Representing Attribute Based Access Control Policies in OWL

Nitin Kumar Sharma
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi, India-110016
Email: mcs142867@cse.iitd.ac.in

Anupam Joshi
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County, Baltimore, MD
Email: joshi@umbc.edu

***Abstract*—Attribute Based Access Control (ABAC) models are designed with the intention to overcome the shortcomings of classical access control models (DAC, MAC and RBAC) and unifying their advantages. In ABAC, the access control is provided based on generic attributes of entities (users, subjects, and objects). Many organizational security policies condition access decisions on attributes. OWL can be used to formally define and process security policies that can be captured using ABAC models. We have defined models, domains, data and security policies in OWL and used a reasoner to decide what is permitted. In this paper we present a way to represent the ABAC-α model using Web Ontology Language (OWL). The enforcement of policies is done based on the inference process. We have used EYE reasoner that infers the logical relationship and deduce the access grant for each requested action.**

I. Introduction

Most organizations have policies, not just internal but also external (statutatory, compliance, legal, ...) that control their behavior. These policies control access rights in their organization. The ability to capture these policies, which are normally expressed in some natural language, in a machine understandable format has been an active thread of research (see for instance [3], [8], [10], [11]). Policy specification is a challenging task. The Web Ontology Language (OWL) [9] provides an efficient way to represent policies formally. Although the language was primarily designed to represent the knowledge content of web information, it has been used to represent security policies [11]. OWL provides an efficient and standard way to write complex ontologies. Real world entities and their relationships are quite complex in nature and OWL helps to capture them easily. Access control models when combined with formal policy specification language like OWL give the ability to write policies that describe entities and relationships in the system, how they affect access control, and how they are grounded out in models that are well understood in the security community. This combination further helps by using the power of reasoning to make access control decisions. A key contribution of our paper is to create an ontology and rules that capture the Attribute Based Access Control model, thus allowing for policies that are grounded out in ABAC. The specific model we capture is ABAC-α [4]. We have used the EYE [1] reasoner to infer more facts from the specified access control model, data and policies.

Access control deals with restricting information flow across the system. The process is based on an underlying model which governs the control on information. There are three successful access control models which are commonly used in practice: Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC).

DAC works based on user identity and object's access control list. Owner of the information decides which type of access (*e.g* read, write or execute) is permitted to different users. In this model it is possible that information may be accessed by unauthorised user. This is possible due to the existence of multiple copies of the information and having no control on them.

MAC avoids the problem by using the concept of security level. The security levels are termed as *clearance* for the subjects and *classification* for the objects. The security levels are application on each copy of the object. Access is granted only if some relationship exists between the security levels of the subject and object. This relationship is captured by some security models like Bell LaPadula [14].

RBAC is based on user roles and associated permissions with each role. Access is granted based on permitted actions for each role. The model defines four different types to cover a wider spectrum. RBAC also suffers from problems like *role explosion* which corresponds to the situation in which different roles may require different permissions. This results in large number of roles and it becomes difficult to manage them.

ABAC model proposed by Jin *et. al* [4] is a general framework which combines the benefits of DAC, MAC and RBAC and goes beyond their limitation. The model is based on *attributes* which are associated with users, subjects and objects. These *attributes* are used to capture identities and access control lists for DAC, security labels, clearances and classifications for MAC and roles for RBAC.

The ABAC model provides access control based on the value and relationship among *attributes*. This allows for more flexibility for policy specifications as any number of *attributes* can be added within the same extensible framework. This also solves the shortcomings of the core RBAC model as appropriate attributes such as `location, business hours` *etc.* can be added within a unified framework. After specifying the authorization policy, dynamic computation of the authorization can be done at the time when the access request is made. As an example, the need to preassign the *roles* to users in RBAC can

be avoided. Based on the authorization policy, the relationship is determined at run time and appropriate access decision is taken. These features makes ABAC suitable for automation in the access control process.

There are many serialisations for OWL ontologies. We have selected `notation 3 (N3)` [7] as it has more human readability. It also provides support for writing rules and formulae. These rules and formulae are based on quantified (existential and universal) first order logic.

In next section, we present the related work which has been done in this area. Section III describes the ABAC_α model and and its core component. Section V describes the representation on the model in OWL and shows how all classical models can be implemented. Finally we conclude with brief comment on the scope of the future work.

II. Related Work

There have been considerable efforts to model access control policies in last few years. The effort was motivated by providing more flexibility in policy specifications and to enable automated inference of access control decisions. XACML is a general-purpose authorization policy model and XML based specification language [10]. XACML enforces access control based on *attributes*. This make XACML more beneficial than RBAC, however it does not consider the semantics of the policies which poses a limitations on its use. In an other approach [5], XACML has been combined with OWL. The basic idea is to decouple the management of constraints, such as SoD, and RBAC hierarchies from the specification and the enforcement of actual XACML policies.

