Ontology-Based Policy Specification and Management

Wolfgang Nejdl[1], Daniel Olmedilla[1], Marianne Winslett[2], and Charles C. Zhang[2]

[1] L3S Research Center and University of Hannover, Germany
{nejdl,olmedilla}@l3s.de
[2] Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
{winslett,cczhang}@cs.uiuc.edu

Abstract. The World Wide Web makes it easy to share information and resources, but offers few ways to limit the manner in which these resources are shared. The specification and automated enforcement of security-related policies offer promise as a way of providing controlled sharing, but few tools are available to assist in policy specification and management, especially in an open system such as the Web, where resource providers and users are often strangers to one another and exact and correct specification of policies will be crucial. In this paper, we propose the use of ontologies to simplify the tasks of policy specification and administration, discuss how to represent policy inheritance and composition based on credential ontologies, formalize these representations and the according constraints in Frame-Logic, and present POLICYTAB, a prototype implementation of our proposed scheme as a Protégé plug-in to support policy specification.

1 Introduction

Open distributed environments like the World Wide Web offer easy sharing of information, but provide few options for the protection of sensitive information and other sensitive resources, such as Web Services. Proposed approaches to controlling access to Web resources include XACML [4], SAML [5], WS-Trust [3] and Liberty-Alliance[1]. All of these approaches to trust management rely on the use of vocabularies that are shared among all the parties involved, and declarative policies that describe who is allowed to do what. Some of these approaches also recognize that trust on the Web and in any other system where resources are shared across organizational boundaries must be *bilateral*.

Specifically, the Semantic Web provides an environment where parties may make connections and interact without being previously known to each other. In many cases, before any meaningful interaction starts, a certain level of trust must be established from scratch. Generally, trust is established through exchange of information between the two parties. Since neither party is known to the other, this trust establishment process should be bi-directional: both parties may have sensitive information that they are reluctant to disclose until the other party has proved to be trustworthy at a certain level. As there are more service providers emerging on the Web every day, and people are performing more sensitive transactions (e.g., financial and health services) via the Internet, this need for building mutual trust will become more common.

To make controlled sharing of resources easy in such an environment, parties will need software that automates the process of iteratively establishing bilateral trust based

on the parties' access control policies, i.e., *trust negotiation* software. Trust negotiation differs from traditional identity-based access control and information release systems mainly in the following aspects:

1. Trust between two strangers is established based on parties' properties, which are proved through disclosure of digital credentials.
2. Every party can define access control and release policies (*policies*, for short) to control outsiders' access to their sensitive resources. These resources can include services accessible over the Internet, documents and other data, roles in role-based access control systems, credentials, policies, and capabilities in capability-based systems. The policies describe what properties a party must demonstrate (e.g., ownership of a driver's license issued by the State of Illinois) in order to gain access to a resource.
3. Two parties establish trust directly without involving trusted third parties, other than credential issuers. Since both parties have policies, trust negotiation is appropriate for deployment in a peer-to-peer architecture such as the Semantic Web, where a client and server are treated equally. Instead of a one-shot authorization and authentication, trust is established incrementally through a sequence of bilateral credential disclosures.

A trust negotiation process is triggered when one party requests to access a resource owned by another party. The goal of a trust negotiation is to find a sequence of credentials $(C_1, \ldots, C_k, R)$, where R is the resource to which access was originally requested, such that when credential C_i is disclosed, its policy has been satisfied by credentials disclosed earlier in the sequence or to determine that no such credential disclosure sequence exists.

The use of declarative policies and the automation of the process of satisfying them in the context of such a *trust negotiation process* seem to be the most promising approach to providing controlled access to resources on the Web. However, this approach opens up a new and pressing question: what confidence can we have that our policies are correct? Because the policies will be enforced automatically, errors in their specification or implementation will allow outsiders to gain inappropriate access to our resources, possibly inflicting huge and costly damages. Unfortunately, real-world policies [10] tend to be as complex as any piece of software when written down in detail; getting a policy right is as hard as getting a piece of software correct, and maintaining a large number of them is only harder.

