Representing Web Service Policies in OWL-DL

Vladimir Kolovski[1], Bijan Parsia[2], Yarden Katz[2], and James Hendler[2]

[1] Maryland Information and Network Dynamics Laboratory Lab,
University of Maryland , College Park , MD 20740
[2] Dept. of Computer Science, University of Maryland, College Park, MD 20742
{kolovski, hendler}@cs.umd.edu, bparsia@isr.umd.edu, yarden@umd.edu

Abstract. Recently, there have been a number of proposals for languages for expressing web service constraints and capabilities, with WS-Policy and WSPL leading the way. The proposed languages, although relatively inexpressive, suffer from a lack of formal semantics. In this paper, we provide a mapping of WS-Policy to the description logic fragment species of the Web Ontology Language (OWL-DL), and describe how standard OWL-DL reasoners can be used to check policy conformance and perform an array of policy analysis tasks. OWL-DL is much more expressive than WS-Policy and thus provides a framework for exploring richer policy languages.

1   Introduction

To provide for a robust development and operational environment, web services are described using machine-readable metadata. This metadata serves several purposes, one of them being describing the capabilities and requirements of a service – often called the service policy. Recently, there have been many different web service policy language proposals, all of them describing languages with varying degrees of expressivity and complexity [17, 4, 1]. However, with most current proposals it is difficult to determine their expressivity and computational properties as most lack formal semantics. One characteristic of the proposed languages is that they involve policy assertions and combinations of assertions. For example, a policy might assert that a particular service requires some form of reliable messaging or security, or it may require both reliable messaging and security. Several industrial proposals (e.g., WS-Policy [17] and Features and Properties [4]) appear to restrict them to a kind of propositional logic with policy assertions being atomic propositions and the combinations being conjunction and disjunction. By mapping the policy language constructs into a logic (e.g., some variant of first order logic) we can acquire a clear semantics for the languages, as well as a good sense of the computational aspects.

Additionally, if we can map the policy languages into a standardized logic, we can benefit from the tools and general expertise one expects to come with a reasonably popular standard. By mapping two policy languages into the same background formalism, we will be able to provide some measure of interoperability between policies written in distinct languages. If we are smart in our mapping, we should also be able use pre-existing reasoners for the standardized logic to do policy processing.

Our language of choice is the Web Ontology Language, OWL [2], and the Resource Description Framework, RDF [11]. Both RDF and OWL are strict subsets of first order logic, with the subspecies OWL-DL being a very expressive yet decidable subset. OWL-DL builds on the rich tradition of description logics where the tradeoff between computational complexity and logical expressivity has been precisely and extensively mapped out and practical, reasonably scalable reasoning algorithms and systems have been developed.

In this paper, we have translated one of the policy languages, WS-Policy, to OWL-DL. WS-Policy is being developed by IBM, Microsoft, BEA, and other major web services vendors and is generally considered to be the policy language with the most momentum. Our approach maps policies to OWL-DL classes. With this, we are able to use our OWL-DL reasoner, Pellet [15] as a policy processor with analysis services that go far beyond what is usually offered. We also tackle another policy-related proposal, Features and Properties, and describe how its boolean predicates can also be translated to OWL-DL. In our evaluation section, we demonstrate how generic OWL-DL reasoners can easily handle processing moderately sized policies.

2   WS-Policy Overview

WS-Policy provides a general purpose model and syntax to describe the policies of a Web service. It specifies a base set of constructs that can be used and extended by other Web service specifications to describe a broad range of service requirements and capabilities. WS-Policy's scope is limited to allowing endpoints to specify requirements and capabilities needed for establishing a connection. Its goal is not 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 for expressing policies and a processing model to interpret them. A policy, as defined in the specification is composed from a combination of assertions and alternatives.

An assertion is the basic, atomic 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 WS-Policy. 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. A set of assertions can be termed an alternative.

A policy is built up using assertions and nested combinations of the operators `wsp:All`, `wsp:ExactlyOne`, and the attribute `wsp:Optional`. This policy syntax is used to describe acceptable combinations of assertions for a given Web service invocation.

