WS-Policy and Beyond: Application of OWL Defaults to Web Service Policies

Vladimir Kolovski[1] and Bijan Parsia[2]

[1] Department of Computer Science,
University of Maryland, College Park, MD USA,
kolovski@cs.umd.edu
[2] School of Computer Science,
The University of Manchester, UK,
bparsia@cs.man.ac.uk

Abstract. Recently, there has been an increased amount of attention dedicated to WS-Policy - it has become a W3C submission and a working group was formed to standardize the specification. In our previous work, we provided a mapping of WS-Policy to OWL-DL. In this paper, we continue that work by analyzing the operation of policy intersection (determining whether two web service policies are compatible). We show how this operation motivates the use of a non-monotonic extension of OWL in the form of OWL default rules. We discuss our prototype implementation of an OWL defaults reasoner based on Baader and Hollunder's terminological defaults.

1 Introduction

Recently, there have been many different web service policy language proposals with varying degrees of expressivity and complexity [21, 6, 1]. One of these languages, WS-Policy became a W3C member submission and is the basis for the WS-Policy working group [3].

In previous work [13] we described a translation of WS-Policy to a standardized logic (OWL-DL). This mapping essentially provided a formal semantics for the framework, and allowed us to use an OWL DL reasoner for policy processing tasks such as determining policy equivalence, incompatibility, containment, incoherence and explanation. In this paper, we provide additional results on the translation by exploring the operation of policy *intersection*. This operation determines whether two policies are compatible and generally involves domain-specific processing. In the official specification of WS-Policy [21], only an approximation algorithm is defined for this operation. Instead, we describe an algorithm based on OWL-DL extended with default rules. Because default logic is computationally more expensive than the logic behind OWL-DL, we do provide clear motivations for our usage of defaults.

To provide reasoning support for OWL defaults we have extended an open-source OWL-DL reasoner (Pellet). Our implementation is based on Baader and Hollunder's

[3] WS-Policy Working Group web site: http://www.w3.org/2002/ws/policy/

terminological default logic [3] (adapted to OWL-DL). To retain decidability, the terminological default logic of Baader and Hollunder restricts the default rules to named individuals only, similar to DL-safe rules. We provide a brief description of our system in Section 6.

2 Preliminaries

In this section we provide brief overview of the WS-Policy framework and Reiter's default logic, which served as the basis of our implementation.

2.1 WS-Policy Framework Overview

The WS-Policy Framework provides a general purpose model and syntax to describe the policies of a Web service. Its scope is limited to allowing endpoints to specify requirements and capabilities needed for establishing a connection. Its initial goal is not to be used as a language for expressing more complex, application-specific policies that take effect after the connection is established. For this purpose, WS-Policy introduces a simple and extensible grammar that consists of *assertions* and *alternatives*.

An assertion is the basic unit of a policy. For example, an assertion could declare that the message should be encrypted. The actual definitions and meaning of the assertions are domain-dependent and not defined in the WS-Policy Framework. An assertion is defined by a unique Qualified Name, and can be a simple string or a complex object with many sub elements and attributes. Note that an assertion can contain a nested policy expression.

A set of assertions is called a policy alternative, and a set of alternatives comprises a policy. For an alternative to be supported by a web service requester, all assertions in that alternative have to be satisfied by that requester. For a policy to be supported by a requester, one or more alternatives need to be supported. Following is a schema outline for the normal form of a policy expression:

```
<wsp:Policy>
   <wsp:ExactlyOne>
      [ <wsp:All> [<Assertion>  </Assertion>]* </wsp:All> ]*
   </wsp:ExactlyOne>
</wsp:Policy>
```

2.2 Default Logic

Reiter's default logic is a nonmonotonic formalism for expressing commonsense rules of reasoning. These rules, called default rules (or simply *defaults*), are of the form:

$$\frac{\alpha \ : \ \beta}{\gamma}$$

where α, β, γ are first-order formulae. We say α is the *prerequisite* of the rule, β is the *justification* and γ the *consequent*. Intuitively, a default rule can be read as: if I can prove the prerequisite from what I believe, and the justification is consistent with what I believe, then add the consequent to my set of beliefs.