In this paper, we take an ontology-based approach to address this problem. Section 2 discusses the use of ontologies for providing abstraction and structuring for policy specification, and further formalizes these concepts and constraints in Frame-Logic / F-Logic [17]. Section 3 describes our proof-of-concept implementation, POLICYTAB, a Protégé [2] plug-in to support policy specification. We discuss related work in section 4 and give future research directions and conclusions in section 5.

2 Using Ontologies to Ease Policy Specification and Management

Ontology-based structuring and abstraction help maintain complex software, and so do they with complex sets of policies. In the context of the Semantic Web, ontologies pro-

vide formal specification of concepts and their interrelationships, and play an essential role in complex web service environments [7], semantics-based search engines [12] and digital libraries [21].

One important purpose of these formal specifications is sharing of knowledge between independent entities. In the context of trust negotiation, we want to share information about credentials and their attributes, which is needed for establishing trust between negotiating parties. Figure 1 shows a simple example ontology for credential IDs.

Each credential class can contain its own attributes; e.g., a Cisco Employee ID credential has three attributes: `name`, `rank` and `department`. Trust negotiation is attributed-based and builds on the assumption that each of these attributes can be protected and disclosed separately. While in some approaches (e.g. with X.509 certificates) credentials and their attributes are signed together as a whole by the credential issuer, in this paper we will rely on cryptographic techniques such as [19] which allow us to disclose credentials with different granularities, hiding attributes not relevant to a given policy.

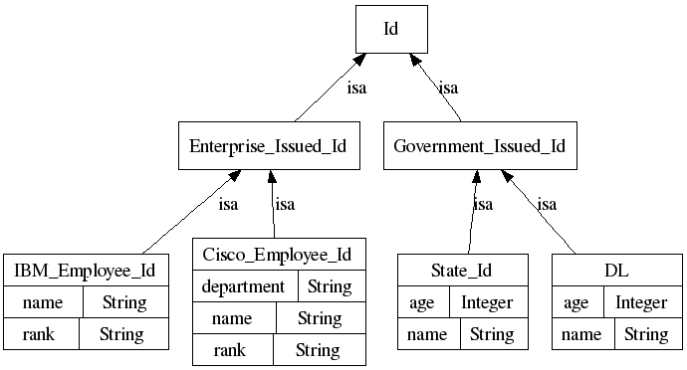

Fig. 1. Simple ID Credential Ontology

In trust negotiation, a party's security policies consist of constraints that the other party has to satisfy; e.g. it has to produce a proof that it owns a certain credential, and that one of the credential attributes has to be within a certain range. Assuming a casino requires any customer's age to be over 21 and requires a state ID to testify that, the policy for its `admits` service can be represented as the following logic program, which uses a simplified version of the PEERTRUST [18, 15] policy language:

Casino:
allowedInCasino(Requester) ←
  type(CredentialIdentifier, "State_Id") @ Issuer @ Requester,
  issuedFor(CredentialIdentifier, Requester) @ Issuer @ Requester,
  age(CredentialIdentifier, Age) @ Issuer @ Requester,
  Age $>$ 21.

In this example, the first two statements in the body of the rule require the requester to prove that he owns a credential of type `State_Id` issued by `Issuer`[3]. If the requester proves that he has it (notice that information about attributes has not been disclosed so far, except for the `issuedFor` attribute), the casino asks for the value of the attribute `age` in the presented credential. Then it verifies whether the requester's age is over 21 and, if successful, admits the requester into the casino.

2.1 Sharing Policies for Common Attributes

Often, credentials share common attributes, and these attributes might share the same policies. Figure 1 shows an example of a simple credential hierarchy, where the concrete credential classes used are depicted in the leaves of the hierarchy. The upper part of the hierarchy represents the different abstract classes: the root represents any ID, which is partitioned into different subclasses according to the issuer of the credential, distinguished between `Government_Issued` and `Enterprise_Issued` IDs. The leaf nodes represent concrete classes which contain the attributes such as `name`, `age`, and `rank`.