There are other policy languages like Rei [11] which is based on deontic concepts. It associates access privileges with different credentials and properties of entities (user, agent, service, etc.). The language caters for the need of pervasive computing environment and provides flexible constructs like *policy objects*, *meta policies* and *speech acts* to express different types of policies. The policy objects represent deontic concepts of rights, obligations, prohibitions and dispensations. The meta policy specifications are for conflict resolution which include constructs for specifying precedence of modality and priority of policies. The speech acts (delegate, revoke, cancel and request) can be used to modify policies dynamically within the system. The policy language is not tied to any specific application and permits domain specific information to be added without modification.

The Rein [8] framework builds on REI and is based on N3 rules. It proposes representation and reasoning over distributed policies in the Semantic Web. It also allows policies to use information and inferences about entities and relationships defined in other policies and resources. Rein uses the cwm reasonng engine [12] to provide distributed reasoning capability over policy networks.

In another work a multi-layer policy framework, KAoS [3] is proposed which supports policies described in OWL. It interacts with the user to take policy specifications in the form of constrained English sentences. After that OWL is used to encode and manage policy-related information. Policy monitoring and enforcement is done automatically based on OWL ontologies. The framework provides KAoS guards which are used for policy information querying and decision-making.

While the policy languages described above were grounded in formal logics (e.g. deontic logic for REI), they were separate from the formalisms used in the security community. ROWLBAC [6] was an effort to bridge this gap by modeling RBAC in OWL. In this representation entities are represented as OWL classes. These entities are Users, Roles, Actions, Permissions etc. Using OWL hierarchies and properties, different ontologies have been suggested. In this representation two different approaches have been introduced to model roles: *role as classes* and *role as values*. The proposed work suggests to go beyond RBAC and discusses about ABAC briefly.

The policy enforcement process infers additional facts about the data to determine information flow across the system. This necessiates a resoning engine to be used. Euler Yet another proof Engine (EYE) [1] is an examples of a Prolog based reasoner. EYE is a theorem prover where users set a goal and it tries to reach it by applying logical rules, mostly working backward from the goal. The reasoner runs in Prolog virtual machine and is compatible with Prolog engines. This machine provides a generic reasoning environment by accepting its input in notation 3 (N3) rules.

In this paper we build on ROWLBAC and use EYE reasoner to infer additonal knowledge from the data.

III. ABAC_α Model

The ABAC_α model was developed as a first step with the goal to develop a large and more general family of ABAC models. The core component of the unified ABAC_α model are: users (U), subjects (S), objects (O), user attributes (UA), subject attributes (SA), object attributes (OA), permissions (P), authorization policies, and constraint checking policies for creating and modifying subject and object attributes.

The idea is to enforce access control based on generic *attributes*. *Attributes* are functions which map an entity to a specific value from its range. For example, *Role* is an *attribute* function and it may map user `Alice` to the role `Dean` in an academic institute. While user *attributes* are created by security administrator, the *attributes* for subjects are created by users at the time of their creation. The same is true for object *attributes*.

Security architects also configure constraints which are functions that are true when conditions are satisfied and false otherwise. Constraints can apply at subject and object creation time, and subsequently at subject and object attribute modification time.

Permissions are privileges that a user can hold on objects and exercise via a subject. Permissions definition is dependent on specific systems built using this model.

Authorization policies are also defined as two-valued boolean functions which are evaluated for each access decision. An authorization policy for a specific permission takes a

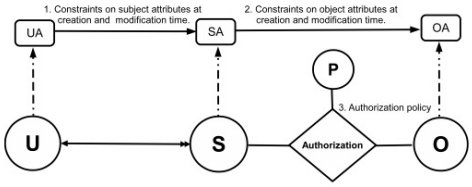

Fig. 1. Unified ABAC Model Structure as Proposed by Xin *et. al* [4]

subject, an object and returns true or false based on attribute values.

The overall structure of the ABAC_α model is shown in figure 1. The model describes four policy configuration points which can be specified using a *Common Policy Language* (CPL). The first configuration point is for authorization policies. One authorization policy is defined for each permission. The second configuration point is constraints for subject attribute assignment. The third is constraints for object attributes assignment at the time of object creation. The fourth is constraints for object attribute modification after the object has been created. Also, the CPL requires to define *attributes* which can be of two types: *set valued* and *atomic*. For example, *roles* associated with any user is a set attribute as the user may have different roles which can be activated based on some policy. On the other hand *createdBy* is an atomic attribute which identifies the user who has created an object. In [4], it has been shown that DAC, MAC and RBAC (flat and hierarchical) can be implemented using the unified ABAC_α model.