Definition 1 *A default theory is a pair $\langle \mathcal{W}, \mathcal{D} \rangle$ where $\mathcal{W}$ is a set of closed first-order formulae (containing the initial world description) and $\mathcal{D}$ is a set of default rules. A default theory is closed if there are no free variables in its default rules.*

Possible sets of conclusions from a default theory are defined in terms of *extensions* of the theory. Extensions are deductively closed sets of formulae that also include the original set of facts from the world description. Extensions are also closed under the application of defaults in $\mathcal{D}$ - we keep applying default rules as long as possible to generate an extension.

Default rules can conflict. A simple example is when two defaults d_1 and d_2 are applicable yet the consequent of d_1 is inconsistent with the consequent of d_2. We then typically end up with two extensions: one where the consequent of d_1 holds, and one where the consequent of d_2 holds.

3 Updated OWL-DL Mapping

In [13] we presented a mapping of WS-Policy to OWL-DL based on the idea that service policy assertions and alternatives were mapped to classes, and web service requesters are mapped to OWL individuals. With this mapping, checking whether a web service requester satisfies a particular policy can then be reduced to simply checking whether the OWL individual representing the requester is a member of the OWL class representing the policy. The mapping was relatively simple since there are only two relevant constructs in a WS-Policy in a normal form (<wsp:exactlyOne>, <wsp:All>). Due to the name of one of the operators (<wsp:exactlyOne>) and the ambiguity in the WS-Policy specifications, we translated it to a logical XOR. Thus the policy $P = ExactlyOne(A,\ B)$ was mapped to the description logic expression: $\mathsf{P} = (\mathsf{A} \sqcup \mathsf{B}) \sqcap \neg(\mathsf{A} \sqcap \mathsf{B})$ (<wsp:All> was mapped to logical conjunction).

However, due to the open world assumption present in OWL-DL, our previous mapping produces non-intuitive results. For example, if a request r comes in such that $r : A$, and the policy P contains only two alternatives, A and B, we will not be able to infer that the request r satisfies P (i.e., r is of type $(\mathsf{A} \sqcup \mathsf{B}) \sqcap \neg(\mathsf{A} \sqcap \mathsf{B})$) unless we explicitly state that $r : \neg B$. To solve this issue, we simplified the mapping to represent <wsp:exactlyOne> as logical disjunction (inclusive OR), and in addition we have made the classes representing the alternatives pair-wise disjoint, so even though a requester supports more than one alternative, he cannot use more than one at a time. This updated translation is more concise than the old one (compare $A \sqcup B$ with $(A \sqcup B) \sqcap \neg(A \sqcap B)$). In this scenario, if a requester comes in that is a member of two alternatives, we will get an inconsistency.

Example 1. Consider the example policy in Figure 1. For each policy assertion, we have a separate OWL class (`RequireDerivedKeys`, `WssUsernameToken10`, `WssUsernameToken11`). Then, each alternative is simply the conjunction of its assertions.

$$\texttt{Alt}_1 \equiv \texttt{RequireDerivedKeys} \sqcap \texttt{WssUsernameToken10}$$
$$\texttt{Alt}_2 \equiv \texttt{RequireDerivedKeys} \sqcap \texttt{WssUsernameToken11}$$

```
(01) <wsp:Policy
        xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
        xmlns:wsp="http://www.w3.org/2006/07/ws-policy" >
(02)   <wsp:ExactlyOne>
(03)     <wsp:All>
(04)       <sp:RequireDerivedKeys />
(05)       <sp:WssUsernameToken10 />
(06)     </wsp:All>
(07)     <wsp:All>
(08)       <sp:RequireDerivedKeys />
(09)       <sp:WssUsernameToken11 />
(10)     </wsp:All>
(17)   </wsp:ExactlyOne>
(18) </wsp:Policy>
```

