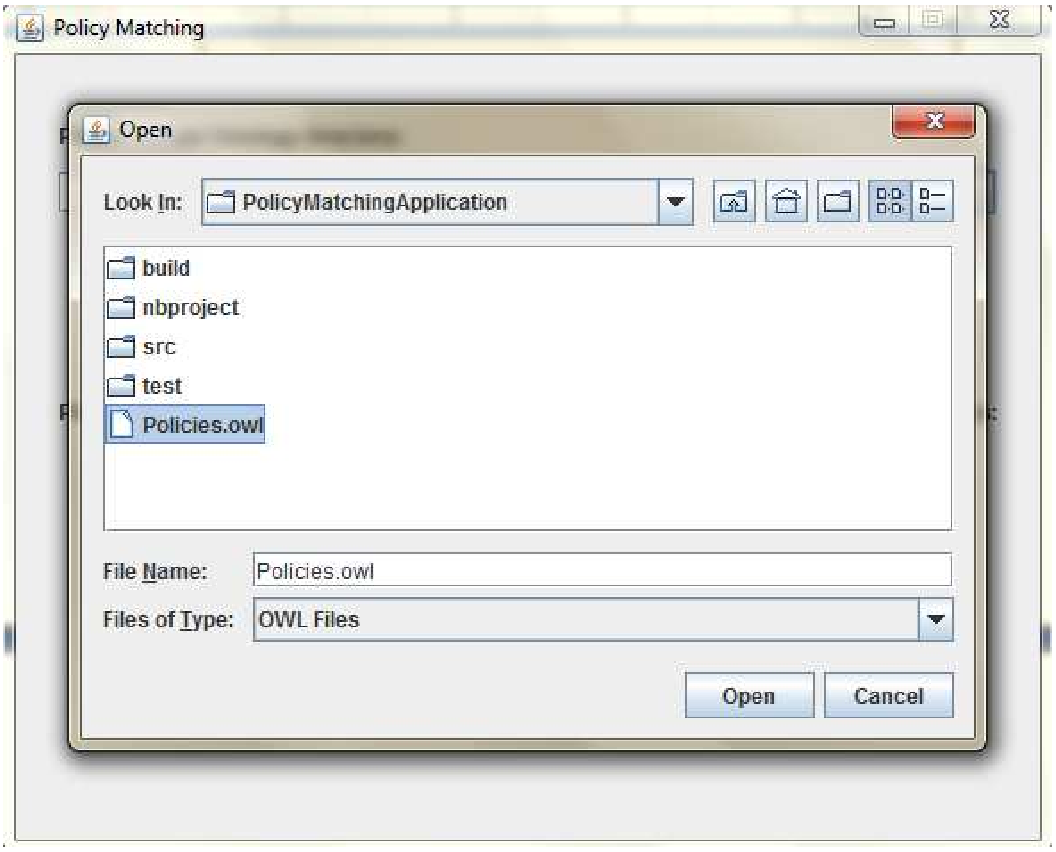
Whenever Policy Matching application starts to run, application will request location of the Policy Ontology from the user. After locating Policy Ontology, the application will allow the user either upload a new policy to the system or match two policies those already exist.


Policy Matching

Please Locate Ontology Directory:

Please click "Upload Policy" to upload a new policy or click " Match Policy" to match policies:

*Figure 5.1: Policy Matching Application*


Policy Matching

Open

Look In: PolicyMatchingApplication

build
nbproject
src
test
Policies.owl

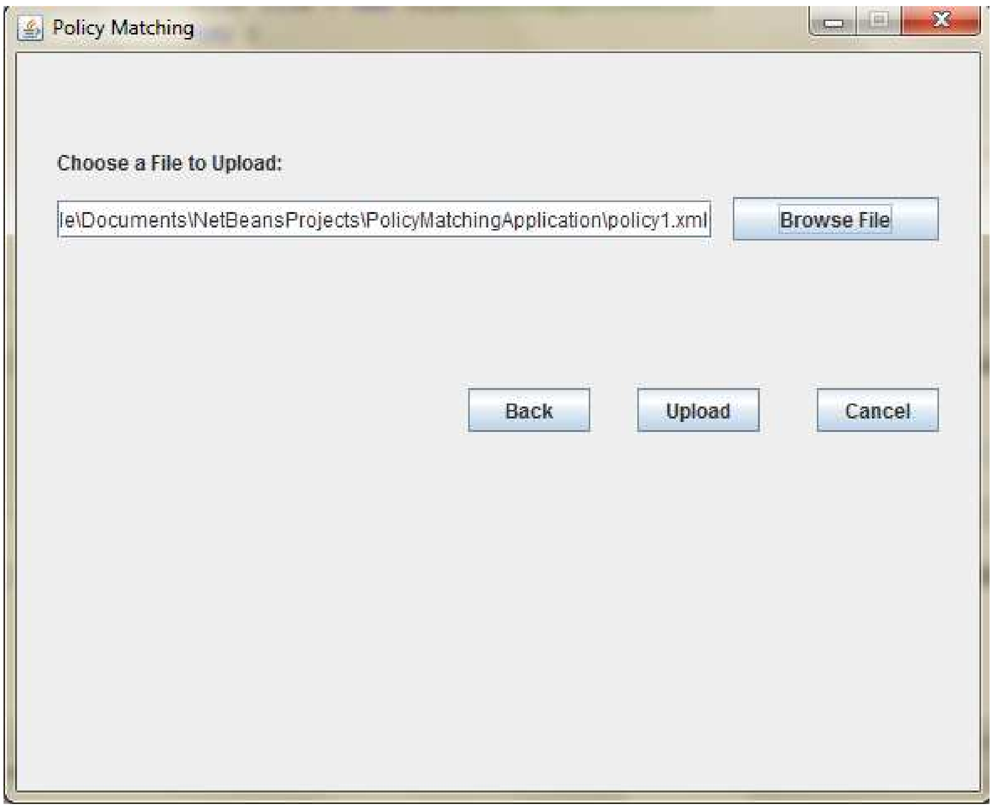
File Name: Policies.owl

Files of Type: OWL Files

*Figure 5.2: Locate Policy Ontology*

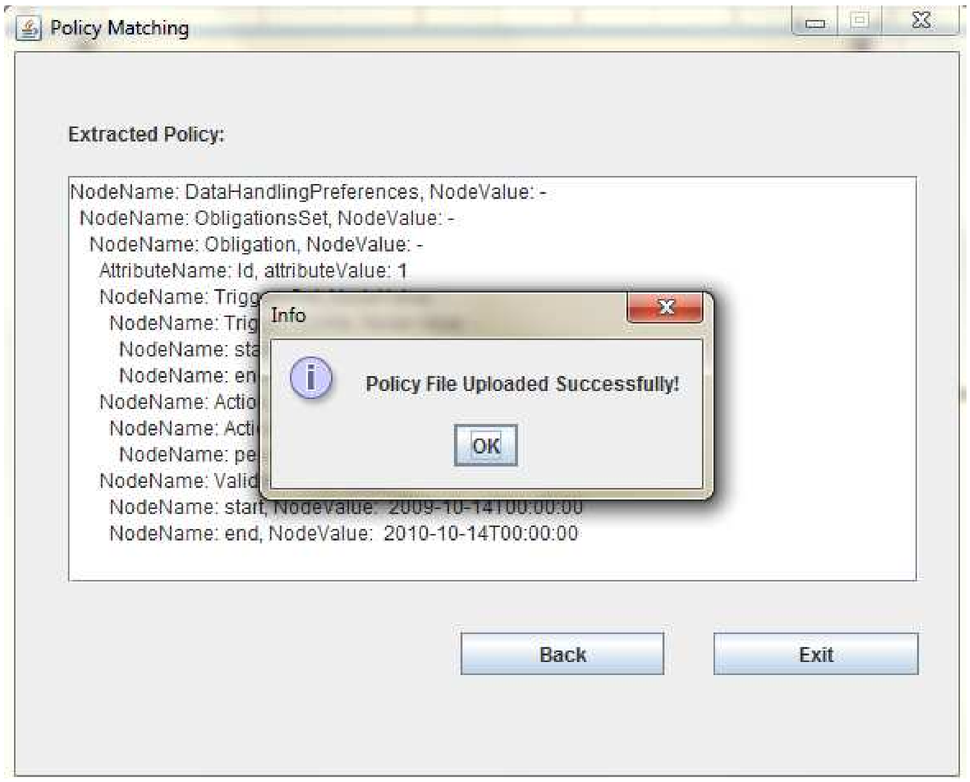
After locating the Policy Ontology, let us assume user selects uploading a new policy. To do so, he or she should specify the directory for the policy to be uploaded. Once policy document is specified the system will parse the document, extract related information and store it in Policy Ontology. If an error occurs user will be notified by the system otherwise extracted policy information will be presented to the user.

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.