IV. ABAC_β Model

The overall structure of the ABAC_β model is shown in figure 2. The basic components of the ABAC_α remains same in the ABAC_β model. The extensions proposed by [2] are as follows:

1) **Context and Context Attributes**: Context is a single entity and possesses a finite set of contextual attributes. These attributes cannot be associated with user, subject or object. In general, contextual attributes are system dependent and may require continuous enforcement of the authorization policy. Examples of these attributes are *time*, *cpu usage*, *free disk space* etc.
2) **Meta Attributes**: In ABAC_β, attributes can also be associated with other existing attributes. These attributes are known as `Meta Attributes`. Example of meta attributes are the *risk level* of general attributes, the *task* associated with roles etc.
3) **Configuration Points**: A new configuration point is added in ABAC_β model where subject attributes can also be modified at some some point after their creation. This is known as constraint on subject attribute at modification time.

In the following section we present an OWL representation of ABAC_α and show how different attributes cab be defined to implement classical access control models in terms of the ABAC_α model. Later in section VI we demonstrate representation of ABAC_β model in OWL.

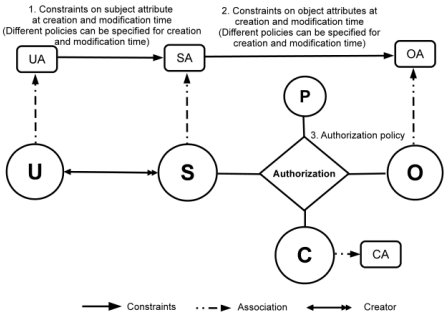

Fig. 2. ABAC_β Model Structure as Proposed by Xin *et. al* [2]

V. Representation of ABAC_α Model in OWL

ABAC_α model is a unified approach to provide classical access controls based on generic attributes. In this section, we first provide representations for basic constructs of ABAC. The implementation for DAC, MAC, RBAC_0 and RBAC_1 along with authorization policy configuration point is shown next. We then show how remaining three configuration points can be configured. The possibility of going beyond ABAC_α model by incorporating contextual attribute is discussed in last subsection.

A. Basic Constructs of ABAC_α

The basic constructs correspond to the various component of ABAC_α model mentioned in the previous section. These are represented as OWL classes using $n3$ as follows:

```
User a owl:class.

Subject a owl:class.

Object a owl:class.

Permission a owl:class.
```

The *attributes* for users, subject and objects are defined as properties:

```
UA a owl:ObjectProperty.

SA a owl:ObjectProperty.

OA a owl:ObjectProperty.
```

We also need to define a `RequestedAction` which corresponds to the request to access a specific object. In order to represent this we have imported the existing ROWLBAC [6] ontology and extended its Action class. The Action class defined in ROWLBAC is as follows:

```
Action a rdfs:Class.
subject a rdfs:Property, owl:FunctionalProperty;
  rdfs:domain Action;
  rdfs:range Subject.
object a rdfs:Property, owl:FunctionalProperty;
  rdfs:domain Action;
  rdfs:range Object.
```

We have extended it to create a RequestedAction class as follows:

```
RequestedAction a owl:class;
  rdfs:subClassOf Action.
permission a rdfs:Property, owl:FunctionalProperty;
  rdfs:domain rbac:Action;
  rdfs:range Permission.
```

Any individual who belongs to the RequestedAction class will have exactly one subject and it tries to acquire exactly one permission on exactly one object. Based on the inference process, it is decided if the `RequestedAction` is a `PermittedAction` or `ProhibitedAction`.

Apart from standard namespaces, various other namespaces are defined and used in our sample ontologies. The `rbac` namespace is required to use ROWLBAC ontology. Each of the classical access control model is defined in a separate namespace. These namespaces are `abdac` (*attribute based* DAC), `abmac` (*attribute based* MAC) and `abrbac` (*attribute based* RBAC). The `data` namespace, defined for each model, contains sample instantiations. For RBAC, we have also defined a `acadDomain` namespace to capture role hierarchy for academic domain.

B. Discretionary Access Control (DAC)

Discretionary protection governs the access of user to the information on the basis of user's identity and authorizations (or rules) that specify, for each user (or group of users) and each object in the system,the access modes (read, write, or execute) the user is allowed on the object [13]. The mechanism is usually implemented by means of Access Control Lists (ACLs). For each access mode there is a corresponding ACL. Users are added to ACL based on the discretion of the owner of the object.

