Semantic Annotations for WS-Policy

Sebastian Speiser
Karlsruhe Service Research Institute (KSRI)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: speiser@kit.edu

***Abstract*—WS-Policy is a standard to express requirements and capabilities in Web service systems. Policies are based on domain-specific assertions. In this paper we present a lightweight approach to semantic annotations of policy assertions. The approach allows matching of requirements and capabilities based not only on the syntactical representation of their corresponding assertions but also on their semantic meaning. Besides vocabulary mismatches our approach can also handle granularity mismatches, e.g. if two capabilities in combination satisfy a single requirement. We present a validation of our approach consisting of a performance evaluation and the realization of a use case, both based on our implementation of the semantic policy matching algorithm. We furthermore show the advantages of our approach compared to existing related work.**

I. Introduction

Web services are broadly used for different tasks such as giving programmatic access to Web applications, providing standardized interfaces for information sources, or enabling the creation of lightweight mashup applications. Furthermore Web services are the most commonly used technology to implement service-oriented architectures (SOA). The functional interface in form of input and output messages and their structure can be described using standards such as WSDL. Services and service users have also non-functional requirements, e.g. that communication is encrypted or that the WS-Addressing standard is supported. WS-Policy provides a standard that can be used to specify policies that express requirements and capabilities in systems based on Web services [1].

Consider the fictitious company MagicMonkeyTrade that wants to implement an automated stock trading system. The company maintains a repository of internal and external Web services with their corresponding WS-Policies. The project manager assigns Alice the task to find a suitable service that delivers stock quotes. Alice is told that the service must be highly secure, for two reasons: (i) the stock quotes must be reliable and not modifiable by attackers, and (ii) no third party should be able to see which stocks are monitored by the company, as this could give hints about their secret trading strategy. Alice retrieves two Web services from the repository, that can deliver stock quotes. One is provided by Bob and supports the `Basic256` algorithm suite as defined in the WS-SecurityPolicy standard, and signed output messages. The other service is provided by Eve and supports the `Basic128` algorithm suite and signed headers.

In order to determine if one of the Web services fulfills Alice's requirement, she has to check if her requirements are a subset of the offered capabilities. This operation is called policy matching and requires domain-dependent knowledge to decide if a requirement (e.g. high security) is fulfilled by a number of capabilities (e.g. support for the `Basic256` algorithm suite and signed messages). Without background knowledge a policy matching tool can only work with syntactical comparisons, which does not work in our example, where requirements and capabilities are specified on different levels of abstraction.

Currently the background knowledge about different domains is captured in natural language specifications, e.g. WS-SecurityPolicy [2]. Based on such a specification, tool producers have to develop, test and integrate specific code for every domain that they want to support for policy matching. We propose a lightweight extension of WS-Policy with references to ontology classes defined in OWL. This allows a formal specification of background knowledge, that can be reused and dynamically loaded by policy matchers. Matching decisions are based on formal semantics, interpreted by standard reasoner software. Furthermore ontologies are well suited for mapping between different vocabularies that are likely to appear in the Web with heterogeneous service providers and users. Thus our proposed approach can increase recall, by matching offered and requested capabilities that have the same meaning but are expressed using different vocabularies, e.g. matching "strong symmetric encryption" with "AES256 encryption".

Compared to previous works that linked ontologies and Web service policies, our approach has the following key advantages:

- Subtle addition of semantic annotations: existing policies without annotations can be processed by tools that are semantic-aware, and semantically annotated policies can be processed by existing tools that ignore annotations.
- It exploits the strengths of existing standards: it uses WS-Policy for policy structuring and OWL for capability matching.

- Ability to handle granularity mismatches: several capabilities in combination can match a single requirement.

In order to validate our approach we implemented a prototype for the matching of semantically annotated WS-Policies. Based on the prototype we realized the MagicMonkeyTrade use case, which can be accessed online via a Web interface available at `http://sawspolicy.appspot.com`. Furthermore we evaluated the performance of semantic policy matching. We have found that while performance is lower compared to pure syntactical matching, in absolute values the performance penalty is very small and compensated by increased recall, i.e. more matching services can be found.

The rest of the paper is structured as follows. Section II gives an overview of WS-Policy and the Web Ontology Language (OWL). Our approach to semantically annotating policies and a definition of policy matching is given in Section III. We describe our implementation in Section IV. The validation in Section V includes a performance evaluation and a description how the MagicMonkeyTrade use case can be realized. Related work is presented in Section VI including an overview of the differences and drawbacks compared to our approach. Finally we conclude in Section VII and give an outlook to future work.