le\Documents\NetBeansProjects\PolicyMatchingApplication\policy1.xml

*Figure 5.3: Locate Policy Document*


Policy Matching

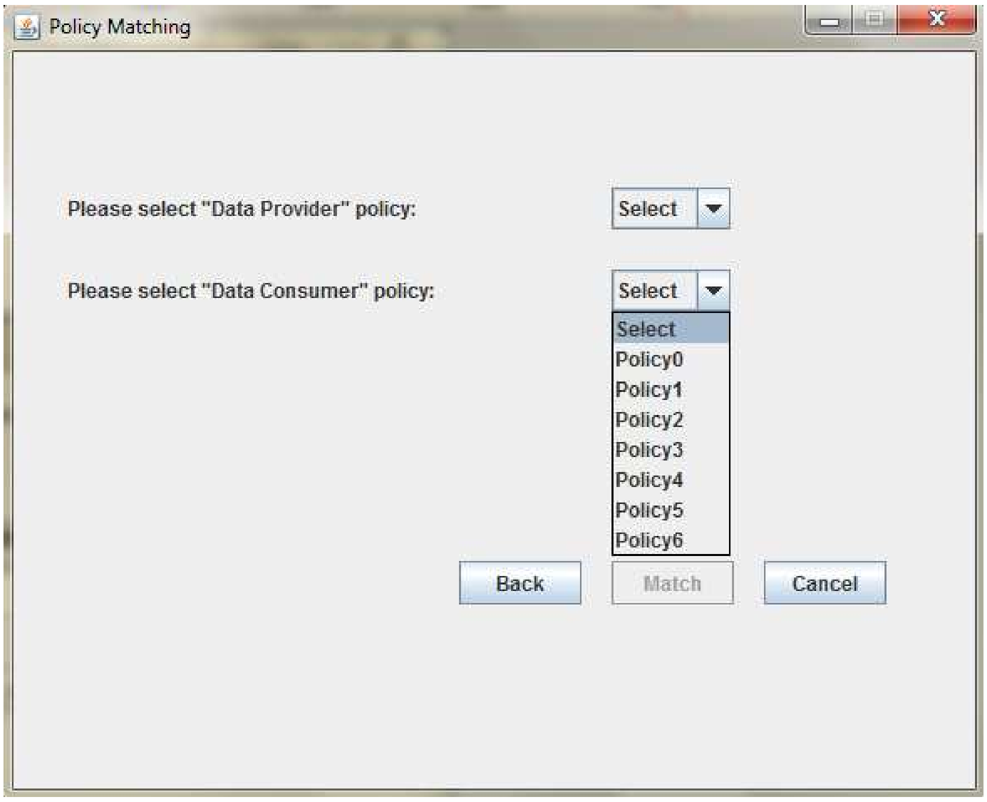
Extracted Policy:

```
NodeName: DataHandlingPreferences, NodeValue: -
 NodeName: ObligationsSet, NodeValue: -
  NodeName: Obligation, NodeValue: -
   AttributeName: Id, attributeValue: 1
   NodeName: Trigg
    NodeName: Trig
     NodeName: sta
     NodeName: en
   NodeName: Actio
    NodeName: Acti
     NodeName: pe
   NodeName: Valid
    NodeName: start, NodeValue: 2009-10-14T00:00:00
    NodeName: end, NodeValue: 2010-10-14T00:00:00
```

Info

Policy File Uploaded Successfully!

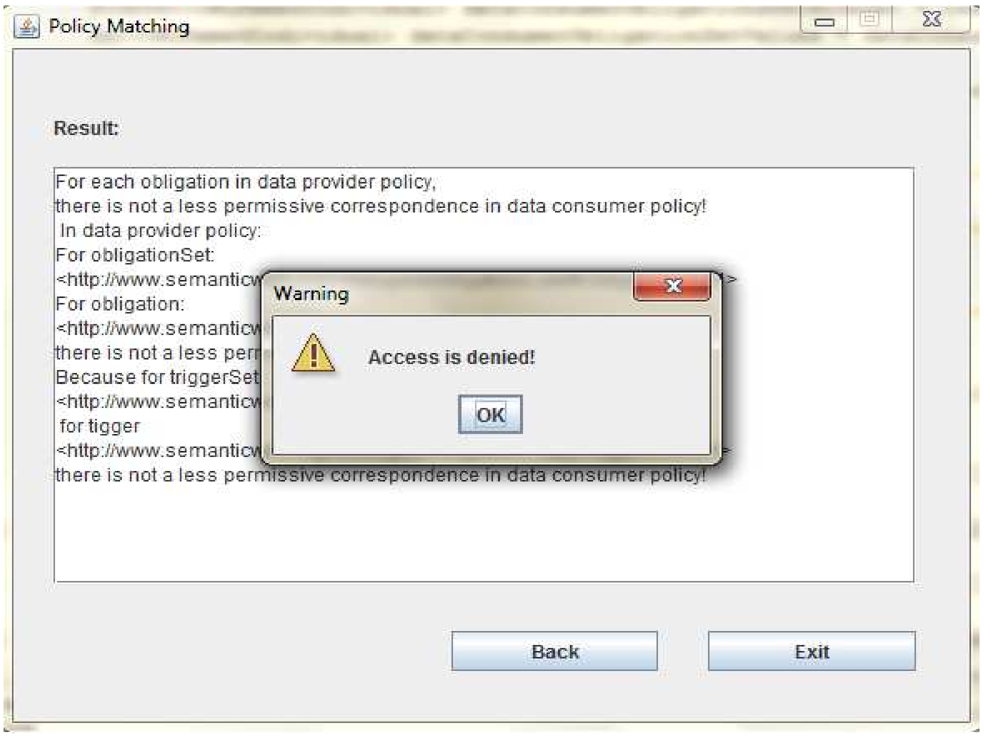
Figure 5.4: Present Extracted Policy Information


Policy Matching

Please select "Data Provider" policy: Select

Please select "Data Consumer" policy: Select

Figure 5.5: Select Policies to Match



```
For each obligation in data provider policy,
there is not a less permissive correspondence in data consumer policy!
 In data provider policy:
For obligationSet:
<http://www.semanticw
For obligation:
<http://www.semanticw
there is not a less perr
Because for triggerSet
<http://www.semanticw
 for tigger
<http://www.semanticw
there is not a less permissive correspondence in data consumer policy!
```

Access is denied!