We define a property `subCreator` which binds the subjects to the user who created it. This is required as objects are accessed by subjects and we need to check if their creator is in the ACL for the desired permission.

```
subCreator a owl:ObjectProperty;
  rdfs:domain Subject;
  rdfs:range User.
```

The ACLs serves as the *attributes* for the objects and play a key role in the access control. There is a one-to-one correspondence between permission and object attribute. For example, we may have `reader` and `writer` attributes for `read` and `write` permissions as follows:

```
reader a owl:ObjectProperty;
  rdfs:subPropertyOf abdac:OA;
  rdfs:domain abdac:Object;
  rdfs:range abdac:User.

writer a owl:ObjectProperty;
  rdfs:subPropertyOf abdac:OA;
  rdfs:domain abdac:Object;
  rdfs:range abdac:User.
```

The `RequestedAcion` is created for accessing an object. For example, if bob wants to write to the project plan, its `RequestedAcion` is created as follows:

```
writeProjectPlanBob a abdac:RequestedAction;
  abdac:subject bob;
  abdac:permission write;
  abdac:object projectPlan.
```

The enforcement of DAC ensures that access is granted based on ACL *attributes*. For the policy *An object can be written only by subjects whose users are in the writer list of this object*, the access rule is specified as:

```
{ ?A a abdac:RequestedAction;
        abdac:subject ?S;
        abdac:object ?O;
        abdac:permission ?P.
  ?S abdac:subCreator ?U.
  ?O data:writer ?U.
  ?P rdfs:label "writeAccess"^^xsd:String.

} => { ?A a abdac:PermittedAction }.
```

The EYE [1] reasoner is used to decide this policy rule and infers if some `RequestedAction` results in `PermittedAction`. The reasoner can be invoked from command line as:

```
eye.sh rule.n3 data.n3 abdac.n3 --pass --nope >
output.n3
```

Here, policy rules are specified in file `rule.n3` while DAC model and its instantiations are spefied in files `abdac.n3` and `data.n3` respectively. Accordingly if Bob happens to be as the `writer` *attribute* of *projectPlan*, the `RequestedAction` would result as the `PermittedAction` in output file.

C. Mandatory Access Control (MAC)

In Mandatory Access Control (MAC), users and objects are assigned security levels. These security levels are called as *clearance* for users and *classification* for objects. Access to a particular object by a subject is permitted only if some relationship exists between these two. The relationship is specified by some security model e.g. Bell LaPadula [14].

The security levels are defined using OWL classes in our ontology. The properties `clearanceLevel` and `classificationLevel` map the individuals of these classes to a numeric value for comparisons.

```
Clearance a owl:Class;
  rdfs:subClassOf
  [ a owl:Restriction;
      owl:onProperty clearanceLevel;
      owl:cardinality "1"^^xsd:nonNegativeInteger
  ].

Classification a owl:Class;
```

```
  rdfs:subClassOf
  [ a owl:Restriction;
      owl:onProperty classificationLevel;
      owl:cardinality "1"^^xsd:nonNegativeInteger
  ].
```

In ABAC$_\alpha$ approach, subject clearance and object sensitivity serve as the *attributes* for subjects and objects. Properties `sclearance` and `sensitivity` bind subjects and objects to their security level as follows:

```
sclearance a owl:ObjectProperty;
  rdfs:subPropertyOf SA;
  rdfs:domain Subject;
  rdfs:range Clearance.

sensitivity a owl:ObjectProperty;
  rdfs:subPropertyOf OA;
  rdfs:domain Object;
  rdfs:range Classification.
```

These classes are instantiated based on domain specific implementation requirements. For example, an organisation may want to implement Bell LaPadula [14] model where subject clearance and object classification come from the same set. For a four element security level set, it can be implement them as follows:

```
# Top Secret
TS a abmac:Clearance, abmac:Classification;
  abmac:clearanceLevel 3;
  abmac:classificationLevel 3.

# Secret
S a abmac:Clearance, abmac:Classification;
  abmac:clearanceLevel 2;
  abmac:classificationLevel 2.

# Confidential
C a abmac:Clearance, abmac:Classification;
  abmac:clearanceLevel 1;
  abmac:classificationLevel 1.

# Unrestricted
UR a abmac:Clearance, abmac:Classification;
  abmac:clearanceLevel 0;
  abmac:classificationLevel 0.
```

The enforcement of policy is then straightforward. For example, the *read* policy rule as per the Bell LaPadula model [14] is: *A subject is allowed to acquire a read permission on a object if the object sensitivity level is less than or equal to the subject clearance*. This is implemented as follows:

```
{ ?A a abmac:RequestedAction;
         abmac:subject ?S;
         abmac:object ?O;
         abmac:permission ?P.
  ?S abmac:sclearance ?sc.
  ?sc abmac:clearanceLevel ?sl.
  ?O abmac:sensitivity ?oc.
  ?oc abmac:classificationLevel ?ol.
  ?P rdfs:label "readAccess"^^xsd:String.
  ?ol math:lessThan ?sl.
} => { ?A a abmac:PermittedAction }.
```