II. Background

Our approach to semantically annotated Web service policies is based on several W3C standards. Most notably we extend Web Services Policy (WS-Policy) [1] with semantic annotations referring to ontologies defined using OWL [3]. These two standards are explained in the following.

A. WS-Policy

WS-Policy defines an XML-based language to specify policies for Web services. Policies can be used to express capabilities or requirements of Web service users and providers. The basic building block of policies is the assertion. Assertions are either simple or complex XML elements identified by qualified names (qnames, consisting of namespace URI and local part). They are defined in external standards (e.g. WS-SecurityPolicy) where they are given a domain-specific meaning, e.g. that a message contains an authentication token. A set of assertions that is a valid required or provided configuration of a policy is called an alternative. A policy is a collection of alternatives. Formally we define a policy $P = \{\mathrm{Alt}_1, \ldots, \mathrm{Alt}_n\}$ as a set of alternatives of the form $\mathrm{Alt}_i = \{A_{i,1}, \ldots, A_{i,k_i}\}$, where $A_{i,j}$ is an assertion.

The WS-Policy standard describes a domain-independent mechanism to find alternatives that are compatible to two policies: the intersection. Two alternatives are compatible if for each assertion in one alternative there is a compatible assertion in the other alternative. It is termed domain-independent because compatibility of assertions is defined

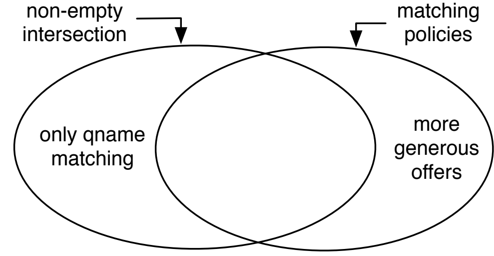

Figure 1. Comparison of Intersection and Matching for Offer and Request Policy

as equality of their qnames without regarding their further structure and contents. In our approach we cover policy matching, where one policy is an offer that specifies capabilities and the other policy is an request that specifies requirements. Offer and request match, if there is an offer alternative satisfying a request alternative. We define that alternative Alt^o syntactically satisfies alternative Alt^r, iff

$$\forall A^r \in \mathrm{Alt}^r . \exists A^o \in \mathrm{Alt}^o : A^o \text{ satisfies } A^r,$$

where assertion satisfaction is given, if the corresponding XML elements have the same content. The relation between intersection and matching is illustrated in Figure 1. The difference is that non-empty intersection can comprise more policy pairs, as compatibility is checked based only on qnames, and matching can comprise more policy pairs, as it is only checked if each required assertion is also in the offer and not if every offered assertion is also required, thus allowing more generous policy offers than required.

Policy matching is useful for the following operations:

- Service discovery: a set of offer policies is matched against a request policy that specifies the desired service properties.
- Compatibility check: a user specifies an offer policy with his capabilities and matches it against the policy of the service that he wants to invoke, in order to see whether he can fulfill its requirements.

The XML serialization of a policy is created using the elements `ExactlyOne` and `All`. The meaning of `ExactlyOne` is, that one of its child elements must hold, whereas `All` means that all child elements must hold. Consider the following policy $P = \{\{\mathtt{A},\mathtt{C}\}, \{\mathtt{A},\mathtt{D}\}, \{\mathtt{B},\mathtt{C}\}, \{\mathtt{B},\mathtt{D}\}\}$, which is serialized as (`A`, `B`, `C`, `D` are assertion elements):

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>              <!-- Alt 1 -->
        <A /><C />
    </wsp:All>
    <wsp:All>              <!-- Alt 2 -->
        <A /><D />
    </wsp:All>

   <wsp:All>             <!-- Alt 3 -->
        <B /><C />
```

```
    </wsp:All>
    <wsp:All>            <!-- Alt 4 -->
        <B /><D />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

This corresponds to the normal form of WS-Policy, where the policy has only one child: an `ExactlyOne` element, which contains for each alternative an `All` element that in turn contains the assertions of the alternative. WS-Policy allows for a compact representation where the both elements can be arbitrarily nested. The example above can be represented more compact as follows (note that `Policy` has the same meaning as `All`):