2.1   Mapping WS-Policy Operators to OWL

In this section, we describe our mapping of the WS-Policy constructs from a normal form policy expression into OWL expressions. A policy in a normal form

is a straightforward XML Infoset representation, enumerating each of its alternatives that in turn enumerate each of its assertions. 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>
```

Listing 1. Normal form of a policy expression

Policy expressions can also be represented in more compact forms, using additional operators such as `wsp:Optional`, however as shown in [17] the policy expressions can all be expanded to normal form. Therefore we only provide a mapping of the constructs used in a normal form policy expression: `wsp:ExactlyOne` and `wsp:All`.

First, we map policy assertions directly into OWL-DL classes (which correspond to atomic propositions). Though WS-Policy assertions often have some discernible substructure, it is not key to their logical status in WS-Policy. Or rather, that substructure is idiosyncratic to the assertion set, rather than being a feature of the background formalism. So a general WS-Policy engine must be adapted to deal with their structure, if it is to do so. The WS-Policy specification asserts: "Assertions indicate domain-specific (e.g., security, transactions) semantics and are expected to be defined in separate, domain-specific specifications."

It seems unfortunate that each domain-specific specification comes with its own domain specific syntax. If we are to capture the semantics of each assertion language, we must separately map each assertion language into OWL. We do provide a general strategy for mapping WS-Policy assertions in the next section.

Mapping `wsp:All` to an OWL construct is straightforward because `wsp:All` means that all of the policy assertions enclosed by this operator have to be satisfied in order for communication to be initiated between the endpoints. Thus, it is a logical conjunction and can be represented as an OWL intersection. Each of the members of the intersection is a policy assertion, and the resulting class expression is a custom-made policy class that expresses the same semantics as the WS-Policy one.

Handling `wsp:ExactlyOne` might be trickier, depending on the interpretation of the operator. There are two possible interpretations:

– `wsp:ExactlyOne` means that a policy is supported by a requester if and only if the requester supports at least one of the alternatives in the policy. In the previous version of WS-Policy there was a `wsp:OneOrMore` construct capturing this meaning. In such case, the `wsp:ExactlyOne` is an inclusive OR , and can be mapped using `owl:unionOf`.
– `wsp:ExactlyOne` means that only one, not more, of the alternatives should be supported in order for the requester to support the policy. This is supported

by [17], where it is stated that although policy alternatives are meant to be mutually exclusive, it cannot be decided in general whether or not more than one alternative can be supported at the same time. Our translation covers this more complicated case.

`Wsp:ExactlyOne` can be translated to OWL in the following way: for n different policy assertions, expressed as OWL classes themselves, `wsp:ExactlyOne` is the class expression consisting of the members of each separate policy class that do **not** also belong to another policy class. In OWL terms, it is the union of all of the classes with the complement of their pair-wise intersections. Because of the pair-wise intersections there is a quadratic increase in the size of the OWL construct that is used as a mapping for `wsp:ExactlyOne`.

Table 1. Mapping of WS-Policy Constructs to OWL

WS-Policy Construct	OWL Expression
Wsp:All (policies A and B)	owl:intersectionOf(A B)
Wsp:ExactlyOne (policies A and B)	intersectionOf(complementOf(intersectionOf(A B)) unionOf(A B))

To more compactly express complex policies, WS-Policy allows nesting of operators. To convert a policy from a compact to a normal form, the properties of `wsp:ExactlyOne` and `wsp:All` can be used. If we are to show that our translation correctly captures the meaning of `wsp:ExactlyOne` and `wsp:All`, we need to prove that the mappings from Table 1. have the same properties as the WS-Policy operators. `wsp:ExactlyOne` and `wsp:All` have the following properties: *commutativity*, *associativity*, *idempotency* and *distributivity*. It can be easily shown that our mappings, which are essentially a logical conjunction and explicit disjunction, also satisfy these properties.

2.2 Mapping Policy Assertions to OWL

In this section we provide a mapping for the building blocks of a policy expression, the policy assertions. Our proposal for mapping assertions is first to create a base class for every general policy assertion, e.g., *wsp:Language*, *wsp:TextEncoding,wsse:BinarySecurityToken* would be mapped to OWL classes `BaseLanguage`, `BaseTextEncoding`, `BaseBinarySecurityToken`. A WS-Policy assertion in normal form consists of attributes and elements. We describe how these are handled separately:

– for attributes, we create a datatype property representing that attribute and use the `owl:hasValue` restriction on that property to create a new class corresponding to the assertion.

– for elements, we create separate classes for all of the elements contained in the policy assertion. Then, the specific assertion class is created by placing `owl:allValuesFrom` restrictions on properties that relate the base assertion class with the generated classes for the elements.

In order to illustrate the approach, consider the following assertion:

```
<wsse:Integrity wsp:Preference="100">
      <wsse:Algorithm  Type="wsse:AlgCanonicalization"
              URI="http://www.w3.org/Signature/xml-exc-c14n"/>
</wsse:Integrity>
```

The translation of this assertion would produce two classes, $Integrity_1$ and $Algorithm_1$, shown below:

Table 2. Translation of Example Policy Assertion

$Integrity_1 \equiv ((\forall$ hasAlgorithm. $Algorithm_1) \cap (=1$ hasAlgorithm.$Algorithm_1) \cap$ ($\exists$hasPreference.100) $\cap$ BaseIntegrity)
$Algorithm_1 \equiv ((=1$ hasType.{"wsse:AlgCanonicalization"}) $\cap$ (=1 hasURI.{"http://www.w3.org/Signature/xml-exc-c14n"}) $\cap$ BaseAlgorithm)

Having this information in hand, we developed an XSL script[1] that takes a WS-Policy expression in normal form and produces valid OWL-DL. For demonstrative purposes, we translated a subset of WS-PolicyAssertions using the approach specified above.

2.3   WS-Policy Merge and Intersection

In this section, we discuss the possibility of expressing `Merge` and `Intersection`.

`Merge` is the process of combining sub-policies together to form a single policy. This operation is needed because a policy might be specified in a distributed way, having its fragments defined in separate files. It is necessary to combine all these policy fragments together to form a single merged policy which could be processed further.

`Merge` works on policies already converted to normal form. The merged policy is a Cartesian product of the alternatives in the first policy and the alternatives in the second policy. There is a straightforward way of doing the `Merge` operation in OWL-DL. First, we translate each of the input policies into OWL-DL as described above. Then, the merged policy is simply the *intersection* of the input policies. Thus, `Merge` also maps cleanly onto OWL-DL. An outline of the proof is shown in Appendix 1.

---

[1] http://www.mindswap.org/2005/services-policies/wsp2owl.xsl

The goal of WS-Policy is to allow endpoints to specify requirements for starting a web service interaction. To achieve this goal, the `Intersection` operation compares two Web services policies for common alternatives. Interaction is possible only when both of the endpoints agree on at least one policy alternative.

Like in `Merge`, the process of coming up with an intersection is carried out in a cross product fashion, comparing each alternative from the first policy with every alterantive from the other one. However, in the case of `Intersection`, if the two alternatives that are being combined do not agree on the same vocabulary, then they combined alternative is not added to the new policy. A vocabulary of an alternative is simply defined as the set of QNames of the assertions in that alternative.

`Intersection` cannot be mapped into a single OWL construct, however using our OWL mappings of the policy assertions it is not difficult to rule out the incompatible alternatives. If the policy assertions are mapped to classes, then to check whether two alternatives are equal, we need to see whether the assertions in the two alternatives are derived from the same base clases. Specifically, evey assertion in the first alternative needs to be derived from the same base class with some assertions from the second alternative, and vice-versa, for the alternatives to be compatible.

3   Policy Processing

One of our arguments for expressing policies using OWL was the ability to reason about policy containment - whether the requirements for supporting one policy are a subset of the requirements for another. That would allow us to be more flexible in determining whether a particular requestor supports a policy, in the cases where the requestor supports a superset of the requirements established by the policy.

In general, we get the following inferences out of the box:

1. policy inclusion (if x meets policy A then it also meets policy B; a.k.a., A rdfs:subClassOf B);
2. policy equivalence (A owl:equivalentTo B);
3. policy incompatibility (if x meets policy A then it cannot meet policy B; a.k.a, A owl:disjointWith B);
4. policy incoherence (nothing can meet policy A; a.k.a., A is unsatisfiable)
5. policy conformance (x meets policy A; a.k.a., x rdf:type A)

One futher reasoning service supported by Pellet, and integrated with Swoop [10], is explanations for inconsistencies [14], which can be used to help debug policy incompatibility, incoherence, and the like. As we add further explanation capability to our systems, this debugging power will grow.