*Figure 5.6: Matching Result*

## 5.1 Example

Let us consider two policy document examples shown in Figure 5.1.1 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 (Obligation1, TriggerSet1)
TriggerAtTime(TriggerAtTime1)
hasTrigger (TriggerSet1, TriggerAtTime1)
hasStart (TriggerAtTime1, 2010-01-01)
hasMaxDelay (TriggerAtTime1, P30D)
Action (Action1)
hasAction (Obligation1, Action1)
ActionDeletePersonalData (ActionDeletePersonalData1)
hasSubAction (Action1, ActionDeletePersonalData1)
hasData (ActionDeletePersonalData1, ref to personal data)
Validity (Validity1)
hasValidity (Obligation1, Validity1)
hasStart (Validity1, 2009-09-14)
hasEnd (Validity1, 2009-09-14)
TriggerSet (TriggerSet2)
hasTriggerSet (Obligation2, TriggerSet2)
TriggerReadData(TriggerReadData1)
hasTrigger (TriggerSet2, TriggerReadData1)
hasData (TriggerReadData2, ref to personal data)
Action (Action2)
hasAction (Obligation2, Action2)
ActionNotifyByEmail (ActionNotifyByEmail1)
hasSubAction (Action2, ActionNotifyByEmail1)
hasUsername (ActionNotifyByEmail1, Bob)
hasAddress(ActionNotifyByEmail, bob@contoso.com)
Validity (Validity2)
hasValidity (Obligation2, Validity2)
hasStart (Validity2, 2009-09-14)
hasEnd (Validity2, 2009-09-14)

Logical axioms for the second policy are likewise:

Policy (Policy2)
AuthorizationSet (AuthorizationSet2)
hasAuthorizationSet (Policy2, AuthorizationSet2)
DownStreamUsageAuthorization (DownStreamUsageAuthorization2)
hasAuthorization (AuthorizationSet2,DownStreamUsageAuthorization2)
hasPermission(DownStreamUsageAuthorization2,true)
ObligationSet (ObligationSet2)
hasObligationSet (Policy2, ObligationSet2)
Obligation (Obligation3)

Obligation (Obligation4)
hasObligation (ObligationSet2, Obligation3)
hasObligation (ObligationSet2, Obligation4)
TriggerSet (TriggerSet3)
hasTriggerSet (Obligation3, TriggerSet3)
TriggerAtTime(TriggerAtTime2)
hasTrigger (TriggerSet3, TriggerAtTime2)
hasStart (TriggerAtTime2, 2009-10-14)
hasMaxDelay (TriggerAtTime2, P365D)
Action (Action3)
hasAction (Obligation, Action3)
ActionDeletePersonalData (ActionDeletePersonalData2)
hasSubAction (Action3, ActionDeletePersonalData2)
hasData (ActionDeletePersonalData2, ref to personal data)
Validity (Validity3)
hasValidity (Obligation3, Validity3)
hasStart (Validity3, 2009-09-14)
hasEnd (Validity3, 2010-09-14)
TriggerSet (TriggerSet4)
hasTriggerSet (Obligation4, TriggerSet4)
TriggerReadDataForPurpose(TriggerReadDataForPurpose1)
hasTrigger (TriggerSet4, TriggerReadDataForPurpose1)
hasData (TriggerReadDataForPurpose1, ref to personal data)
hasPurpose (statistics)
Action (Action4)
hasAction (Obligation4, Action4)
ActionNotifyUser (ActionNotifyUser1)
hasSubAction (Action4, ActionNotifyUser1)
hasUsername (ActionNotifyUser1, Bob)
Validity (Validity4)
hasValidity (Obligation4, Validity4)
hasStart (Validity4, 2009-09-14)
hasEnd (Validity4, 2010-09-14)

```
DataHandlingPolicy {
authorizations: ListAuthorizations {
   authorization:
       DownStreamUsageAuthorization {
            allow: “false” }
},
obligations: ListObligations {
     n0: Obligation {
          triggers: ListTriggers {
                 t1: TriggerAtTime {
                      start: 01/01/2010,
                      maxDelay: 30 days } },
         action:ActionDeletePersonalData{
                      personalData:    "ref    to
personal data"},
          validity: Validity {
                      start: 9/14/2009,
                      end: 9/14/2009 } },
     n1: Obligation {
         triggers: ListTriggers {
               t1: TriggerReadData {
                      dataRef:  "ref  to  personal
data" }},
             action: ActionNotifyByEmail {
                      userName : "Bob",
                      address:
"bob@contoso.com" },
               validity: Validity {
                      start: 9/14/2009,
                       end: 9/14/2009 } } } }
```

```
DataHandlingPreference {
authorizations: ListAuthorizations {
   authorization:
       DownStreamUsageAuthorization {
            allow: “true” }
},
obligations: ListObligations {
      n0: Obligation {
          triggers: ListTriggers {
                 t1: TriggerAtTime {
                        start: 10/14/2009,
                        maxDelay : 365 days } },
         action:ActionDeletePersonalData{
                      personalData:    "ref    to
personal data"},
            validity: Validity {
                          start: 9/14/2009,
                          end: 9/14/2010} },
         n1: Obligation {
              triggers: ListTriggers {
                 t1: TriggerReadForPurpose {
                        personalData:   "ref   to
personal data"
                        purpose: "statistics"} },
               action:  ActionNotifyUser {
                           userName: "Bob" },
                  validity: Validity {
                            start: 9/14/2009,
                            end: 9/14/2010 }}
}}
```

*Figure 5.1.1: Policy Examples*

After representing the data provider and consumers’ policies in OWL, system will try to find an equally or more permissive authorizationSet in the data provider’s policy for each authorizationSet in the data consumer’s policy.  In our example, data consumer policy’s authorizationSet involves just one authorization: DownStreamUsageAuthorization. For this kind of authorizations, the system concludes data consumer’s policy is equally or less permissive than data provider’s policy if and only if one of the following conditions exists:

- If allowed is false in the data controller's policy: data controller specifies that he/she will not forward data to the third parties.
- If allowed is true in both data controller and data providers' policy: both agree on that data consumer may forward data to the third parties.