This somewhat degenerated hierarchy however does not yet allow for policy re-use. For this we have to exploit attribute inheritance. In our example, all leaf nodes share the `Name` attribute, which therefore can be moved up to the root class `Id`. We are now able to specify common policies for the `Name` attribute at the `Id` level. Similarly, we will move `Rank` up so that it becomes an attribute of `Enterprise_Issued_Id`, and `Age` an attribute of `Government_Issued_Id`. A subclass automatically inherits its superclass's attributes, which might be local or inherited from the superclass's superclass. In the following, we will use Frame-Logic / F-Logic [17] to represent these constraints. So, in the context of F-Logic, we use `type inheritance` (also `structural inheritance`) to represent this constraint, which is defined as

$$If \;\; I \models p[mthd@q_1, \ldots, q_k \approx\!> s] \;\; and \;\; I \models r :: p \;\; then \;\; I \models r[mthd@q_1, \ldots, q_k \approx\!> s]$$

where the symbol $\approx\!>$ denotes either `=>` or `=>>`, I is any F-structure and $r :: p$ represents the fact that "r is *subclass of* p" .

This leads to the refined ontology as described in figure 2, where each leaf node has the same set of attributes as in figure 1, but inherits them from higher levels. This makes it possible to specify shared policies for these shared attributes, similar to method inheritance in object oriented programming languages.

2.2 Composing and Overriding Policies

Now, given such credential ontologies, we can specify security policies at different levels. Being able to inherit and compose these security policies simplifies policy maintenance, though of course we have to distinguish between the case where we compose inherited and local policies and the case where the local policy specified for an attribute

[3] As an extra hint, in the PEERTRUST language, for a statement such that *"lit_i @ Authority"*, *Authority* specifies the peer who is responsible for evaluating lit_i or has the authority to evaluate lit_i. In addition, *Authority* can be a nested term containing a sequence of authorities, which are then evaluated starting at the outermost layer.

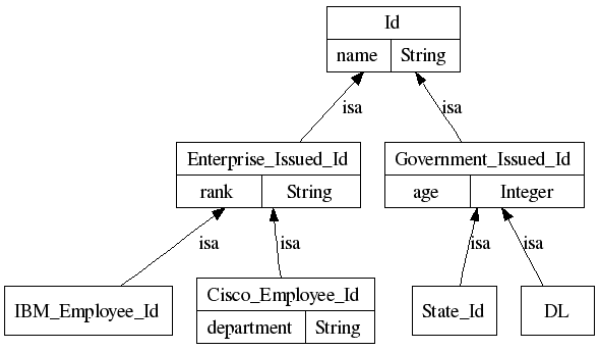

Fig. 2. Refined ID Credential Ontology

of a specific class overrides the policy inherited from a superclass. In this paper we will describe *mandatory policies* and *default polices*.

To model a policy in F-Logic, we have the following signature declaration for the class `policy`

```
policy [
        name  => string,
        value  => string,
        type   => string
]
```

where `name` is the unique name of the policy, `value` is the text that describes the policy (expressed in a suitable policy language)) and `type` describes if the policy is default or mandatory. To express the constraint that `type` can only contain the strings "Default" or "Mandatory" and only one of them at the same time, we define the following integrity constraint

$$false \leftarrow C : policy, C[type \texttt{->} V], not\ V = \text{“}Default\text{”}, not\ V = \text{“}Mandatory\text{”}$$

Moreover, we want to assure that any class in our knowledge base has the possibility to define policies. Therefore we need to declare a meta class called `metaClass` from which all the classes will be an instance.

```
metaClass [
        policySlot     =>> policy,
        overallPolicy  =>> policy
]
```

This meta class contains a property `policySlot` whose value is a set of policies (possibly empty) attached to the class and a property `overallPolicy` whose value

is the set of policies (possibly empty) of all the policies, directly attached to the class and inherited from superclasses, that apply to this class. The derivation rule

$$C2[overallPolicy \texttt{ ->> } P] \leftarrow C2 :: C1, C1[policySlot \texttt{ ->> } P] \tag{1}$$

assures that any policy in a direct superclass is inherited to the subclass. We further have

$$C[overallPoliy \texttt{ ->> } P] \leftarrow C[policySlot \texttt{ ->> } P] \tag{2}$$

to add the policies attached to the the current class ($C :: C$ is not true in F-Logic[4]).

Finally, in order to assure that any class in the knowledge base (except the policy class defined above) will be an instance of `metaClass` we need the following derivation rule

$$C : metaClass \leftarrow not\ C = policy, not\ C : policy \tag{3}$$

Figure 4 depicts the hierarchy of classes and instances in our driver license example.