Thus we see that with a fairly simple mapping, we can 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. We can also experiment with extensions to WS-Policy, by using more expressive constructs from OWL at the policy language, as well as the assertion language, level. We can experiment with extensions before having to write a yet another processor for them. Of course, if it turns out that we really want to restrict ourselves to a very inexpressive subset, then we may still want to build specific reasoners and processors that are tuned for that sublanguage. But there again, our tools can help us. Pellet does expressivity analysis of ontologies, so can help determine what logic we are really using and the price of extensions.

Furthermore, ontology development techniques can be useful for policy development as well. Most human generate ontology develop iteratively, with specializations added to the class tree over time. Similarly, we can build up our policies from more general ones. A general policy could be very restrictive, setting tough guidelines for all of a companies policies.

If we have a similar style mapping for another policy language, we will be able to do policy analysis and integration across policy languages. We have taken the first steps in this direction with providing a translation of the Features and Properties compositors.

However, some care must be taken given the open world semantics of OWL. For example, an OWL reasoner does not assume that because it cannot prove that x conforms to policy A, that x does not conform to policy A. It is unclear what the WS-Policy authors intend, though a closed world assumption is not unlikely. However, even if there is a closed world assumption on WS-Policies, we can handle at least some of those cases by adding explicit disjoint statements at translation time. In the following section, we delve into the issue of open vs. closed world semantics.

4   The Semantics of Policies

Many of the current web policy languages do not have a formal semantics, leaving the meaning of certain language constructs unclear. The WS-Policy language provides for a good example. In our translation of WS-Policy documents into OWL, we assume, of course, OWL's *open world* semantics. Under this assumption, the failure to *prove* an assertion leaves us with no conclusion about the assertion's truth or falsity. That is, in light of incomplete knowledge, some statements about policies simply remain *unknown*. By contrast, in the *closed world* assumption the failure to prove an assertion ϕ leads to the conclusion that $\neg\phi$ is the case.

While the open world assumption was made for OWL ontologies, and the choice can certainly be justified[2], it seems that WS-Policy operates under closed world assumption. The Intersection operation in WS-Policy, which is used to

---

[2] The idea behind the open world assumption is that if an assertion is not made in your knowledge base, you should not infer that it is false, because the assertion might still be made in some other ontology on the Web. Thus, the open world assumption is more "web like."

determine the policy on which both endpoints agree, does not include those alternatives that have no matching assertions. In other words, if the provider has an assertions indicating support of a specific functionality, and the requester is missing that assertions in his policy, then they are not compatible with each other.

Let us contrast open and closed world assumptions. Suppose that a policy is devised to express the constraints for gaining web access. A person fulfills the requirement for web access if he or she are either a registered user or a guest user. The policy can be expressed in OWL as follows:

Table 3. Web Access Policy class definition

`WebAccessPolicy ⊑ Policy`
`WebAccessPolicy ≡ (RegisteredUser ⊔ GuestUser)` `                   ⊓ ¬(RegisteredUser ⊓ GuestUser)`

It is easy to see what kind of individual will fail to belong to `WebAccessPolicy`. Since our definition of this class corresponds to the WS-Policy `ExactlyOne` operator, its members must be instances of either `GuestUser` or `RegisteredUser`, but *not both*. However, as a consequence of OWL's open world semantics, it is not enough for an individual i to simply belong to `GuestUser` or `RegisteredUser` (and not to both) for i to satisfy the second conjunct of the `WebAccessPolicy` class definition. Rather, in the case that i : `RegisteredUser`, it must *also* be provable that i : ¬`GuestUser`, and vice versa.

Contrast this with a translation of the above policy into a closed world language, such as Prolog, given below.[3]