In our example data consumer policy's DownStreamUsageAuthorization's allow element's value is false. Thus system will decide AuthorizationSet1 is equally or less permissive and will continue with ObligationSet.

After examining authorizationSet, system will try to find for each obligationSet in the data provider's policy, an equally or less permissive obligationSet in the data consumer's policy. Let us start to analyze first and only obligationSet, instance ObligationSet2, in the data provider policy.

ObligationSet2 involves two obligations, Obligation3 and Obligation4. For each obligation in the data provider policy, system has to find an equally or less permissive obligation in the data consumer policy. Each obligation composed of three parts: TriggerSet, Action and Validity.

```
///data consumer
 n0 : Obligation {
        triggers : ListTriggers {
            t1 : TriggerAtTime {
                  start : 01/01/2010,
                   maxDelay : 30 days } },
………………….
```

```
 /// data provider
   n0 : Obligation {
         triggers : ListTriggers {
              t1 : TriggerAtTime {
                     start : 10/14/2009,
                     maxDelay : 365 days } },
…………………………
```

Take the triggerSet, TriggerSet3, of first obligation, Obligation3, in data provider side and consider first trigger, TriggerAtTime2. We can say a trigger is equally or less permissive if and only if trigger in data consumer's policy has a small time window:

Policy.TriggerAtTime.startdate $\geq$ Preference.TriggerAtTime.startdate and
Policy.TriggerAtTime.startdate + Policy. TriggerAtTime.maxDelay $\leq$
Preference.TriggerAtTime.startdate + Preference. TriggerAtTime.maxDelay

Therefore system will send a query to reasoner to check "is there a TriggerAtTime whose *startDate* is bigger than "10/14/2009"

TriggerAtTime and (hasStart some dateTime [>= "2009-10-14T00:00:00"^^dateTime])

Answer is instance TriggerAtTime1. To calculate second part consider following modification: whenever we have a trigger which is TriggerAtTime type we can calculate an endDate by adding maxDelay to startDate. Therefore instance TriggerAtTime2 will have endDate:

10/14/2010 and instance TriggerAtTime1 will have 01/02/2010. Now new query will be: “is there a TriggerAtTime whose startDate is bigger than “10/14/2009”and endDate is smaller than 10/14/2010. Answer is instance TriggerAtTime1.

> TriggerAtTime and (hasStart some dateTime [>= "2009-10-14T00:00:00"^^dateTime]) and (hasEnd some dateTime [<= "2010-10-14T00:00:00"^^dateTime])

In our simple example TriggerSet3 contains just one trigger but in principle a triggerSet may contain several triggers. Thus abovementioned process should be done for each trigger in the triggerSet.

Before matching actions consider following query which will return an obligation instance that TriggerAtTime1 belongs, Obligation1:

> Obligation and (hasTriggerSet some (TriggerSet and (hasTrigger some (TriggerAtTime and (hasStart some dateTime [= "2010-01-01T00:00:00"^^dateTime]) and (hasEnd some dateTime [= "2010-02-01T00:00:00"^^dateTime])))))

Answer is Obligation1. Then system will go on by comparing actions:

```
///data consumer
………..
  action :ActionDeletePersonalData {
                personalData:    "ref    to
personal data"},
……
```

```
///data provider
………..
  action :ActionDeletePersonalData {
                personalData:    "ref    to
personal data"},
……
```

System will query to find an obligation which has an action “ActionDeletePersonalData” and for this action data property hasData’s value is “ref to personal data”

> Obligation and (hasAction some (ActionDeletePersonalData and hasData value "ref to personal data"))

Result is instance Obligation1, same with result of triggerSet so system will go on with validity.

```
///data consumer
…….
validity: Validity {
                start : 9/14/2009,
                 end : 9/14/2009 } },
```

```
///data provider
…….
            validity: Validity {
                    start : 9/14/2009,
                    end : 9/14/2010} },
```

A validity is equally or less permissive than the other one if and only if first one's start date is bigger than the second one's while end date of first one is smaller than the second one. The following query will search for an obligation whose validity time frame is inside time frame of validity that specified in data provider side.

> Obligation and (hasValidity some (Validity and (hasStart some dateTime [>= "2009-09-14T00:00:00"^^dateTime]) and (hasEnd some dateTime [<= "2010-09-14T00:00:00" ^^dateTime] )))

Instances Obligation1 and Obligation2 will be answer for the query. Since Obligation1 was result of triggerSet and action, system concludes for Obligation3 there is a less permissive obligation, Obligation1 in data consumer policy.