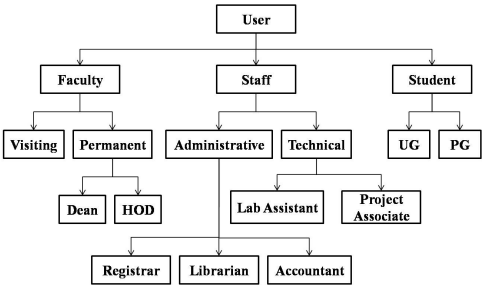

Fig. 3. Example Role Hierarchy

D. Role Based Access Control (RBAC)

The ABAC$_\alpha$ model support only flat and hierarchical RBAC which are termed as RBAC$_0$ and RBAC$_1$ respectively. The model does not strive to provide other types of RBAC including dynamic separation of duties.

In this section, we first present an example domain and then describe how RBAC$_0$ and RBAC$_1$ can be represented in OWL. We leverage the work done by Finin et al. [6].

1) Example Domain: We have considered a small portion of an academic scenario as an example domain for our ontology development. Figure 3 depicts the role hierarchy in such a domain.

The role hierarchy is self explanatory and defined in the `acadDomain` ontology. There are different privileges associated with each role. For example, a user with `Faculty` role is allowed to perform `giveGrades` action which users with different roles may not be allowed to do. Similarly different printers may be accessed by different roles based on the place they are located in.

2) Roles as Attributes: In ABAC$_\alpha$ roles serve as *attributes* and access control is implemented based on the privileges associated with these attributes. Our ontology uses ROWLBAC ontology for creating and activating roles. We have defined roles for users, subjects and objects as follows:

```
# User attributes are user roles
uRole a owl:ObjectProperty;
  rdfs:subPropertyOf UA;
  rdfs:domain User;
  rdfs:range rbac:Role.

# Subject attributes are subject roles
sRole a owl:ObjectProperty;
  rdfs:subPropertyOf SA;
  rdfs:domain Subject;
  rdfs:range rbac:Role.

# Object attributes are object roles
oRole a owl:ObjectProperty;
  rdfs:subPropertyOf OA;
  rdfs:domain Object;
  rdfs:range rbac:Role.
```

Roles are created by defining individuals of the `Role` and its sub classes. For example, `PermanentFaculty` and

VisitingFaculty roles are defined as:

```
PermanentFaculty  a owl:Class;
  rdfs:subClassOf Faculty.
ActivePermanentFaculty a rbac:ActiveRole;
   rdfs:subClassOf PermanentFaculty.
PermanentFaculty rbac:activeForm
ActivePermanentFaculty.

VisitingFaculty a owl:Class;
  rdfs:subClassOf Faculty.
ActiveVisitingFaculty a rbac:ActiveRole;
   rdfs:subClassOf VisitingFaculty.
VisitingFaculty rbac:activeForm
ActiveVisitingFaculty.
```

At this point the static separation of duties can be provided using OWL's disjointWith feature. For example, the static separation between PermanentFaculty and VisitingFaculty is provided as:

```
PermanentFaculty owl:disjointWith VisitingFaculty.
```

The access control policies are defined by creating object roles and specifying which roles are allowed to access that object:

```
# creating printer object which can be accessed by
# any Faculty or Student
printerGCL a abrbac:Object;
 abrbac:orole acadDomain:Faculty,
acadDomain:Student.
```

3) Enforcement of RBAC: For flat RBAC the access to a particular object is permitted if there exists *some* role attribute of the subject which is listed in the role attributes of the object for requested permission. The policy to enforce print access to a printer is defined as follows:

```
{ ?A a abrbac:RequestedAction;
           abrbac:subject ?S;
           abrbac:object ?O;
           abrbac:permission ?P.
  ?P rdfs:label "printAccess"^^xsd:String.
  ?S abrbac:srole ?r.
  ?O abrbac:orole ?r.

} => { ?A a abrbac:PermittedAction }.
```

In hierarchical RBAC, we also need to check if there exists *some* subject role which has a child-parent relationship with the object role attributes. For example, if a printer can be accessed by any Faculty then a subject with PermanentFaculty role must be allowed to access the printer as being PermanentFaculty implies Faculty. This is done as:

```
{ ?A a abrbac:RequestedAction;
           abrbac:subject ?S;
           abrbac:object ?O;
           abrbac:permission ?P.
  ?P rdfs:label "printAccess"^^xsd:String.
  ?S abrbac:srole ?r1.
  ?O abrbac:orole ?r2.
  ?r1 rdfs:subClassOf ?r2.

} => { ?A a abrbac:PermittedAction }.
```

The EYE [1] reasoner infers the hierarchical relationship among roles and decides for permitted actions. In case when role hierarchy is very large, policy rules may also scale up with diverse complexity. Clearly, we need efficient reasoner to handle these situations.