```
not(X, Y) :- \+ X ; \+ Y.
policy(X) :- webAccessPolicy(X).
webAccessPolicy(X) :- (guestUser(X) ; registeredUser(X)),
                      not(guestUser(X), registeredUser(X)).
```

Unlike in the OWL case, the knowledge base consisting of the assertions {`guestUser(bob)`, `regularUser(john)`} will be sufficient to conclude that both `webPolicyAccess(bob)` and `webPolicyAccess(john)`. Since it is not provable that `guestUser(john)` and `regularUser(bob)` (which would disqualify both from our policy), we simply assume that they are *not* such. This constitutes the closed world assumption. The behavior of this example might be more reasonable than its OWL counterpart, depending on the specific policy and associated knowledge base.

4.1   Bridging Open and Closed Assumptions

It would be desirable to have a way to 'turn on' the closed world effect as needed in our own policies, depending on the specific application, without committing to

---

[3] Note that in Prolog, `;` stands for disjunction, `\+` for negation, and `,` for conjunction.

it across the board (which Prolog does.) Furthermore, there are cases where the open world effect can force us to model our policies unnaturally. These counter intuitive results of open world semantics for policy developers can be handled with a closed world mechanism. Consider the following example:

A research lab in College Park uses OWL to specify its policies. In the research lab, there are two types of employees: senior employees and non-senior (regular) employees, both subclassed from the Employee class. A senior employee is modelled as a kind of regular employee. Every employee has been specified a set of rights for use of devices in the lab. While senior employees are able to delegate rights to use certain devices, regular employees cannot. For example, a senior employee might delegate the right to use the conference room printer to a regular employee. Now consider two individuals, Evren who is a regular employee, and Ryu who is a senior one. If we specify that Ryu delegates the right to Evren to use the conference room printer, there is no harm done since Ryu is a Senior Employee. However, if we specify that Evren delegates the right to use the, say, conference printer to Ryu, we would expect a contradiction since regular employees are not able to delegate rights. However, because of the open world assumption, the fact that Evren is delegating rights, and isn't defined to be a *non-senior* employee, allows the OWL reasoner to *infer* that Evren is a *senior* employee. This is the opposite of what the policy writer had in mind. The undesirable consequence is illustrated below.

Table 4. Undesirable consequence of OWA for policy modelling

Policy definition
`DelegationConfPrinter ⊑ Delegation`
`DelegationConfPrinter ≡ ∃delegationGiver.SeniorEmployee`   `⊓ ∃delegationType.RightToUseDevices`
`RightToUseConfPrinter ⊑ RightToUseDevices`

Knowledge base
`evren : RegularEmployee`
`ryu : SeniorEmployee`
`badOWA : DelegationConfPrinter`
`delegationGiver(badOWA, evren)`
`delegationReceiver(badOWA, ryu)`

The above policy, paired with the shown knowledge base, will yield the inference that Evren is of type `SeniorEmployee`. A possible fix for the problem would be to make `SeniorEmployee` and `RegularEmployee` disjoint, though this would break the perfectly correct modeller's intuition that: (1) the two kinds of employees should share the superclass `Employee`, and (2) a senior employee is a regular employee *plus some privileges*, captured by making the former of a subclass of the latter.

A better solution that would allow us to keep the current class hierarchy intact is to use a *default rule*. Essentially, we'd like to enforce that any individual

who is not *already known* to be a senior employee does not have delegation rights. The individual `ryu` in the above knowledge base clearly does not fall in this category, being asserted to be a senior employee. The modal operators **K** and **A**, discussed as an extension for the description logic $\mathcal{ALC}$ in [3], allow us to express this constraint as a sentence in the knowledge base.[4] The **K** operator can be read as what is "known" to be the case, while the **A** operator can be read as a "default assumption":

$$\textbf{K}\texttt{Employee} \sqcap \neg\textbf{A}\texttt{SeniorEmployee} \sqsubseteq \neg\texttt{Delegation}$$