Now system will consider next obligation, Obligation4, in ObligationSet2. Again it will start from triggerSet. Obligation4 has TriggerSet4 which involves TriggerReadForPurpose1 instance.

```
///data consumer
 n1 : Obligation {
       triggers : ListTriggers {
           t1 : TriggerReadData {
                dataRef : "ref to personal
data" }
          },
…………..
```

```
  ///data provider
     n1 : Obligation {
          triggers : ListTriggers {
              t1 : TriggerReadForPurpose {
                 personalData : "ref to
personal data"
                 purpose : "statistics" } },
………
```

To compare hierarchical triggers system uses following rules: (Assume C is data consumer policy while P is data provider preferences)

- If data provider and consumer both have "TriggerReadData" then compare data.
  C:TriggerReadData ⊴ P:TriggerReadData ⇔ (C.dataRef ==P.dataRef )

- If data provider and consumer both have "TriggerReadForPurpose" then compare data and purposes.
  C: TriggerReadForPurpose ⊴ P: TriggerReadForPurpose ⇔
  ((C.dataRef == P.dataRef ) ∧ (C.purpose == P.purpose ) )

- If data consumer has "TriggerReadData" and data provider has "TriggerReadForPurpose" then compare only data.
  C: TriggerReadData ⊴ P: TriggerReadForPurpose ⇔ ( C.dataRef == P.dataRef )

In our example we have TriggerReadData for data consumer policy while TriggerReadForPurpose for data provider preference. Hence query will be:

TriggerPersonalDataAccessed and (hasData value "ref to personal data")

Result of query will be instance TriggerPersonalDataAccessed1 which belongs to Obligation2:

Obligation and hasTriggerSet some (TriggerSet and (hasTrigger some (TriggerPersonalDataAccessed and (hasData value "ref to personal data" ))))

Now system will reason about actions.

```
///data consumer
……
        action : ActionNotifyByEmail {
            userName : "Bob",
            address:   "bob@contoso.com"
},
```

```
 ///data provider
……..
                action: ActionNotifyUser {
                        username : "Bob" },

…….
```

To compare hierarchical actions system uses following rules: (Assume C is data consumer policy while P is data provider preferences)

- If data provider and consumer both have "ActionNotifyUser" then we will compare usernames.
  C:ActionNotifyUser ⊴ P:ActionNotifyUser ⇔ (C.userName ==P.userName)

- If data provider and consumer both have "ActionNotifyByEmail" then we will compare usernames and addresses.

  C:ActionNotifyByEmail ⊴ 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⊴R:ActionNotifyUser ⇔ (C.userName == P.userName)

In our example data consumer's policy has ActionNotifyByEmail and data provider's preference has ActionNotifyUser. Thus query will be:

ActionNotifyByEmail and (hasUsername value "Bob")

Result is instance ActionNotifyByEmail1 which belongs to Obligation2:

Obligation and (hasAction some (ActionNotifyByEmail1 and (hasUsername value "Bob")))

Same obligation instance with triggerSet therefore system will continue with validity.

```
///data consumer
…………..
          validity : Validity {
                 start : 9/14/2009,
                  end : 9/14/2009 } }
 } }
```

```
 ///data provider
…………..
               validity : Validity {
                        start : 9/14/2009,
                        end : 9/14/2010 } }
} }
```

System is searching for an obligation which has validity such that its start date is bigger than defined in the data provider policy and end date is smaller than defined in the data provider policy. The following query will search for an obligation whose validity time frame is inside time frame of validity that specified in data provider side.

Obligation and (hasValidity some (Validity and (hasStart some dateTime [>= "2009-09-14T00:00:00"^^dateTime]) and (hasEnd some dateTime [<= "2010-09-14T00:00:00" ^^dateTime] )))

Instances Obligation1 and Obligation2 will be answer of the query. Since Obligation2 is in the result set, system concludes that for Obligation4 there exists a less permissive obligation, Obligation2 in data consumer policy.

Finally system has finished comparing all obligations in ObligationSet2 in data provider policy. Now system checks if Obligation1 and Obligation2 are in the same obligationSet or not:

ObligationSet and (hasObligation value Obligation1) and (hasObligation value Obligation2)

Result is ObligationSet1. Since both obligations are in the same obligationSet system will conclude that for ObligationSet2 in data provider policy there exists a less permissive obligationSet in data consumer policy. Hence data consumer's policy is less permissive than data provider's policy and access may be permitted.

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.

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 the existing technologies can be a little difficult, because of poor documentation and incompatibility issues.  For example in our work, we have faced difficulties to retrieve relevant data types from ontology and use it properly within java code. Similarly, we have faced to send proper data types to the ontology from a java code.

# Chapter 6

## 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. In this chapter we will provide some examples to indicate Policy Matching application usage.

### 6.1 Social Network Example

Let us look a famous social network, Facebook [115]. Registered users can define privacy settings for their personal information, contact information and other related information such that photos, videos etc. about their profiles. Assume a data provider defines a policy in which he/she states that: Personal information can be seen by everyone while contact and other related information can be seen only by his/her family and friends. These restrictions easily can be granted by defining proper authorizations.