Mandatory Policies Mandatory policies are used when we want to mandate that policies of a higher level are always enforced at lower levels. Assume the ontology depicted in figure 3 and that we want to hire an experienced driver to accomplish a certain highly classified and challenging task. Before we show the details of the task to an interested candidate, we want the candidate to present a driver's license, which can be proved to satisfy the following mandatory policies as specified at the different levels:

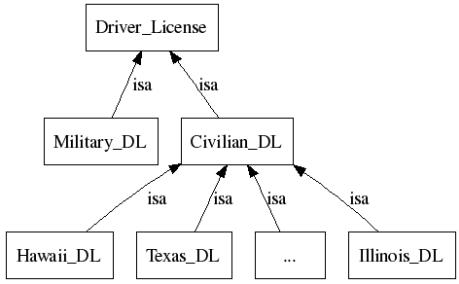

Fig. 3. Driver License Ontology

At the `Driver_License` level, we enforce generic requirements for driver licenses; e.g., a driver license has to be signed by a federally authorized certificate authority and must not have expired.

At the `Civilian_DL` level, we require that the driver license is non-commercial, assuming commercial drivers may have a conflict of interests in the intended task.

At the `Illinois_DL` level, we require that the category of the driver license is not F, assuming F licenses are for farm vehicles only. At the `Military_DL` level, we

[4] In the F-Logic notation, the operator $C1 :: C2$ represents "$C1$ is *subclass of* $C2$" while $C1 : C2$ means "$C1$ is an *instance of* $C2$"

can specify policies such as "the driver license must be for land passenger vehicles" as opposed to fighter planes or submarines.

So for an Illinois driver, the overall policy is: must hold a valid driver license, as qualified by the policy at the Driver_License level; must hold a non-commercial driver license, as required by the Civilian_DL policy; and the driver license must not be for farm vehicles only. The advantage of using mandatory policies here is twofold: first, shared policies such as the generic driver license requirements are only specified once at a higher level, which means a more compact set of policies; second, it gives a cleaner and more intuitive logical structure to policies, which makes the policies easier to specify and manage.

Default Policies Let us now assume that all driver licenses include the specification of driving experience, expressed in years of driving. Suppose that a specific task requires the following policy: in most cases, 4 years' driving experience is required; however, if the driver comes from Texas, he/she needs only 3 years' experience (assuming it is harder to get a driver's license in Texas).

To simplify the specification of this policy, we can use the default policy construct. A superclass's default policy is inherited and enforced by a subclass if and only if the child does not have a corresponding (overriding) policy. In our example, we can specify at the `Driver_License` level that the driving age has to be at least 4 years; then at the `Texas_DL` level, specify an overriding policy that the driving age has to be at least 3 years.

It is of interest to note that the same result can be achieved here without using default policies: we can move the shared 4-year mandatory policy down to *every* concrete driver license class except `Texas_DL`, where we require 3 years. However, the power of policy sharing is lost.

To summarize, on one hand, mandatory policies must be enforced at lower levels in the hierarchy, that is, they can not be overridden. On the other hand, default policies are inheritable, but they can be overridden at lower levels. In order to formalize this, we assume that if two policies have the same value in the property `name`, the most specific one overrides the other one. Taking that into account, we need to refine equation (1) in a way that overridden policies are not included in the overall policy. The derivation rule would be

$$C2[overallPolicy \texttt{->>} P] \leftarrow C2 :: C1, C1[policySlot \texttt{->>} P, \quad (4)$$
$$not(C2[policySlot \texttt{->>} P2, P2[name \texttt{->} N], P[name \texttt{->} N])$$

Finally, only default policies can be overridden and therefore we need the following integrity constraint to avoid that mandatory policies are overridden

$$false \leftarrow C2 : C1, C1[policySlot \texttt{->>} P1], C2[policySlot \texttt{->>} P2], \quad (5)$$
$$P1[name \texttt{->} N, type \texttt{->} \text{“}Mandatory\text{”}], P2[name \texttt{->} N]$$