Fig. 1. Example policy

The policy class P is equivalent to the disjunction of the alternative classes:

$$\mathtt{P} \equiv \mathtt{Alt}_1 \sqcup \mathtt{Alt}_2$$

In addition, we add a disjoint axiom for the alternatives:

$$\mathtt{Alt}_1 \sqsubseteq \neg \mathtt{Alt}_2.$$

4 Policy Processing Services

In our previous work [13] on WS-Policy, we described the services that DL reasoners provide regarding policies: containment, equivalence, incompatibility, incoherence (nothing can satisfy the policy) and policy conformance, among others. Thus, the mapping allows us to use an off-the-shelf OWL reasoner as a policy engine and analysis tool, and an off-the-shelf OWL editor as a policy development and integration environment. OWL editors can also be used to develop domain specific assertion languages (essentially, domain ontologies) with a uniform syntax and well specified semantics.

There is one additional reasoning service that is useful for policies and warrants more discussion. It has been argued (see [4] for example) that explanation is a crucial requirement for a policy language. To address this requirement, we can use recent advances in the field of debugging OWL ontologies [11], esp. in providing explanations for both ontology inconsistencies and arbitrary entailments for OWL-DL.

For example, the *why* query mentioned in [4] can be handled by the explanation for arbitrary entailments. If a user asks why the requester r satisfies the policy P, then the debugging framework is simply asked to provide justification for the type assertion r:P. On the other hand, if a web service request causes an inconsistency (for example because of violating a domain disjointness constraint), then the debugging framework can provide explanation of why the inconsistency occurred. More specifically, if an OWL-DL ontology is inconsistent, [11] provides the minimal set of axioms in the ontology that causes the inconsistency (the set of axioms is called a *justification*).

These techniques are already implemented in Pellet, and there is also a UI for debugging implemented in SWOOP.

5 Policy Intersection

Policy intersection is used when a web service requester and provider both express policies and want to compute the compatible policy alternatives between them. This commutative and associative function takes two policies as input and returns a policy containing the compatible alternatives. As defined in [21], two alternatives are compatible if each assertion in the first alternative is compatible with an assertion in the second, and vice-versa. If two policy alternatives are compatible, their intersection is an alternative containing all of the assertions in both alternatives.

Determining whether two policy alternatives are compatible involves domain-specific processing. In an attempt to automate the operation, one might be tempted to mark the incompatible policy assertions as mutually disjoint classes. Then, to determine whether two policies A and B are compatible we only check whether $\mathsf{A} \sqcap \mathsf{B}$ is satisfiable. However, this will prevent us from having entities support assertions of different types, since it will render the policy ontology inconsistent. Since it is usually the case that entities do support different assertion types (example: an entity can support some specific encoding and some type of reliability, and encoding and reliability are different assertion types), the simple approach of marking incompatible assertions as disjoint classes is incorrect.

To overcome this problem, we introduce an additional property in the policy ontology - `compatibleWith`. Then, for two policy assertion classes A and B, if we want to say that A is not compatible with B, we can simply use $\mathsf{A} \sqsubseteq \neg\exists \texttt{compatibleWith}.\mathsf{B}$.

As stated in [21], assertion authors are encouraged to factor assertions such that two assertions of the same assertion type are typically compatible. We can model this using inheritance hierarchies (with exceptions). For instance, the policy modeler can state that for two classes representing assertions C, D, which she knows are compatible, every pair of classes C_i, D_i that are subclasses of C, D (i.e., $C_i \sqsubseteq C$ and $D_i \sqsubseteq D$) is also compatible by default. This can be expressed with the following default rule:

$$\frac{\mathsf{C}(x) \wedge \mathsf{D}(y) \ : \ \texttt{compatibleWith}(x, y)}{\texttt{compatibleWith}(x, y)}$$

In the cases when two assertions are incompatible (even though they are a inherit from the same type) the policy developer can add a disjoint axiom by hand, overriding the default rule above.