Which can be read as "If one is known to be an employee, but is not already a senior employee, then one does not have the ability to delegate rights." Note that the use of **K** and **A** here introduces a closed world assumption with respect to the rule. If it is only asserted that `evren` is an `Employee` for example, then by default he is *not* a `SeniorEmployee`, preventing us from errorneously inferring that he is one.

5   WSDL

In addition to WS-Policy, we explored another proposal, Features and Properties [4] that has also been put forth as a candidate for describing web service policies. Integrating WSDL 2.0 with Features and Properties produced a framework that allows users to specify web service capabilities and requirements in the service description, with expressiveness similar to WS-Policy. The framework in question is based on three concepts, `Features`, `Properties` and `Compositors`. Simply put, a `Feature` represents a piece of functionality, identified by a URI. An example of a `Feature` would be encryption. `Properties` are the parameters of a `Feature`, also identifiable by a URI. For an encryption `Feature`, `Property` might be the algorithm used, part of message encrypted, etc. `Compositors` are used for combining multiple `Features` and `Properties`. There are four `Compositors` defined in the proposal:

1. *all*: this compositor specifies that a service invocation MUST comply with all the children elements
2. *choice*: specifies that a service invocation MUST comply with exactly one of the possibly many children elements
3. *one-or-more*: specifies that a service invocation MUST comply with at least one of the possibly many children elements
4. *zero-or-more*: specifies that a service invocation MAY comply with one or more of the children elements

The compositors in WSDL do provide more options than WS-Policy, however they too can be mapped to OWL, as shown in the following table:

---

[4] Two similar solutions are possible using these operators. The **K** operator can also be used to formulate an equivalent constraint in the form of a query. Another alternative is to use a *default rule*; we refer the reader to [3] for the details of both alternatives.

Table 5. Mapping of Features and Properties Compositors to OWL

WSDL Compositor	OWL Expression
all (policies A and B)	owl:intersectionOf(A B)

choice (policies A and B)	intersectionOf(
	    complementOf(intersectionOf(A B))
	    unionOf(A B))

one-or-more	owl:unionOf
zero-or-more	means *optional*, not mapped to any OWL expressions

6   Evaluation

One of the benefits of expressing policies in OWL is the possibility of using an off-the-shelf OWL reasoner as a policy engine and analysis tool. In this section, we show that currently available DL reasoners can easily process moderately-sized policies. For the purpose of our evaluation, we have selected three reasoners, Pellet [15], FaCT [8] and Racer [6]. We also created a random policy generator, a script that creates policy assertions and specific policy classes and individuals that have the structure of a WS-Policy in OWL form. The translation of the synthetic WS-Policies to OWL-DL took neglibile time (<0.3 sec) using the XSLT mentioned above.

Table 6 summarizes the results of classifying these policy ontologies with the reasoners.

Table 6. Classifying Policy ontologies using off-the-shelf DL reasoners

Policy Size (assertions, policies)	Pellet (sec.)	Racer (sec.)	FaCT (sec.)
(100,10)	0.81	0.91	1.03
(100,20)	1.00	1.32	1.20
(200,20)	1.53	1.45	1.55
(200,40)	2.17	1.75	2.30
(1000,100)	15.54	22.32	16.22

Every policy in the test samples contains multiple (at most 8) alternatives, and each alternative has multiple assertions, thus the samples represent reasonably complex policies. Even in a case with 200 assertions and 40 policies, all of the reasoners performed well (around 2 sec.) The evaluation supports our claim that OWL Reasoners are more than ready to be used as policy processing tools.

7   Related Work

To the best of our knowledge, there have been no previous attempts of expressing WS-Policy in OWL. There have been, however, numerous proposals for web service policy languages based on XML or OWL. The main difference between our work and related policy languages is the level of expressivity - WS-Policy is focused on those aspects of a service required to establish a connection between endpoints and it does not require a great deal of expresivitiy. Most of the languages discussed in this section on the other hand, have a bigger scope of being able to specify high-level, application-specific, heterogeneous policies.