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <A /><B />
  </wsp:ExactlyOne>
  <wsp:ExactlyOne>
    <C /><D />
  </wsp:ExactlyOne>
</wsp:Policy>
```

Note that every compact representation can be normalized, and thus the formal model can abstract from the syntactical representation of the policy.

B. OWL: Web Ontology Language

In computer science, the term ontology refers to a formalized specification of knowledge about a domain (cf. Gruber [4]). Ontologies define a vocabulary referring to concepts of the domain, their attributes and relations amongst them. Specifying this knowledge in a machine-readable language with formally defined semantics, allows computers to interpret this knowledge and possibly infer further implicit knowledge. The Web Ontology Language (OWL) [3] is a W3C standard defining a language, based on Web technologies, for specifying ontologies with formal semantics based on description logics (DL, cf. Baader et al. [5]). The DL used for OWL has the advantage that it is decidable and inference algorithms exist that are more efficient than first-order logic reasoning. There exist several implementations of OWL inference engines, so-called OWL reasoners.

OWL ontologies can describe individuals, relations between individuals, classes which refer to groups of individuals that have something in common, and relations between classes. Individuals, classes and relations are all labeled with unique resource identifiers (URI). In the following we will use the names `C,D` for class names, and `R` as a relation name. In OWL we can define that class `C` is a subclass of class `D`, denoted as $\mathtt{C} \sqsubseteq \mathtt{D}$, which means that every individual belonging to `C` also belongs to `D`. If two classes `C` and `D` mutually are subclasses of each other, they are called equivalent, denoted as $\mathtt{C} \equiv \mathtt{D}$. Expressing subclass relationships is not only possible for simple, named classes but also for complex class definitions. The following table shows the two types of complex class definitions used in this paper:

$\mathtt{C} \sqcap \mathtt{D}$	Intersection, referring to all individuals belonging to `C` and `D`.
$\exists\, \mathtt{R.C}$	Existential restriction, referring to all individuals that have an `R` relationship to an individual belonging to class `C`

Based on such class descriptions, a reasoner can infer new relations between classes. Consider the following example:

$$\begin{aligned} \mathtt{AES256} &\sqsubseteq \mathtt{SymmEncryption} \\ \mathtt{A} &\sqsubseteq \exists \mathtt{uses.AES256} \end{aligned}$$

From this knowledge an OWL reasoner can infer the axiom $\mathtt{A} \sqsubseteq \exists \mathtt{uses.SymmEncryption}$, as it is known that each individual belonging to `A` has a `uses` relationship to an individual belonging to `AES256`, and thus also belongs to `SymmEncryption`.

Besides the human-friendly description logics syntax, that we used so far, OWL also specifies serializations in machine-friendly formats, e.g. RDF/XML.

III. SEMANTIC ANNOTATIONS FOR WS-POLICY

Currently the check, if an offered assertion satisfies a required assertion is done based on syntactical properties that are interpreted according to the meaning that is given in a human readable specification. Comparison can be domain-independent by just comparing the qnames of the assertions, as it is proposed in WS-Policy intersection. However this leads to wrong matches, e.g. two `<sp:AlgorithmSuite>` assertions would be matched, even if their child nodes specify incompatible algorithms. Another possibility is to compare equivalence of the assertion element, its attributes and recursively for its child elements. This can be overly strict, e.g. two assertions would not match, even though they specify compatible but syntactically different properties. Domain-dependent matching can overcome these problems, e.g. by introducing a plugin architecture for Java code modules that compare assertions from the same specification. However this requires that either all policy engines support the same plugin architecture, or that numerous plugins for the same specification have to be written and tested for compatibility. Another drawback is that matching of different vocabularies with the same meaning is only possible if the plugins cover different specifications.

We propose to use OWL ontologies as background knowledge, as they provide a declarative way to specify relationships and matchings between different assertions. In our approach assertions are represented by classes and an offer assertion class `O` matches a request class `R`, if $\mathtt{O} \sqsubseteq \mathtt{R}$, i.e. if the offer is a subclass of the request. Consider for example

Table I
DEFINITION OF ASSERTION CLASS

Attributes given	Assertion Class
none	$\mathtt{A}$
modelReference	$\mathtt{A} \equiv \mathtt{M}$
liftingSchema	$\mathtt{A} \equiv \mathtt{L}$
both	$\mathtt{A} \equiv \mathtt{M} \sqcap \mathtt{L}$

that we have the following background knowledge:

$$\begin{aligned} & \mathtt{AlgorithmSuite} \sqcap \exists \mathtt{suite.Basic256} \\ \equiv \; & \mathtt{AlgorithmSuite} \sqcap \exists \mathtt{symmetric.AES256} \sqcap \\ & \exists \mathtt{digest.SHA1} \sqcap \ldots \end{aligned}$$

With this knowledge, a standard OWL reasoner can infer that an assertion offering the Basic256 algorithm suite, is a subclass of an assertion requesting symmetric encryption with AES256, thus a match is found. Furthermore OWL is suitable for expressing mapping ontologies, as classes are identified by URIs that can be easily reused across different ontologies. For example one could specify that $\mathtt{provider} : \mathtt{AESEncrpytion} \sqsubseteq \mathtt{user} : \mathtt{StrongEncr}$.

We propose a technique for annotating policy assertions with semantic references. The approach is inspired by SAWSDL [6], which is a lightweight way to give semantic annotations to WSDL files and XML schema definitions. References are defined by adding two attributes to assertion XML nodes, which can be ignored by existing applications that do not need the annotations. This allows for a subtle introduction of semantics into policies and engines.

Each assertion can contain one or both of the optional attributes modelReference and liftingSchema. The value of modelReference is defined to be a URI denoting an OWL class (in the following denoted as M). The value of liftingSchema is a link to a XSL transformation that generates a complex OWL class definiton serialized as RDF/XML from the assertion's XML node and its children. The complex class is denoted as L in the following. The XSL transformations are useful to annotate a whole class of assertions without specifying a named class for each, e.g. AlgorithmSuite elements can be transformed into a class description that states that there exists a suite with the value as specified in its child elements. For each assertion a new class, denoted A, is created and defined as equivalent to the intersection of L and M, if they are given (see Table I).

Let us consider the following example, consisting of four artifacts: request policy, offer policy, XSL transformation and an ontology describing background knowledge. The request policy uses a high level vocabulary requiring strong encryption:

```
<wsp:policy>
  <ex:Assertion modelReference=
       "http://ex.org/#StrongEncr" />
</wsp:policy>
```

The offer policy is based on technical security policy assertions and links to corresponding ontological classes:

```
<wsp:policy>
  <sp:AlgorithmSuite modelReference=
       "http://ex.org/#AlgoSuite"
       liftingSchema="security.xsl">
     <sp:Basic256 />
  </sp:AlgorithmSuite>
</wsp:policy>
```

The referenced "security.xsl" transformation lifts the XML description to a class definition:

```
<xsl:stylesheet>
 <xsl:template match="/AlgorithmSuite">
  <rdfs:subClassOf>
   <owl:Class>
    <owl:intersectionOf>
     <rdf:Description about="#AlgoSuite" />
     <xsl:apply-template select="node()" />
    <owl:intersectionOf>
   </owl:Class>
  </rdfs:subClassOf>
 </xsl:template>
 <xsl:template match="Basic256">
  <!-- RDF/XML for
       Exists suite.Basic256 -->
 </xsl:template>
</xsl:stylesheet>
```

This generates the following complex concept definition for the AlgorithmSuite assertion: $\mathtt{Algo} \sqcap \exists \mathtt{suite.Basic256}$. Assume that additionally the following background knowledge is given:

$$\begin{aligned} \mathtt{Algo} \sqcap \exists \mathtt{suite.Basic256} &\sqsubseteq \mathtt{Encr} \sqcap \exists \mathtt{symm.AES256} \\ \mathtt{Encr} \sqcap \exists \mathtt{symm.AES256} &\sqsubseteq \mathtt{StrongEncr} \end{aligned}$$

A standard OWL reasoner can infer that the class of the offer policy assertion ($\mathtt{Algo} \sqcap \exists \mathtt{suite.Basic256}$) is a subclass of the assertion in the request policy ($\mathtt{StrongEncr}$). Thus the offer matches the request.

We furthermore propose that a semantic WS-Policy engine maintains a global repository of XSLT files that map assertions from common policy assertion specifications (e.g. WS-SecurityPolicy) to semantic annotations. One way to realize this, would be to associate qname patterns to XSL transformations, that are applied if no explicit semantic annotations are given for an assertion matching the pattern. Such a repository allows semantic policy matching even for policies that are not annotated by their specifiers.

Matching assertions based on their class definitions allows us to deal with vocabulary mismatches. However it is also possible that there are granularity mismatches. Consider for example that the request policy requires high security which is defined as the combination of strong encryption and signed messages. An offer based on technical vocabulary may define two separate assertions for encryption and message signing. Thus we have to perform subclass checking

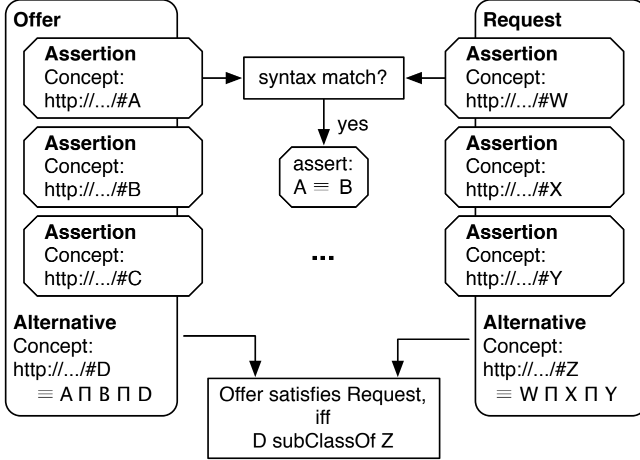

Figure 2. Illustration of algorithm: Syntax checks complementing class definitions from semantic annotations and background knowledge

for the intersection of the classes of multiple assertions. However we cannot simply intersect all assertions of an alternative, as this might be too restrictive in cases where some assertions are not semantically annotated but have a match on syntactical basis. Therefore we define that an offer alternative satisfies a request alternative, if there are two assertion subsets of the request alternative, such that every assertion in the first subset is syntactically matched by an offered assertion, and the intersection of the classes from the second subset is a superclass of the intersection of the classes from the offer alternative. The union of both subsets has to include all assertions from the request alternative. Formally we specify that an offer alternative Alt^o matches a request alternative Alt^r, iff