Moreover the data provider may put more restrictions such that only the family members may tag his/her photos and videos. Data subject also wants to be notified whenever his/her data are accessed. Let us now formalize these restrictions with the extended XACML policy language.

```
<Policy>
        <AuthorizationSet>
                <Authorization>
                        <RoleAuthorization>
                                <role> Family </role>
                        </RoleAuthorization>
                </Authorization>
        </AuthorizationSet>
        <ObligationsSet>
                <Obligation >
                        <TriggersSet>
                                <TriggerPersonalDataAccessed>
                                        <data> Personal Information </data>
                                        <data> Contact Information </data>
                                </ TriggerPersonalDataAccessed >
                                <TriggerPersonalDataAccessedForPurpose>
                                        <data> Other Information </data>
                                        <purpose> Tag </purpose>
                                </ TriggerPersonalDataAccessedForPurpose >
                        </TriggersSet>
                        <Action>
                                < ActionNotifyUser >
                                        <username> Emine </ username >
                                </ActionNotifyUser >
                        </Action>
                        <Validity>
                                <start> 2009-10-14T00:00:00 </start>
                                <end> 2010-10-14T00:00:00 </end>
                        </Validity>
                </Obligation>
        </ObligationsSet>
</Policy>
```

*Figure 6.1.1: Policy Example for Family – Data Provider*

Expressing policies by using OWL has one more important advantage: It is very flexible thus new features can be added easily. In our example we have to add a new authorization subclass “RoleAuthorization” to Policy Ontology and this subclass should have a data property called hasRole which specifies authorization role.

OWL axioms for abovementioned example are the followings:

```
Policy (Policy1)
AuthorizationSet (AuthorizationSet1)
hasAuthorizationSet (Policy1, AuthorizationSet1)
RoleAuthorization (RoleAuthorization1)
hasAuthorization (AuthorizationSet1, RoleAuthorization1)
ObligationSet (ObligationSet1)
hasObligationSet (Policy1, ObligationSet1)
Obligation (Obligation1)
hasObligation (ObligationSet1, Obligation1)
TriggerSet (TriggerSet1)
hasTriggerSet (Obligation1, TriggerSet1)
TriggerPersonalDataAccessed (TriggerPersonalDataAccessed1)
hasTrigger (TriggerSet1, TriggerPersonalDataAccessed1)
hasData (TriggerPersonalDataAccessed1, Personal Information )
hasData (TriggerPersonalDataAccessed1, Contact Information )
hasData (TriggerPersonalDataAccessed1, Other Information )
TriggerPersonalDataAccessedForPurpose (TriggerPersonalDataAccessedForPurpose1)
hasTrigger (TriggerSet1, TriggerPersonalDataAccessedForPurpose1)
hasData (TriggerPersonalDataAccessedForPurpose1, Other Information )
hasPurpose (TriggerPersonalDataAccessedForPurpose1,Tag)
Action (Action1)
hasAction (Obligation1, Action1)
ActionNotifyUser (ActionNotifyUser1)
hasSubAction (Action1, ActionNotifyUser1)
hasUsername (ActionNotifyUser1, Emine)
Validity (Validity1)
hasValidity (Obligation1, Validity1)
hasStart (Validity1, 2009-10-14)
hasEnd (Validity1, 2010-10-14)
```

Assume that the data subject specifies similar policies for the friends and others: Friends may access personal, contact and other information but without tagging right. Others may access only the personal information. Now let us consider a data consumer whose relation with data provider is friendship. Data consumer wishes to access the data provider’s other information. Hence he/she define a policy:

```
<Policy>
        <AuthorizationSet>
                <Authorization>
                        <RoleAuthorization>
                                <role> Friend </role>
                        </RoleAuthorization>
                </Authorization>
        </AuthorizationSet>
        <ObligationsSet>
                <Obligation >
                        <TriggersSet>
                                <TriggerPersonalDataAccessedForPurpose>
                                        <data> Other Information </data>
                                        <purpose> Tag </purpose>
                                </ TriggerPersonalDataAccessedForPurpose >
                        </TriggersSet>
                        <Action>
                                < ActionNotifyUser >
                                        <username> Emine </ username >
                                </ActionNotifyUser >
                        </Action>
                        <Validity>
                                <start> 2009-10-14T00:00:00 </start>
                                <end> 2010-10-14T00:00:00 </end>
                        </Validity>
                </Obligation>
        </ObligationsSet>
</Policy>
```

*Figure 6.1.2: Policy Example for Friend-Data Consumer*

To compare hierarchical triggers as we mentioned in previous chapters, the system uses following rules: (Assume C is data consumer policy while P is data provider preferences)

- If data provider and consumer both have "TriggerPersonalDataAccessed" then compare data.

  C: TriggerPersonalDataAccessed $\trianglelefteq$ P: TriggerPersonalDataAccessed $\Leftrightarrow$ (C.dataRef ==P.dataRef )

- If data provider and consumer both have "TriggerPersonalDataAccessedForPurpose" then compare data and purposes.

  C: TriggerPersonalDataAccessedForPurpose ⊴
  P: TriggerPersonalDataAccessedForPurpose ⇔
  ((C.dataRef == P.dataRef ) ∧ (C.purpose == P.purpose ) )

- If data consumer has "TriggerPersonalDataAccessed" and data provider has "TriggerPersonalDataAccessedForPurpose" then compare data.

  C: TriggerPersonalDataAccessed ⊴ P: TriggerPersonalDataAccessedForPurpose ⇔
  ( C.dataRef == P.dataRef )

However in our example data provider has "TriggerPersonalDataAccessed" while data consumer has "TriggerPersonalDataAccessedForPurpse", thus access will be denied. It is also possible to compare data consumer policy with data provider's family policy but in this case data provider's family authorization mismatches with data consumer's friend authorization and access will be denied again.