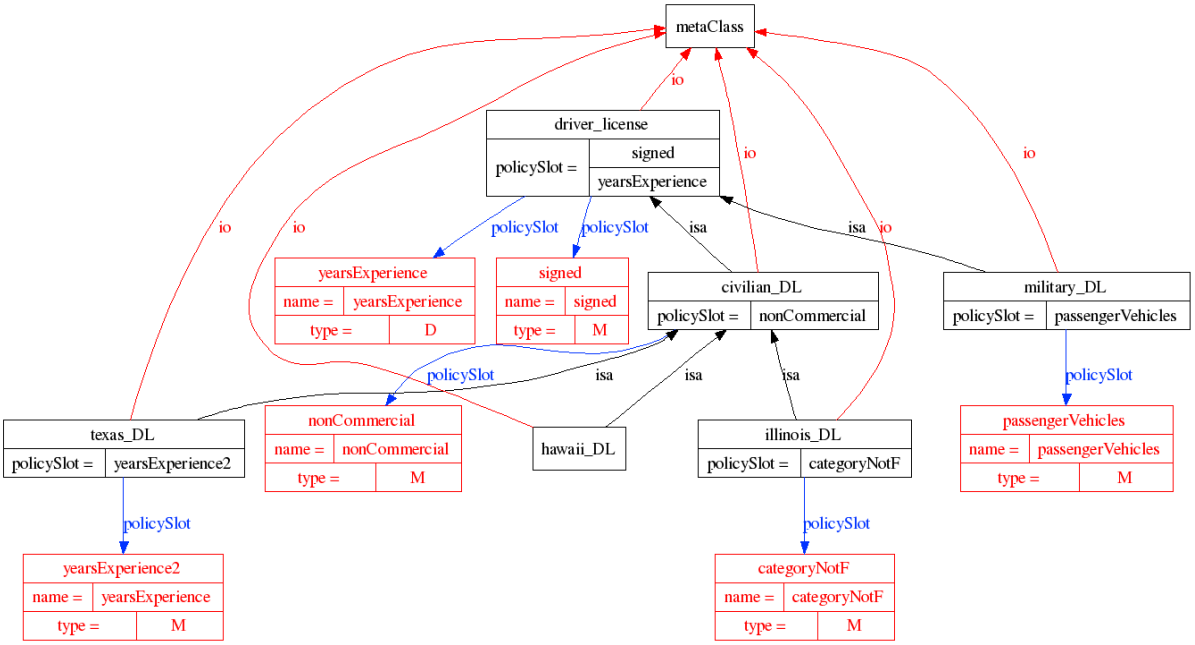

Fig. 4. Driver License Knowledge Base

3 POLICYTAB: Making Protégé a Policy Management Tool

To support policy specification as discussed in the previous sections, we have implemented POLICYTAB (available at http://www.l3s.de/peertrust/, a plug-in for the ontology editor Protégé. We chose Protégé because it is widely used and is extensible by means of plug-ins. The POLICYTAB plug-in adds a new tab to the main window of Protégé (see figure 5 for an example), which consists of the following elements:

- *The Class Relationship Pane.* This panel on the upper left corner of the window displays the existing classes in the knowledge base as a tree.
- *The Superclass Pane.* This panel on the lower left corner shows the superclasses of the class currently selected in the Class Relationship Pane.
- *The PolicyView Form.* This form occupying the upper right part of the window contains the information of the class currently selected in the Class Relationship Pane.
- *The Associated Policies Pane.* This pane at the lower right part of the window displays the policies attached to the current selected class or to the current selected slot.

We describe the PolicyView Form and the Associated Policies Pane in more detail in the following sections.