$$\begin{aligned} &\exists \mathrm{Sy} \subseteq \mathrm{Alt}^r \ \exists \mathrm{Se} \subseteq \mathrm{Alt}^r : \\ &\quad \mathrm{Sy} \cup \mathrm{Se} = \mathrm{Alt}^r \ \wedge \\ &\quad (\forall A^r \in \mathrm{Sy} \ \exists A^o \in \mathrm{Alt}^o : A^o \text{ satisfies } A^r) \ \wedge \\ &\quad (\bigsqcap_{A^o \in \mathrm{Alt}^o} c(A^o) \sqsubseteq \bigsqcap_{A^r \in \mathrm{Se}} c(A^r)), \end{aligned}$$

where $c(A)$ denotes the ontology concept that A is annotated with.

IV. Implementation of SA WS-Policy Matching

We have implemented a prototype for matching of semantically annotated WS-Policies (SA WS-Policy) by extending the policy engine Apache Neethi[1], a WS-Policy framework, and using the OWL reasoner HermiT[2]. We extended the policy loading mechanism to generate a temporary ontology for each policy. For each assertion a new class is added to the ontology and if specified, it is set equivalent to the classes obtained from the `modelReference` attribute and the `liftingSchema` transformation as described in the previous section. Afterwards the policy is normalized and for each alternative a new class is created that is equivalent to the intersection of the classes representing the child assertions. The current implementation does not yet support a global XSLT repository.

Matching policies is based on an OWL reasoner that loads the ontologies of the policies as well as any ontologies containing background knowledge (e.g. vocabulary mappings). The actual matching is first done syntactically where for each equivalent request and offer assertion, the equivalence is transferred to the ontological knowledge base as an equivalence axiom of the classes representing the two assertions. If the syntactical matching was not successful, offer alternatives are tested if they are equivalent to or a subclass of a request alternative. Due to the preceding transfer of syntax to semantic matches, this approach can also detect if an offer satisfies the request by a combination of syntactical and semantic assertion matches. As noted previously it is not sufficient to test only pairs of single assertions for semantic matchings as this would not be able to overcome granularity mismatches. Figure 2 illustrates the algorithm for matching alternatives.

Our implementation can be tested online via a Web interface, and is also available with source code, both at `http://sawspolicy.appspot.com`. It works both with semantically annotated policies and with policies that have no annotations. In the latter case, a pure syntactic matching is performed. Furthermore it is possible to ignore the annotations in order to compare the performance of pure syntactic against semantic matching.

V. Validation

This section is split into two parts. First we describe an experiment for performance evaluation and discuss the obtained results. Secondly we describe the realization of the MagicMonkeyTrade use case, which was presented in the introduction.

A. Performance Evaluation

For performance measurements we compared pure syntactical policy matching to our approach. As a basis for our test data we selected 4 different assertion types from WS-SecurityPolicy, and in turn for each type 4 different specific assertions. These are listed in Table II.

Each assertion is annotated with a reference to an ontology, that specifies background knowledge, such as: `Basic256` and `Basic256Sha256` imply the symmetric encryption algorithm AES256, and `BodyHeader` implies both signed `Body` and `Header`.