First, we look at XML-based policy languages. The Web Services Policy Language [1], developed at Sun Microsystems, is suitable for specifying a wide range of policies, including authorization, quality-of-service, quality-of protection, reliable messaging, privacy, and application-specific service options. WSPL is of particular interest in several respects. It supports merging two policies, resulting in a single policy that satisfies the requirements of both, assuming such a policy exists. Policies can be based on comparisons other than equality, allowing policies to depend on fine-grained attributes such as time of day, cost, or network subnet address. By using standard data types and functions for expressing policy parameters, a standard policy engine can support any policy. The syntax is a strict subset of the OASIS eXtensible Access Control Markup Language (XACML [5]) Standard.

In essence, a WSPL policy is a sequence of one or more rules, where each rule represents an acceptable alternative. A rule contains a number of predicates, which correspond to policy assertions in WS-Policy. All of the predicates need to be satisfied for the rule to be satisfied. However, only one of the rules can be satisfied for the policy to be satisfied. A WSPL Policy on an operator level is in Disjunctive Normal Form, thus expressible in OWL-DL.

WSPL defines a standard language for use in specifying predicates that constrain domain-specified vocabulary items. Each predicate places a constraint on the value of an Attribute. Possible constraints are: equals, greater than, greater than or equal to, less than, less than or equal to, setequals and subset. Unfortunately , the OWL datatyping formalism is not expressive enough to generally represent datatype predicates such as the ones mentioned. There has been a recent proposal of an extension to OWL-DL, caled OWL-E [13] which adds datatype group-based class constructors to allow the use of datatype expressions in class restrictions. OWL-E is interesting because it adds much more datatype expressiveness and it is still decidable.

The Platform for Privacy Preferences Project (P3P [12]) enables Web sites to express their privacy practices in a standard XML-based format that can be retrieved automatically and interpreted easily by user agents. Similar to what we hav edone with WS-Policy, there has been a number of attempts to use an RDF or OWL schema to describe the semantics of P3P. According to [18], there exists a data-centric relational semantics for P3P in which a P3P policy is modeled as a relational database, that further allows to express P3P using RDF. However, it is important to take note that modal logical statements can be made about data

types in the P3P schema. This issue is investigated in detail by Hogben [7], which provides a complete OWL schema that captures the semantics of P3P. Having P3P modelled in OWL allows the authors to perform syntactic and semantic validation on the policies.

Moving to OWL-based systems, Rei [9] 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, DAML+OIL 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.

KaOS Policy and Domain Services [16] 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.

8   Conclusion and Future Work

In this section we provide a summary of our contributions and possible future directions:

1. By providing a mapping for the formalism of WS-Policy we have shown that it is an expressive subset of OWL-DL
2. Currently available OWL reasoners perform reasonably well as policy processors, without any modification, and we have preliminary empirical results to show for it.
3. OWL-DL provides an interesting framework for exploring richer policy languages with minimal implementation cost. An interesting direction would be integration of our policy mapping with OWL-S profiles, which seems like a natural next step.
4. In the cases when OWL is not suitable, we have clear extensions we can add to address these issues. We covered the K operator and OWL with datatype predicates in this paper.
5. Finally, since other policy languages (WSDL, WSPL) also seem to be subsets of (a slightly extended) OWL, OWL-DL seems to be the right language for specifying policies in general.

Acknowledgements

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. M. Dean and G. Schreiber. Owl web ontology language reference w3c recommendation., feb 2004.