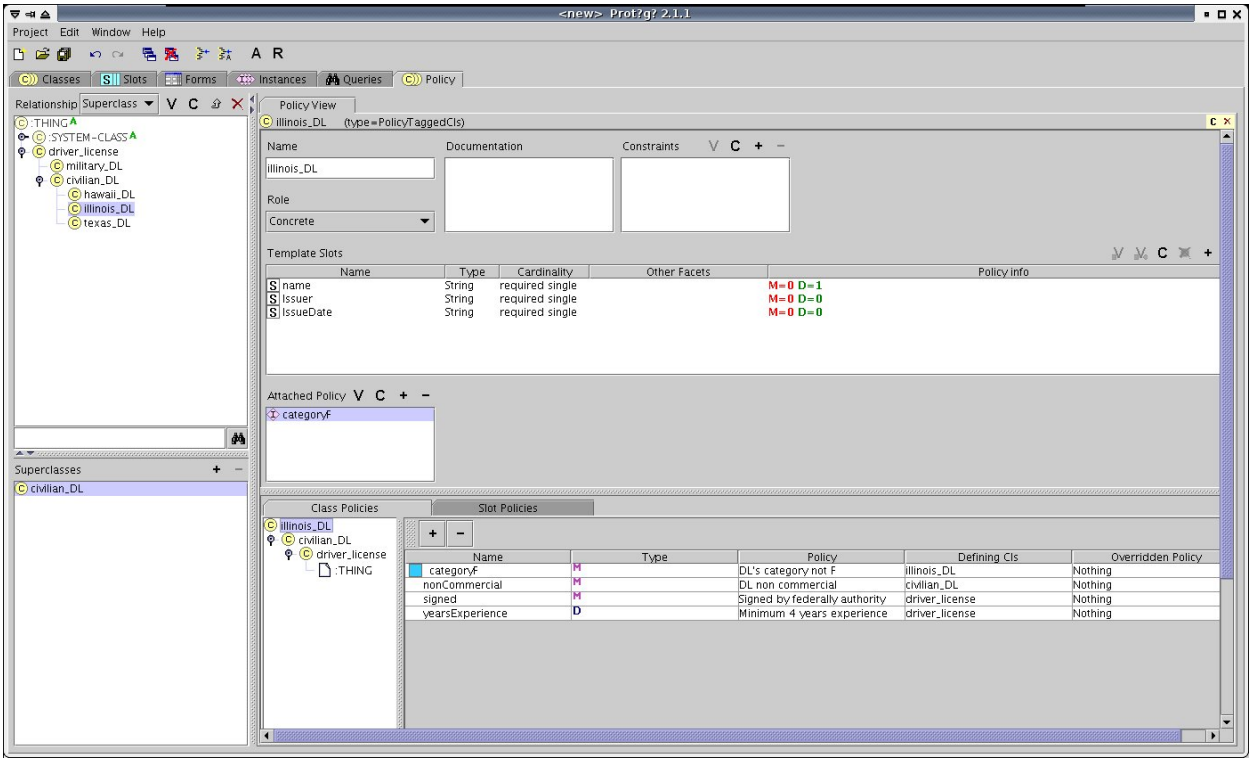

Fig. 5. Screenshot of the POLICYTAB plug-in

3.1 The PolicyView Form

The PolicyView Form contains the information related to the currently selected class (see figure 5). Each class has the usual properties available in Protégé

- *Name:* the name of the class
- *Documentation:* extra comments and explanations
- *Role:* describes if the class is `concrete` or `abstract`
- *Constraints:* specify constraints to the class
- *Template Slots:* show the different properties of the class

In Protégé, a class's properties are called *slots*. To add a slot to the current class, click the + icon for the `Template Slots` table, a slot creation dialog will pop up (see figure 6), where you can specify the new slot's name, type, etc. POLICYTAB automatically checks name conflicts, and does not permit the specification of a slot with the same name as in one of its superclasses.

3.2 The Associated Policies Pane

The Associated Policies Pane displays the policies that apply to the currently selected class, i.e. the overall set of policies that should be satisfied by a requester in order