Policy offers were created by generating all policies that have 2 or 3 assertion types with each 1 or 2 assertions. The different assertion types are `wsp:ExactlyOne` elements containing the assertions, and combined via `wsp:All` to

[1]Available at `http://ws.apache.org/commons/neethi/`
[2]Available at `http://hermit-reasoner.com/`

Table II
ASSERTIONS OF DIFFERENT TYPES FOR PERFORMANCE TEST

Assertion Type	**Assertions**
`SecurityToken`	`HttpsToken`, `SamlToken`, `KerberosToken`, `X509Token`
`SecurityHeader`	`Lax`, `Strict`, `LaxTimestampFirst`, `LaxTimestampLast`
`SignedParts`	`Body`, `Header`, `BodyHeader`, only WS-Addressing `Headers`
`AlgorithmSuite`	`Basic256`, `TripleDes`, `Basic128`, `Basic256Sha256`

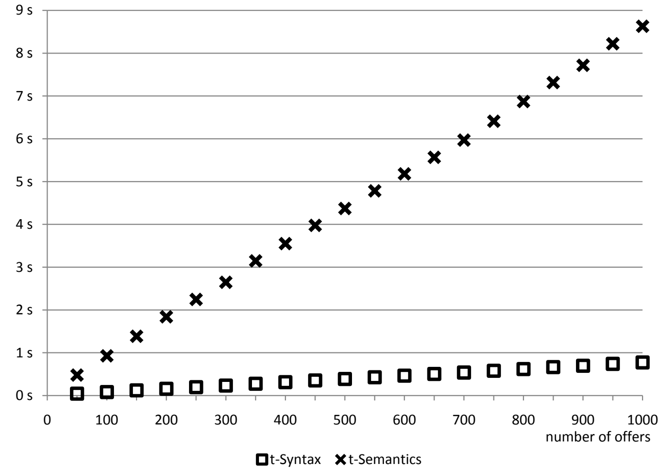

Figure 3. Results of performance evaluation

form the complete policy. In total this results in 4,600 policies. We distributed the offers randomly in buckets of 50 policies each.

Policy requests were created by generating all policies that require 2 assertions from different assertion types. We matched 10 requests at a time against offer policies taken from an increasing number of buckets. We measured the time consumed by policy matching for each group of request policies and for each number of offer policies and took the average value across the different request groups. The times shown are the durations of the matching process, including syntactical matching, reasoning and subclass checking for the alternatives. For both syntactical and semantic matching the setup and loading times are not measured. The measurements were conducted using the Java method `System.currentTimeMillis()`. The tests were preceded by a dry run of syntax and semantic matching, during which the just-in-time compiler of the Java virtual machine could optimize the execution. The results are shown in Figure 3. The tests were run on a laptop with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of memory.

The chart shows that syntactical matching is faster than semantic matching. This is not surprising, as our semantic approach includes syntax matching as a subtask. However we put the numbers in perspective by the following observations:

- The absolute consumed time for matching one offer and one request is small: about 1 millisecond with semantics enabled. As each matching of an request and offer pair is independent of other pairs, scalability can be achieved by parallelization.
  reasoning. Large parts of the ontology underlying this reasoning is independent of the requests, namely the background knowledge and the ontologies of the offer policies. An optimized semantic policy repository could precompute inferences from the request-independent ontologies and thus significantly reduce matching time.

Compared to syntactical matching, our approach brings the benefit of increased recall. For example a request requiring signed headers, can be satisfied by a policy offering signed body and header, which is not possible only using syntax without background knowledge. We did not conduct numerical measurements of recall improvement, as in the absence of any standard test data, our approach would be the yardstick and the test data could be arbitrarily tweaked to yield the desired result. Instead we present in the next section how a practical example can be realized with our approach.

B. Realization of Use Case

Remember that Alice has the task to select a highly secure Web service that can deliver stock quotes for her company MagicMonkeyTrade. She retrieved two services, provided by Bob, respectively Eve, which specify WS-Policies on a low technical level.

First Alice specifies her high level request policy:

```
<wsp:Policy>
  <ex:HighSecurity modelReference=
    "http://ex.org/highlevel#HighSecurity" />
</wsp:Policy>
```

She analyzes the requirements of her project manager and realizes that high security in his sense can be realized by signed messages and strong encryption, so that nobody can modify quotes or monitor the requested quotes. She specifies in the high level ontology:

$$\texttt{StrongEncr} \sqcap \texttt{SignedMessage} \sqsubseteq \texttt{HighSecurity}.$$

However the policies of the offered services use lower level vocabulary. The policy of Bob's service is:

```
<wsp:Policy>
  <sp:AlgorithmSuite sawsp:modelReference=
      "http://ex.org/lowlevel#Basic256">
      <sp:Basic256 />
  </sp:AlgorithmSuite>
  <sp:SignedParts sawsp:modelReference=
  "http://ex.org/lowlevel#SignedBodyHeader">
    <sp:Body />
    <sp:Header />
  </sp:SignedParts>
</wsp:Policy>
```

Eve's service has the following policy:

```
<wsp:Policy>
  <sp:AlgorithmSuite sawsp:modelReference=
      "http://ex.org/lowlevel#Basic128">
      <sp:Basic128 />
  </sp:AlgorithmSuite>
  <sp:SignedParts sawsp:modelReference=
      "http://ex.org/lowlevel#SignedHeader">
      <sp:Header />
  </sp:SignedParts>
</wsp:Policy>
```

The low level ontology, which is used by Bob and Eve in their annotations, specifies besides others that the `Basic128/256` algorithm suites support the AES encryption algorithm with keylength of 128, respectively 256 bits, and that AES is a symmetric encryption algorithm.

Alice is a specialist for automated trading systems but has no knowledge about encryption algorithms or digital signatures. Thus she consults with MagicMonkeyTrade's IT department, who tell her about a mapping ontology that relates low level security concepts to the high level concepts used by Web service users and their managers. It is maintained by the IT department which updates the mappings regularly, e.g. by checking which encryption algorithms can be considered as secure. The mapping ontology contains besides others the axiom:

$$\texttt{SymmetricEncr} \sqcap \exists \texttt{keylength}.256 \sqsubseteq \texttt{StrongEncr}.$$

It also defines that for a message to be considered a `SignedMessage` both its header and body must be signed.

Alice loads the mapping ontology into her semantic WS-Policy matcher, and finds out that Bob's service fulfills her needs of high security, given the background knowledge from the annotations in the policies and the high level, low level and mapping ontologies. This conclusion is based on the following inferences: Bob's service supports `Basic256` and thus AES, a symmetric encryption algorithm, with 256 bit keys. Therefore the service provides strong encryption, which together with the support of signed messages means high security. Eve's service is neither a syntactical nor a semantic match and is discarded for this project.

VI. Related Work

Several works have dealt with the underpinning of WS-Policy with OWL ontologies.

Sriharee et al. propose an approach that uses WS-Policy assertions that point to policies represented in an OWL model [7]. Their model defines policies as sets of rules, which have to be evaluated using an external rule-based reasoning system. In contrast to our work Sriharee et al. do not concentrate on matching of WS-Policies as a whole but on special assertions that model business rules and require reasoning beyond OWL.

Verma et al. reuse the qname of a policy assertion as identifier for an ontology class [8]. Furthermore they introduce optional attributes for assertions, one of them for specifying how other assertions can be matched, e.g. by a subclass relationship. Their approach, like ours, extends WS-Policy carefully and does not destroy compatibility with existing tools. However it has two drawbacks. First, it does not allow users to specify custom classes for an assertion, but reuses its qname. Second, it compares single assertions with each other, and thus cannot overcome granularity mismatches.

The work by Dumitru et al. combines WS-Policy with semantic Web services modeled with WSMO in two ways [9]: (i) using policies as non-functional properties in WSMO descriptions, and (ii) using WSMO service descriptions as WS-Policy assertion. The second approach is related to ours. However it does not aim at adding semantics to existing policies, but introduces new assertions that need reasoning about WSMO services in order to decide their matching.

Kolovski and Parsia show how WS-Policies can be mapped to a class definition in OWL-DL with default rules [10]. Like Verma et al. they map each assertion type to a single class. To compute the compatibility of two policies they also compare individual assertions which is ineffective for granularity mismatches.