Data consumer may access data if and only if he/she defines an equally or less permissive policy in which he/she states other information will be accessed without tagging purpose.

As we see from this basic example, Policy Matching application is fit very well to the nature of social network access control issues. Users' privacy policies can be represented by Policy Ontology easily and OWL reasoner can be used to find mismatches if they exist.

## 6.2 Commercial Application Example

Let us have a look website of CardScan [12] 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 following example: Suppose a data provider use 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 actual data consumer is the CardScan system and according to the policy between the data provider and the system, system may forward data to the third parties or not. Figure 6.2.1 illustrates this situation.

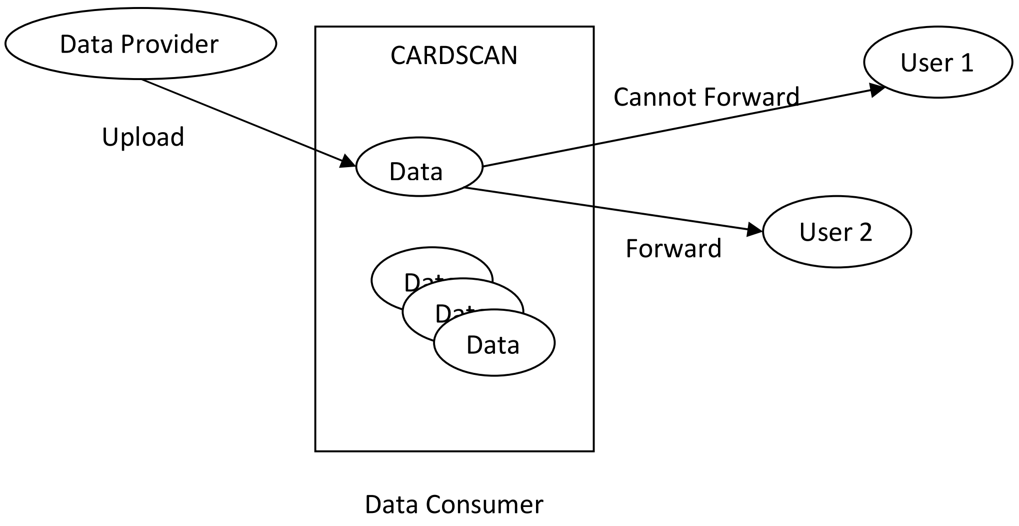


*Figure 6.2.1: CardScan Structure*

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 6.2.2.

```
<Policy>
        <AuthorizationSet>
                <Authorization>
                        < DownStreamUsageAuthorization >
                                <allow>  false  </allow>
                                <subject> user1 </subject>
                        </ DownStreamUsageAuthorization >
                        < DownStreamUsageAuthorization >
                                <allow> true </allow>
                                <subject> user2 </subject>
                        </ DownStreamUsageAuthorization >
                </Authorization>
        </AuthorizationSet>
        <ObligationsSet/>
</Policy>
```

*Figure 6.2.2: Data Provider Policy*

From these two application examples we can observe that 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.

Specifying and using access control with Semantic Web services are getting popular day by day. Functionalities of OWL reasoner give us great advantages on analyzing effect and consequences of set of access decisions. Flexibility of adding/removing classes and relations in OWL allow us to improve defined ontology structure.

# Chapter 7

## CONCLUSION

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. As we have shown in chapter 6 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.

Specifying and using access control with Semantic Web services are getting popular day by day. Functionalities of OWL reasoner give us great advantages on analyzing effect and consequences of set of access decisions. Flexibility of adding/removing classes and relations in OWL allow us to improve defined ontology structure.

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 continue to investigate the Semantic Web services in relation with access control models and declarative policy languages.

## BIBLIOGRAPHY

[1] Access Control Models: http://en.wikipedia.org/wiki/Access_control

[2] Primeife Project: http://www.primelife.eu/images/stories/h5.3.2-seconddesign/h5.3.2.html

[3] Pierangela Samarati and Sabrina De Capitani di Vimercati, Access Control: Policies, Models, and Mechanisms, Lecture Notes in Computer Science; Vol. 2171, pages 137-196, 2000

[4] XACML: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[5] OWL: www.w3.org/TR/owl-features/

[6] http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html

[7] Pizza Ontology: http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/

[8] Semantic Web: http://semanticweb.org/wiki/Main_Page

[9] RDF: www.w3.org/RDF/

[10] Hermit: http://www.hermit-reasoner.com/

[11] Facebook: www.facebook.com

[12] CardScan: www.cardscan.net

[13] Feng Huang, Zhiqiu Huang and Linyuan Liu, A DL-based Method for Access Control Policy Conflict Detecting, Asia-Pacific Symposium on Internetware, Beijing, China, 2009

[14] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough and B. Thuraisingham, ROWLBAC: Representing Role Based Access Control in OWL, SACMAT: Symposium on Access Control Models and Technologies, Colorado, USA, 2008

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

[16] Logic-Based Access Control Policy Specification and Management, Vladimir Kolovski: www.cs.umd.edu/Grad/scholarlypapers/papers/VKolovski.pdf