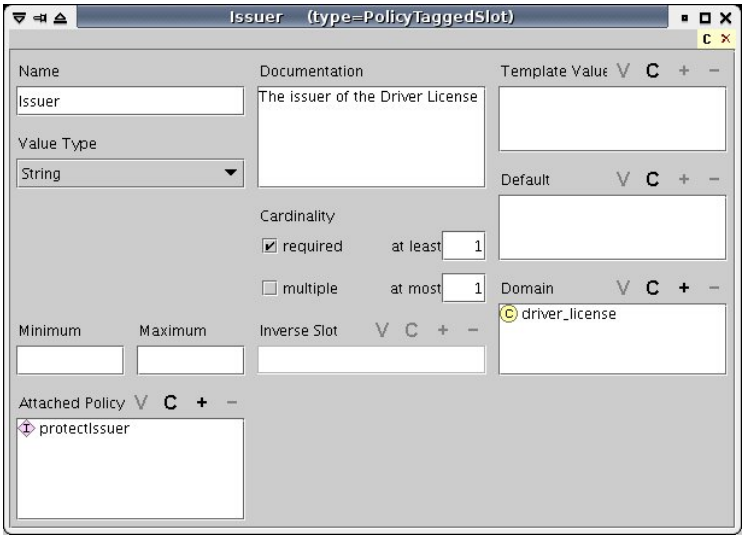

Fig. 6. New Slot Creation

to get access to the resource represented by the class. A policy's type can be either `mandatory` or `default`. Overriding of policies is done by explicitly selecting which class to be overridden (the combobox `Overridden Policy` shows only overridable policies or `Nothing` as valid values)[5]. POLICYTAB's automatic overridability checking prevents the user from unintentionally overriding a mandatory policy and hence reduces policy specification errors.

This pane contains two different tabs: `Class Policies` and `Slot Policies` (see figure 7). Class policies are specified to protect the whole class and correspond to the concepts described in section 2. The slot policies are protectors for each specific property, and have a finer-grained protection. A requester has to satisfy the relevant slot policies in addition to the class polices in order to access a certain property of the class. This is crucial in Trust negotiation as the process relies on disclosure of party's properties, not necessarily whole credentials. The tab `Slot Policies` displays the policies that apply to the slot currently selected in the PolicyView form. Both class policies and slot policies are inherited by the subclasses in the hierarchy.

Once a class is selected in the Class Relationship Pane, the Associated Policies Pane shows this class's inheritance hierarchy as well as its class level policies. For each single policy it displays the name, the type (`mandatory` or `default`), the string with the policy description, the class where that policy is defined and the class whose corresponding policy is overridden (or `Nothing` if there isn't any). Automatic resolution of overriding and inheritance gives the user a clear view of the current class's effective policies, and showing the original defining class of the inherited policies in addition to

[5] As to our knowledge there not exists yet an F-Logic inference engine integrated in Protégé, our current implementation "hard-codes" in the plug-in the inference rules presented in the paper.

the policy tree helps the user understand the policy composition hierarchy and capture the intuitions and implications behind it.

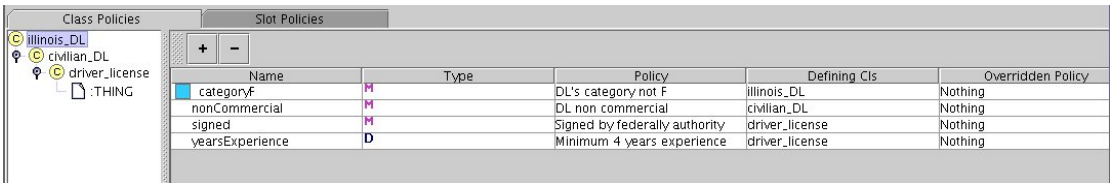

Fig. 7. View of overall policy applicable to a class

4 Related Work

Recent work in the context of the Semantic Web has focused on how to describe security requirements. KAoS and Rei policy languages [16, 22] investigate the use of ontologies for modeling speech acts, objects, and access types necessary for specifying security policies on the Semantic Web. Hierarchies of annotations to describe capabilities and requirements of providers and requesting agents in the context of Web Services are introduced in [11]. Those annotations are used during the matchmaking process to decide if requester and provider share similar security characteristics and if they are compatible.

Ontologies have also been discussed in the context of digital libraries for concepts and credentials [6]. An approach called "most specific authorization" is used for conflict resolution. It states that policies specified on specific elements prevail over policies specified on more general ones. In this paper we explore complementary uses of ontologies for trust negotiation, through which we target iterative trust establishment between strangers and the dynamic exchange of credentials during an iterative trust negotiation process that can be declaratively expressed and implemented. Work done in [9] defines *abstractions* of credentials and services. Those abstractions allow a service provider to request for example a credit card without specifically asking for each kind of credit card that it accepts. We add to this work in the context of policy specification the concept of *mandatory* and *default* policies.

Ontology-based policy composition and conflict resolving have also been been discussed in previous work. Policy inheritance is done by *implication* in [13], but it does not provide any fine-grained overriding mechanism based on class levels. *Default properties* are discussed in [14], short of generalizing the idea to policies. The approaches closest to our default and mandatory policy constructs are the *weak* and *strong* authorizations in [8], where a strong rule always overrides a weak rule, and SPL in [20], which forces the security administrator to combine policies into a structure that precludes conflicts. Compared to these approaches, we find ours particularly simple and intuitive, while its expressiveness well serves general trust negotiation needs.