3. F. M. Donini, D. Nardi, and R. Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic*, pages 1529–3785, 2001.

4. J. D. et al. Wsdl annotation proposal.
http://lists.oasis-open.org/archives/wsrm/200403/msg00082.html.

5. S. Godik and T. Moses. Oasis extensible access control markup language (xacml) version 1.1. oasis committee specification, July 2003.

6. V. Haarslev and R. Mller. Racer: A core inference engine for the semantic web. *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools*, 2003.

7. G. Hogben. Describing the p3p base data schema using owl. In *A WWW2005 Workshop on Policy Management for the Web*, 2005.

8. I. Horrocks. The fact system. http://www.cs.man.ac.uk/ horrocks/FaCT/.

9. L. e. a. Kagal. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.

10. A. Kalyanpur, B. Parsia, and J. Hendler. A tool for working with web ontologies. In *In Proceedings of the International Journal on Semantic Web and Information Systems, Vol. 1, No. 1, Jan - March*, 2005.

11. O. Lassila and R. Swick. Resource description framework (rdf) model and syntax specification, February 1999.

12. P3P. Platform for Privacy Preferences Project. http://www.w3.org/P3P/.

13. J. Z. Pan and I. Horrocks. Owl-e: Extending owl with expressive datatype expressions. Technical report, Victoria University of Manchester, 2004.

14. B. Parsia, E. Sirin, and A. Kalyanpur. Debugging owl ontologies. In *The 14th International World Wide Web Conference (WWW2005)*, Chiba, Japan, May 2005.

15. Pellet. Pellet - owl dl reasoner, 2003. http://www.mindswap.org/2003/pellet.

16. A. Uszokand and J. Bradshaw. Kaos policies for web services. In *W3C Workshop on Constraints and Capabilities for Web Servies*, October 2004.

17. WS-Policy. Web services policy framework (ws-policy).
http://www-106.ibm.com/developerworks/library/specification/ws-polfram/.

18. T. Yu, N. Li, and A. Anton. A formal semantics for p3p. In *ACM Workshop on Secure Web Services*, October 2004.

9   Appendix

Theorem 1. `Merge` *between two policies, as defined in WS-Policy, is equivalent to the conjunction of the OWL translations of the two policies.*

Proof: Consider two policies, P1 and P2 in normal form:
$P_1 = \text{ExactlyOne}(A_1, A_2, A_3, ... A_n)$
$P_2 = \text{ExactlyOne}(B_1, B_2, B_3, ... B_n)$

Then, their translations to OWL would have the following form:
$O_1 = (A_1 \cup A_2 \cup A_3 \cup ... \cup A_n) \bigcap \neg((A_1 \cap A_2) \cup (A_1 \cap A_3) \cup ... \cup (A_{n-1} \cap A_n))$
$O_2 = (B_1 \cup B_2 \cup B_3 \cup ... \cup B_n) \bigcap \neg((B_1 \cap B_2) \cup (B_1 \cap B_3) \cup ... \cup (B_{n-1} \cap B_n))$

A merged policy can be mapped to the following OWL expression,
$P1$ merge $P_2 = ((A_1 \cap B_1) \cup (A_1 \cap B_2) \cup ... \cup (A_n \cap B_m)) \bigcap$
$\neg((A_1 \cap B_1 \cap A_1 \cap B_2) \cup (A_1 \cap B_1 \cap A_1 \cap B_3) \cup ... \cup (A_n \cap B_{m-1} \cap A_n \cap B_m))$.

We are going to show that $(O_1 \cap O_2) \Leftrightarrow (P_1$ merge $P_2)$. The proof follows the divide and conquer approach - we first split up both of the expressions in two disjoint parts, then show that the subexpressions are equivalent.

For the first part,
$((A_1 \cap B_1) \cup (A_1 \cap B_2) \cup ... \cup (A_n \cap B_m)) \Leftrightarrow$
$(A_1 \cup A_2 \cup A_3 \cup ... \cup A_n) \cap (B_1 \cup B_2 \cup B_3 \cup ... \cup B_n)$
holds because $\cap$ distributes over $\cup$.

After eliminating $\neg$, the second part is to prove that

$$((A_1 \cap B_1 \cap A_1 \cap B_2) \cup (A_1 \cap B_1 \cap A_1 \cap B_3) \cup ... \cup (A_n \cap B_{m-1} \cap A_n \cap B_m)) \quad (1)$$

is equivalent to:

$$((A_1 \cap A_2) \cup (A_1 \cap A_3) \cup ... \cup (A_{n-1} \cap A_n)) \cap ((B_1 \cap B_2) \cup ... \cup (B_{n-1} \cap B_n)) \quad (2)$$

After applying distributive law, (2) can be written in DNF as well:

$$((A_1 \cap A_2 \cap B_1 \cap B_2) \cup (A_1 \cap A_2 \cap B_1 \cap B_3) \cup ... \cup (A_n - 1 \cap A_n \cap B_{(m-1)} \cap B_m)) \quad (3)$$

Having both of the expressions in DNF, we can easily show that each disjunct from (1) can be expressed using a combination of disjuncts in (3), and vice-versa. Having $(1) \subseteq (3)$ and $(3) \subseteq (1)$ means that $(1) = (3)$, thus these subexpressions are equivalent,too. Having proven that the corresponding subexpressions are equivalent, we conclude that $(O_1 \cap O_2) \Leftrightarrow (P_1$ merge $P_2)$.**Q.E.D**