E. Other Policy Configuration Points

The ABAC$_\alpha$ model provides four policy configuration points. The first one, authorization policy, has been explained in previous subsections. This subsection covers remaining three configuration points:

1) Constraint for Subject Attribute at Creation time (ConstrSub): This configuration point sets the attrubutes of the subjects at the time when they are created. The subject attributes are consistent with the attributes of the creating user. This consistency is guided by the underlying access control model. We define ConstrSubAction as a class which has a user, a subject and one pair of attribute name and attribute value pair.

```
ConstrSubAction a rdfs:Class.
subjectConstrSub a rdfs:Property,
owl:FunctionalProperty;
  rdfs:domain ConstrSubAction;
  rdfs:range Subject.
userConstrSub a rdfs:Property,
owl:FunctionalProperty;
  rdfs:domain ConstrSubAction;
  rdfs:range Object.
attrNameConstrSub a rdfs:Property,
owl:FunctionalProperty;
  rdfs:domain ConstrSubAction;
  rdfs:range xsd:String.
attrValueConstrSub a rdfs:Property,
owl:FunctionalProperty;
  rdfs:domain ConstrSubAction;
  rdfs:range <attrClassName>.
```

Here, the range of attrValueConstrSub is defined as the set of individuals of the desired attribute class. For each proposed attribute name and attribute value pair, a ConstrSubAction is created. The policy, based on the access control model, decides if that action is permitted. For DAC, we do not require this feature as the access grant is decided based on SubCreator attribute. For MAC, we create a ConstrSubAction as follows:

```
addAttrCindy a abmac:ConstrSubAction;
  abmac:subjectConstrSub cindy;
  abmac:userConstrSub Cindy;
  abmac:attrNameConstrSub "sclearance"^^xsd:String;
  abmac:attrValueConstrSub TS.
```

Here the range of attrValueConstrSub is Clearance. This action is requested by user Cindy to assign Top Secret (TS) clearance attribute value for

its subject cindy. The attribute name is sclearance. The policy rule for this attribute constraint is as follows:

```
{ ?A a abmac:ConstrSubAction;
   abmac:userConstrSub ?U;
           abmac:subjectConstrSub ?S;
           abmac:attrNameConstrSub ?An;
           abmac:attrValueConstrSub ?Av.
  ?An string:matches "sclearance".
  ?U abmac:uclearance ?uc.
  ?uc abmac:clearanceLevel ?ul.
  ?Av abmac:clearanceLevel ?Avl.
  ?ul math:notLessThan ?Avl.
} => { ?A a abmac:PermittedConstrSub }.
```

The above policy rule ensures that the proposed clearance level of the subject is permitted only if it is same or less than the clearance level of the user.

Similar policy rules can be written for flat and hierarchical RBAC. In this case the range of attrValueConstrSub is Role. For flat RBAC, we check if the proposed subject role is an existing user role while for hierarchical RBAC, role inheritence is also checked in the same way as it was done in the case of authorization policy. A sample policy rule for flat RBAC is:

```
{ ?A a abrbac:ConstrSubAction;
   abrbac:userConstrSub ?U;
           abrbac:subjectConstrSub ?S;
           abrbac:attrNameConstrSub ?An;
           abrbac:attrValueConstrSub ?Av.
  ?An string:matches "srole".
  ?U  abrbac:urole ?Av.
} => { ?A a abrbac:PermittedConstrSub }.
```

2) Constraint for Object Attribute at Creation time (ConstrObj): This configuration point sets the attrubutes of the objects at the time when they are created. We define ConstrObjAction as a class which has a subject, an object and one pair of attribute name and attribute value pair. For DAC, a subject is allowed to specify members of the access control lists of an object at the time of its creation. A subject can specify another attribute for the object such as createdby. However, the value of this attribute must be same as the creator of the subject. This can be specified in the form of a policy rule as:

```
{ ?A a abdac:ConstrObjAction;
   abdac:subjectConstrObj ?S;
           abdac:objectConstrObj ?O;
           abdac:attrNameConstrObj ?An;
           abdac:attrValueConstrObj ?Av.
  ?An string:matches "createdby".
  ?S abdac:subCreator ?Av.
} => { ?A a abdac:PermittedConstrObj }.
```

Following Bell LaPadula model for MAC, a subject can create an object with the security level being same or higher than the security level of the subject. This is specified in a policy rule as:

```
{ ?A a abmac:ConstrObjAction;
   abmac:subjectConstrObj ?S;
           abmac:objectConstrObj ?O;
           abmac:attrNameConstrObj ?An;
           abmac:attrValueConstrObj ?Av.
  ?An string:matches "sensitivity".
  ?S abmac:sclearance ?sc.
  ?sc abmac:clearanceLevel ?sl.
  ?Avs abmac:classificationLevel ?Avl.
  ?Avl math:notLessThan ?sl.
} => { ?A a abmac:PermittedConstrObj }.
```

This feature is not application to RBAC as it does not talk about object attribute assignment by subjects.