The basic algorithm would be as follows: for two policies A and B and a default theory $KB = \langle \mathcal{W}, \mathcal{D} \rangle$ (where $\mathcal{W}$ is an OWL-DL ontology and $\mathcal{D}$ is a set of defaults), to determine whether they are compatible start with the alternatives of A and try to find one compatible alternative in B, and vice-versa. If for at least one alternative in one policy, we succeed in finding compatible alternatives in the other policy, we conclude that the policies can intersect. The intersection of the policies is the policy containing the mutually compatible set of alternatives. To determine whether two alternatives are compatible, we try to match their assertions. For each assertion $\texttt{Assert}_\mathsf{a} \in \mathsf{A}$, we try to find

an assertion $\texttt{Assert}_\texttt{b} \in \texttt{B}$ s.t. $KB \models \texttt{compatibleWith}(\texttt{Assert}_\texttt{a}, \texttt{Assert}_\texttt{b})$. If the assertion has a nested policy, then we try to match it with a nested policy from the other alternative, by asking recursively whether they are compatible.

6 OWL Defaults

Both of the default logic scenarios described above could be plausibly met with Reiter's default logic, which is one of the most studied non-monotonic logics. Reiter's default logic, while very expressive, is, like many non-monotonic formalisms, known to be computationally difficult even in the propositional case. In [3], Baader and Hollunder showed that even a restricted form of defaults coupled with a description logic that contains a smaller set of constructors than OWL-DL was undecidable. They also showed that if one restricted the defaults to apply only to named individuals (or, equivalently, restricted the logic to closed defaults), then a robust decidability ensued.

We have implemented a prototype of the terminological defaults of Baader and Hollunder that is based on recent advances in description logic reasoning: tableaux tracing for the description logic $\mathcal{SHOIN}$ and incremental reasoning support. The implementation is provided as an extension to Pellet and it provides *realization* of individuals in terminological default theories. We have also provided a UI for defaults by extending the open source OWL Ontology editor SWOOP. More specifically, we added support for default rules editing and updating the current ontology with the set of inferred facts from the defaults. We refer the reader to [12] for more details.

7 Related Work

There have been a number of proposals for ontology-based web policy systems [16, 10, 18, 8] - because of lack of space, we will only briefly cover Rei and KaOS.

Rei [10] is a policy specification language based on a combination of OWL-Lite, logic-like variables and rules. It allows users to develop declarative policies over domain specific ontologies in RDF and OWL. Rei allows policies to be specified as constraints over allowable and obligated actions on resources in the environment. A distinguishing feature of Rei is that it includes specifications for speech acts for remote policy management and policy analysis specifications like what-if analysis and use-case management. Our goal is to encode WS-Policy in a not very expressive logic formalism (so as to be able to perform policy analysis), and our opinion is that we do not need a language as expressive as Rei for WS-Policy.

KaOS Policy and Domain Services [18] use ontology concepts encoded in OWL to build policies. These policies constrain allowable actions performed by actors which might be clients or agents. The KAoS Policy Service distinguishes between authorizations and obligations. The applicability of the policy is defined by a class of situations which definition can contain components specifying required history, state and currently undertaken action. Even though we use the same representation language as KaOS (OWL-DL), our reasoning support is provided by tableaux-based description logic reasoners which are sound and complete for OWL-DL. In addition, by using Pellet we were able to leverage its ontology debugging support.

In addition, there are a number of proposals [20, 14] of policy/authorization languages based on logic programs extended with default rules - the difference with our approach is that we use description logics as the underlying logic formalism.

8 Conclusions and Future Work

While most policy language proposals are based on logic programs, in this paper we explored the alternative of using OWL-DL as a language for expressing web service policies. We argued that the policy services that DL reasoners provide out of the box, the advances in explanation mechanisms for DL, and the ability to closely integrate OWL-DL with default logic make an OWL-based policy framework worth exploring. Also, OWL-DL is a W3C standard, a language with clear syntax and semantics that is ubiquitous in the Semantic Web. As a consequence, the number of reasoners and OWL-DL editors has been growing steadily. A policy language based on OWL-DL should be able to capitalize on the popularity of OWL-DL.

Despite the advantages mentioned above, policies, being associated with rules first and foremost, seem to demand greater expressivity than OWL-DL (as argued in [9], for example) in the form of monotonic rules. However, because of the recent advances in hybrid (description logic + logic programs) knowledge bases, and successful implementations ([15]) we believe that OWL-DL combined with rules is reaching a maturity level where it will be a suitable alternative for a policy framework.