Several other approaches present ontology-based policy languages that are not based on WS-Policy, e.g. KAoS by Uszok et al. [11] and Rei by Kagal et. al [12]. Besides their drawback of not being compatible to WS-Policy, there are other disadvantages, e.g. Rei requires rule-based reasoning that is outside of the scope of the underlying ontology language. This is common with some of the works discussed above, that also do not only rely on OWL but need extensions. In our work we accepted that OWL is not a suitable formalism for representing the structure of policies, composed of assertions. Thus we rely on WS-Policy with its clear semantics for assertion combination and employ OWL reasoning for assertion matching.

Anderson presents with WS-PolicyConstraints a language to specify policy assertions with semantics defined by XPath expressions on Web service messages [13]. Such assertions allow policy engines to check compliance of messages without relying on domain-dependent assertions with semantics defined in external standards. One of the drawbacks of WS-PolicyConstraints is that it is dependent on the message structure and syntax and thus it is difficult to map between different vocabularies. Another problem is that for matching

two policy assertions, it is necessary to check containment for XPath queries. The approach by Anderson is useful for testing message compliance and can be complemented by our proposed semantic annotations to support policy matching.

VII. Conclusions

In this paper we showed a lightweight approach to specify semantic annotations in WS-Policy. We defined the notion of syntactical and semantic policy matching. We validated our approach by realizing an exemplary use case and implementing a semantic policy matcher. Performance evaluation showed that our unoptimized prototype is slower than pure syntactical matching but in absolute terms still sufficiently fast. The advantage of semantic policy matching over syntactical matching is the increased recall, meaning that more correctly matching policies are found. Compared to previous works that combined WS-Policy with ontologies our approach has the following key advantages: (i) it is compatible to existing tools and policies, and (ii) it can handle granularity mismatches.

As future work we plan to implement the global XSL transformation registry as proposed in Section III and develop transformations for common standards such as WS-SecurityPolicy. Furthermore we want to extend our engine to incorporate the WS-PolicyAttachment standard that can be used to specify the relations between policies and their subjects.

Another plan for the future is build a justification component for the policy matcher. Currently the only result of policy matching is a boolean decision if the policies match or not. The justification component will give the reasons for a decision that are useful for two purposes: (i) in case of a match it points to a compatible pair of policy alternatives, and (ii) in case of non-matching policies, it can help the owner of the offer policy to see what he has to change in order to satisfy the request.

Acknowledgement

The authors would like to thank Andreas Harth for the useful discussions. The work presented in this paper was supported by the European project SOA4All.

References

[1] W3C, *Web Services Policy 1.5 - Framework*. W3C Recommendation, 2007, available at `http://www.w3.org/TR/ws-policy/`.

[2] OASIS, *WS-SecurityPolicy 1.2*. OASIS Standard, 2007, available at `http://www.oasis-open.org/specs/`.

[3] W3C, *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 2009, available at `http://www.w3.org/TR/owl2-overview/`.

[4] T. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, Eds., *The Description Logic Handbook*. Cambridge University Press, 2003.

[6] W3C, *Semantic Annotations for WSDL and XML Schema*. W3C Recommendation, 2007, available at `http://www.w3.org/TR/sawsdl/`.

[7] N. Sriharee, T. Senivongse, K. Verma, and A. Sheth, "On Using WS-Policy, Ontology, and Rule Reasoning to Discover Web Services," in *Intelligence in Communication Systems, IFIP International Conference, INTELLCOMM 2004*, 2004, pp. 246–255. [Online]. Available: http://www.springerlink.com/content/w1p3f4tkp6auvx2g

[8] K. Verma, R. Akkiraju, and R. Goodwin, "Semantic matching of web service policies," in *Proceedings of the Second on Semantic and Dynamic Web Processes (SDWP2005)*, 2005.

[9] R. Dumitru, J. Kopecký, I. Toma, and D. Fensel, "Aligning WSMO and WS-Policy," in *Proceedings of the Second Semantic Web Policy Workshop (SWPW'06)*, 2006.

[10] V. Kolovski and B. Parsia, "WS-Policy and Beyond: Application of OWL Defaults to Web Service Policies," in *Proceedings of the Second Semantic Web Policy Workshop (SWPW'06)*, 2006.

[11] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement," in *Proceedings of Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, 2003.

[12] L. Kagal, T. Finin, and A. Joshi, "A Policy Language for a Pervasive Computing Environment," in *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, 2003.

[13] A. H. Anderson, "Domain-Independent, Composable Web Services Policy Assertions," in *Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*. IEEE Computer Society, 2006.