3) Constraint for Object Attribute at Modification time ConstrObjMod: This configuration point modifies the attrubutes of the objects at some point of time after their creation. We define ConstrObjModAction as a class which has a subject, an object and one pair of attribute name and attribute value pair. In DAC, the user who created an object, *i.e.*, the owner of the object can modify its access control lists. MAC (with tranquility) does not permit modification of an objects classification. $RBAC_0$ and $RBAC_1$ do not speak to this issue.

For example, in DAC, a policy rule to check if a reader can be added to the access control list of an object can be written as:

```
{ ?A a abdac:ConstrObjModAction;
   abdac:subjectConstrObjMod ?S;
           abdac:objectConstrObjMod ?O;
           abdac:attrNameConstrObjMod ?An;
           abdac:attrValueConstrObjMod ?Av.
  ?An string:matches "reader".
  ?S abdac:subCreator ?Sc.
  ?O abdac:createdby ?Sc.
} => { ?A a abdac:PermittedConstrObjMod }.
```

Here the policy rule checks if the subject creator is same as the object creator. In that case, the object attribute modification is granted.

F. Beyond $ABAC_\alpha$

The $ABAC_\alpha$ model covers only flat and hierarchical RBAC models. It does not talk about dynamic access controls like *dynamic separation of duties* and other variants of RBAC. However, the ROWLBAC work has already shown that such properties can be modeled in OWL. Of greater interest are Contextual and environmental factors, which are also not covered in the scope of $ABAC_\alpha$. These aspects of access control leads to a more complete model called $ABAC_\beta$ [2] that has been proposed. However, it is possible to incorporate some contextual attribute in the current representation scheme.

For example, consider the policy: *A PG student is allowed to print on a printer placed in specialized computing lab.*. With the help of some addition attributes, the policy can be enforced easily. We define a property locatedIn and use it bind objects to their locations. After adding PGStudent to the object roles, the policy rule can be written as:

```
{ ?A a abrbac:RequestedAction;
          abrbac:subject ?S;
          abrbac:object ?O;
          abrbac:permission ?P.
  ?P rdfs:label "printAccess"^^xsd:String.
  ?S abrbac:srole ?r1.
  ?O abrbac:orole ?r2.
  ?r1 rdfs:subClassOf ?r2.
  ?O abrbac:locatedIn 'specializedCompLab'.

} => { ?A a abrbac:PermittedAction }.
```

This shows how a specific contextual attribute such as location can be added in the OWL formalism we've developed for $ABAC_\alpha$. This shows the path we hope to take to more fully capture such attributes, and eventually the $ABAC_\beta$ model in OWL.

VI. Representation of $ABAC_\beta$ Model in OWL

A. *Context and Context Attributes*

The context entity is represented as a basic OWL class:

```
Context a owl:Class.
```

Next, we associate a finite set of attributes with `Context`. These are defined in terms of OWL properties:

```
CA a owl:ObjectProperty.

contextDate
  a owl:ObjectProperty;
  rdfs:subPropertyOf CA;
  rdfs:domain Context;
  rdfs:range xsd:date.

contextTime
  a owl:ObjectProperty;
  rdfs:subPropertyOf CA;
  rdfs:domain Context;
  rdfs:range event:Event.

contextDay
  a owl:ObjectProperty;
  rdfs:subPropertyOf CA;
  rdfs:domain Context;
  rdfs:range event:Day.
```

Here we have defined three different context attributes: *contextDate*, *contextTime* and *contextDay*. These represent the date , time and day at which the request is made for accessing an object. The range of *contextTime* and *contextDay* is deifined in `event` ontology [15]. This ontology also defines other entities related to time.