During the past couple of years, there has been great advances [7, 5, 19, 17] in the area of automated trust negotiation (ATN) between policy entities. ATN deals with the problem of exchanging of sensitive credentials between strangers in order to establish trust. We plan to investigate how we can integrate our OWL-based system with such mechanisms.

Finally, it is unfortunate that we cannot provide clear semantics for policy intersection because its dependence on domain-specific reasoning. The WS-Policy framework requires each domain to specify its own policy assertions, but there is no generic, domain-independent language for expressing these assertions. As a result, every domain has its own language (with unclear semantics) that makes it hard to reason and analyze the assertions. We plan to investigate how we could couple OWL with concrete domains (e.g. XPath) so as to be able to express and give semantics to some of these domains. A promising step toward a domain-independent policy assertions language is [2]; we plan to investigate the idea further.

References

1. A. H. Anderson. An introduction to the web services policy language. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, 2004.
2. Anne Anderson. WS-PolicyConstraints: A domain-independent web services policy assertion language, November 2005. Available at http://research.sun.com/projects/xacml/IntroToWSPolicyConstraints.pdf.
3. Franz Baader and Bernhard Hollunder. Embedding Defaults Into Terminological Knowledge Representation Formalisms. *J. Autom. Reasoning*, 14(1):149–180, 1995.

4. P.A. Bonatti, G. Antoniou, M. Baldoni, C. Baroglio, C. Duma, N. Fuchs, A. Martelli, W. Nejdl, D. Olmedilla, J. Peer, V. Patti, and N. Shamheri. The rewerse view on policies.
5. Piero A. Bonatti and Pierangela Samarati. A uniform framework for regulating service access and information release on the web. *J. Comput. Secur.*, 10(3):241–271, 2002.
6. Jacques Durand et al. Wsdl annotation proposal. http://lists.oasis-open.org/archives/wsrm/200403/msg00082.html.
7. 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 *European Semantic Web Symposium*, May 2004.
8. Stephan Grimm, Steffen Lamparter, Andreas Abecker, Sudhir Agarwal, and Andreas Eberhart. Ontology based specification of web service policies. In *Semantic Web Services and Dynamic Networks Workshop*, 2004.
9. R. Montanari J. Bradshaw, L. Kagal and A. Toninelli. Rule-based and ontology-based policies: Toward a hybrid approach to control agents in pervasive environments. In *Proceedings of the ISWC2005 Semantic Web and Policy Workshop*, 2005.
10. L. et al Kagal. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.
11. Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics - Special Issue of the Semantic Web Track of WWW2005*, 3(4), 2005.
12. Vladimir Kolovski, Bijan Parsia, and Yarden Katz. Implementing owl defaults. Technical report, University of Maryland - College Park, 2006. http://www.mindswap.org/ kolovski/defaults.pdf.
13. Vladimir Kolovski, Bijan Parsia, Yarden Katz, and Jim Hendler. Representing web service policies in owl-dl. In *International Semantic Web Conference (ISWC)*, 2005.
14. Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003.
15. Boris Motik, Ulrike Sattler, and Rudi Studer. Query answering for owl-dl with rules. In *Proc. of ISWC 2004*, pages 549–563.
16. W. Nejdl, D. Olmedilla, M. Winslett, and C. Zhang. Ontology-based policy specification and management. In *2nd European Semantic Web Conference (ESWC)*, May 2005.
17. K. Seamons, M. Winslett, and T. Yu. Limiting the disclosure of access control policies during automated trust negotiation, 2001.
18. A. Uszokand and J. Bradshaw. Kaos policies for web services. In *W3C Workshop on Constraints and Capabilities for Web Servies*, October 2004.
19. W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. Technical Report TR-2000-05, 24 2000.
20. T. Y. C. Woo and S. S Lam. Authorization in distributed systems : A formal approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 33–51, 1992.
21. WS-Policy. Web services policy framework (ws-policy). http://www-106.ibm.com/developerworks/library/specification/ws-polfram/.