5 Conclusions and Future Research Directions

Ontologies can provide important supplemental information to trust negotiation agents both at compile time to simplify policy management and composition. This paper has explored some important benefits of using ontologies.

For compile time usage, ontologies with their possibility of sharing policies for common attributes provide an important way for structuring available policies. In this context we propose two useful strategies to compose and override these policies, building upon the notions of mandatory and default policies, and formalize the constraints corresponding to these kinds of policies using F-Logic. We also present a prototype implementation, POLICYTAB, which shows that the proposed policy specification mechanism is implementable and effective.

Future work will investigate multiple inheritance and resolution of conflicting policies in ontology hierarchies, and whether we need disjunction for composing these policies. We are also working on a closer integration into our PEERTRUST system, with suitable import/export facilities to and from POLICYTAB. Finally, an interesting research area to consider in the future is policy validation, i.e. whether the final ontologies plus policy rules are consistent and correct with respect to a set of background constraints.

Acknowledgments

The authors thank Rubén Lara for useful discussions and help in the modeling with F-Logic and the anonymous reviewers for their useful comments. The research of Nejdl and Olmedilla was partially supported by the projects ELENA (http://www.elena-project.org, IST-2001-37264) and REWERSE (http://rewerse.net, IST-506779). The research of Winslett was supported by DARPA (N66001-01-1-8908), the National Science Foundation (CCR-0325951,IIS-0331707) and The Regents of the University of California.

References

1. *Liberty Alliance Project. http://www.projectliberty.org/about/whitepapers.php*.
2. *The Protégé Ontology Editor and Knowledge Acquisition System. http://protege.stanford.edu/*.
3. *Web Services Trust Language (WS-Trust) Specification. http://www-106.ibm.com/developerworks/library/specification/ws-trust/*.
4. Xacml 1.0 specification http://xml.coverpages.org/ni2003-02-11-a.html.
5. Assertions and protocol for the oasis security assertion markup language (saml); committee specification 01, 2002.
6. N. R. Adam, V. Atluri, E. Bertino, and E. Ferrari. A content-based authorization model for digital libraries. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):296–315, 2002.
7. A. Ankolekar. Daml-s: Semantic markup for web services.
8. E. Bertino, S. Jojodia, and P. Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy*, pages 94–109, Oakland, CA, 1996. IEEE Computer Society Press.

9. P. Bonatti and P. Samarati. Regulating Service Access and Information Release on the Web. In *Conference on Computer and Communications Security*, Athens, Nov. 2000.
10. Cassandra policy for national ehr in england. http://www.cl.cam.ac.uk/users/mywyb2/publications/ehrpolicy.pdf.
11. G. Denker, L. Kagal, T. Finin, M. Paolucci, and K. Sycara. Security for daml web services: Annotation and matchmaking. In *Proceedings of the 2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.
12. M. Erdmann and R. Studer. How to structure and access xml documents with ontologies. *Data and Knowledge Engineering*, 36(3), 2001.
13. W. Emayr, F. Kastner, G. Pernul, S. Preishuber, and A. Tjoa. Authorization and access control in iro-db.
14. R. Fikes, D. McGuinness, J. Rice, G. Frank, Y. Sun, and Z. Qing. Distributed repositories of highly expressive reusable knowledge, 1999.
15. R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st First European Semantic Web Symposium*, Heraklion, Greece, May 2004.
16. L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.
17. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.
18. W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: automated trust negotiation for peers on the semantic web. In *Workshop on Secure Data Management in a Connected World (SDM'04)*, Toronto, Aug. 2004.
19. P. Persiano and I. Visconti. User privacy issues regarding certificates and the tls protocol. In *Conference on Computer and Communications Security*, Athens, Nov. 2000.
20. C. Ribeiro and P. Guedes. Spl: An access control language for security policies with complex constraints, 1999.
21. S. B. Shum, E. Motta, and J. Domingue. Scholonto: an ontology-based digital library server for research documents and discourse. *Int. J. on Digital Libraries*, 3(3):237–248, 2000.
22. G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, Rei and Ponder. In *2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.