It is straightforward to include context attribute in the policy enforcement. For example consider the policy: *The system design plan can be read from SIT Lab by permanent faculty on weekdays from 10:00 AM to 5:00 PM.*. This policy rule is represented as follows:

```
{ ?A a abrbac:RequestedAction;
        abrbac:subject ?S;
        abrbac:object ?O;
        abrbac:permission ?P;
        abrbac:context ?C.
  ?P rdfs:label "readAccess"^^xsd:String.
  ?S abrbac:srole ?r.
  ?O abrbac:readRole ?r.
  ?O abrbac:accessFromLoc ?locList.
  ?S abrbac:sLocation ?loc.
  ?loc list:in ?locList.
  ?C abrbac:contextDay ?d.
  ?d list:in ("Monday" "Tuesday" "Wednesday"
"Thursday" "Friday").
  ?C abrbac:contextTime ?t.
  acadDomain:workHour1 time:includes ?t.

} => { ?A a abrbac:PermittedAction  }.
```

Here `workHour1` is defined to include the specified time as follows:

```
workHour1 a event:Event;
  abrbac:startTime "10:00:00"^^xsd:time;
  abrbac:endTime "17:00:00"^^xsd:time.
```

VII. Conclusion

In this paper we have shown how the *Attribute Based Access Control* model can be represented using *Web Ontology Language* (OWL). This is a starting step towards formally specifying and enforcing machine understable policies that can be captured in the ABAC model, which is one of the most general access control models available today. The unified $ABAC_\alpha$ model is proven to provide DAC, MAC and RBAC. We have written ontologies for each of these classical control models. The policy enforcement is shown using inference based resoner EYE [1]. The performance of reasoning process is subject to further analysis. It would be interesting to see how reasoning process scales up as the number of classes and rules increases. We would explore this issue in our future work.

The basic $ABAC_\alpha$ model is not complete and does not cover static/dynamic separation of duties. It also lacks in subjects carrying additional attributes other than the corresponding users to reflect contextual information. All these features are represented in a more complex, complete and generic next level model: $ABAC_\beta$ [2]. In ongoing work we are modeling more complex policies by capturing the $ABAC_\beta$ model in OWL. It would eventually lead to a complete access control system in place with more generic features.

Acknowledgment

This work was partially done while author Joshi was on sabbatical at the Center of Excellence in Cyber Systems and Information Assurance, EE Department, at IIT Delhi. Support for his sabbatical visit from IIT Delhi is greatfully acknowledged. Support was also provided in part to Joshi by NSF award *CNS 1228673*. The authors appreciate the discussions with Dr. G. Athithan, Ms. Anu Khosla, Prof. Tim Finin, and Dr. Lalana Kagal that helped in developing some of these approaches. Author Sharma would also like to acknowledge the help provided by Prof. Kolin Paul and Prof. S.K. Gupta.

REFERENCES

[1] R. Verborgh, Jos De Roo, *Drawing Conclusions from Linked Data on the Web The EYE Reasoner*, IEEE Software, May-June 2015.

[2] Xin Jin, *Attribute-Based Access Control Models and Implementation in Cloud Infrastructure as a Service*, Phd Dissertation, The University of Texas, San Antonio, May 2014.

[3] J. Bradshaw, A. Uszok, M. Breedy, L. Bunch, T. Eskridge, P. Feltovich, M. Johnson, J. Lott, and M. Vignati, *The KAoS Policy Services Framework*, in Eighth Cyber Security and Information Intelligence Research Workshop (CSIIRW 2013), 2013.

[4] Xin Jin, Ram Krishnan and Ravi Sandhu, *A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC*, Proceedings of 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2012), Paris, France, July, 2012.

[5] Rodolfo Ferrini, Elisa Bertino, *Supporting RBAC with XACML+OWL*, SACMAT'09, Stresa, Italy, June 2009.

[6] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. H. Winsborough, and B. Thuraisingham. *ROWLBAC - Representing Role Based Access Control in OWL*, Proceedings of the 13th Symposium on Access control Models and Technologies, June 2008.

[7] T. Berners-Lee, D. Connolly, E. Prud?homeaux, Y. Scharf, *Experience with N3 rules*. In W3C Rules language Workshop, 2005.

[8] L. Kagal, T. Berners-Lee, *Rein: Where Policies Meet Rules in the Semantic Web*, Technical Report, MIT, 2005.

[9] M. Dean and G. Schreiber. *OWL Web Ontology Language Guide, 2004*. W3C Recommendation 2004, http://www.w3.org/TR/owl-guide/.

[10] S. Godik, T. Moses. *OASIS extensible access control markup language (XACML). OASIS Committee Specification cs-xacml-specification-1.0*, November 2002.

[11] L. Kagal, *Rei : A Policy Language for the Me-Centric Project* HP Labs, Tech. Rep., Sep. 2002.

[12] T. Berners Lee, *Cwm*, 2000.

[13] R. Sandhu, P. Samarati. *Access Control: Principals and Practice*, IEEE Communications Magazine, September 1994.

[14] D. E. Bell, L. J. LaPadula, *Secure Computer Systems: Mathematical Foundations*, vol. 1, 1973: MITRE Corp.

[15] *Event Ontology: http://eulersharp.sourceforge.net/2003/03swap/event.*