Authorization and Privacy for Semantic Web Services

Lalana Kagal and Tim Finin, *University of Maryland*

Massimo Paolucci, Naveen Srinivasan, and Katia Sycara, *Carnegie Mellon University*

Grit Denker, *SRI International*

Web Services will soon handle users' private information. They'll need to provide privacy guarantees to prevent this delicate information from ending up in the wrong hands. More generally, Web Services will need to reason about their users' policies that specify who can access private information and under what conditions.

Providing guarantees for security and privacy is paramount to the success of Semantic Web Services. In this article, the authors describe OWL-S policy annotations and extend the OWL-S Matchmaker and OWL-S Virtual Machine to support the processing of those policies.

These requirements are even more stringent for Semantic Web Services that exploit the Semantic Web to automate their discovery and interaction because they must autonomously decide what information to exchange and how.

In our previous work, we proposed ontologies for modeling the high-level security requirements and capabilities of Web Services and clients.[1] This modeling helps to match a client's request with appropriate services—those based on security criteria as well as functional descriptions. For example, a Web Service could state that it can perform OpenPGP encryption and requires an invoker that can authenticate itself and communicate in XML. We added functionality to the DAML-S Matchmaker[2] (an earlier version of the OWL-S Matchmaker) that lets it verify if a service's capabilities fulfill the invoker's security requirements and vice versa. Our results assist coarse-grain matching decisions such as "Does the service provide encryption?" or "What kind of credential do I have to provide to authenticate myself to the service?"

In this article, we propose a more fine-grain security markup of service parameters in OWL-S. We extend our previous work with annotations about the security and privacy policies of services. We express these annotations in Rei, a logic-based language that lets you define rules and constraints over domain-specific ontologies.[3] Our work aims to provide security and policy annotations for OWL-S service descriptions and enforcements by extending the OWL-S Matchmaker for policy matching and the OWL-S Virtual Machine (VM)[4] with policy enforcement and security mechanisms.

Role of policies

Policies specify who can use a service and under which conditions, how information should be provided to the service, and how the provided information will be used. Policies should be part of Web Service representations—particularly those on the Semantic Web (see the "Related Work" sidebar for more background information).

In our work, a client-server model involves a client that wants to invoke a Web Service. We view the use of policies as *symmetric*—policies that constrain both the provider and requester. You can easily extend this model to a service-service architectural model.

Here, we address two kinds of policies: *privacy* and *authorization*. Privacy policies specify under what conditions you can exchange information and the legitimate uses of that information. For example, a privacy policy might say that a provider could give a requester a key to access private information only if the key is encrypted during transmission. When a requester discovers the policy, it should decide whether it can satisfy this condition. The requester might have its own privacy policy that requires keeping certain information confidential, so it likewise can't share unencrypted private information. The requestor's privacy policy prevents it from interacting with Web Services that don't perform the needed encryption.

Privacy policies help specify data confidentiality during transmission as well as after receipt. Consider a service that says it won't distribute details

1541-1672/04/$20.00 © 2004 IEEE
Published by the IEEE Computer Society

Related Work

Today, Web Services—and Semantic Web Services even more so—have a ways to go to realize their potential. Standardization groups such as the Organization for the Advancement of Structured Information Standards (OASIS) and the World Wide Web Consortium (W3C) have focused on syntactical issues of Web Services' interoperability and security. But these organizations are just starting to explore how semantically rich annotations will facilitate the discovery, selection, composition, invocation, and runtime monitoring of Web Services.

Relevant related work stems from the areas of security for Web Services and trust and privacy policies for the Semantic Web. Lately, significant standardization efforts have arisen for XML-based security, such as WSS (Web Services Security) and SAML (Security Assertion Markup Language), sponsored by OASIS technical committees, and the Liberty Alliance Project's security specifications. This work doesn't consider Web Services' semantic aspects. In addition to the specification of security, efforts on Semantic Web trust and privacy policies—although not specifically targeted toward Semantic Web Services—are also relevant for our work.[1,2]

There has also been a significant amount of research in security policies for distributed systems. KAoS provides a policy representation language based on OWL.[3] Although this is an interesting approach, OWL can't adequately capture the full range of policy constraints. Several efforts are under way to add syntax for rules in OWL.[4] Ponder is a policy specification language developed at Imperial College.[5] Although flexible and expressive, it's mostly a syntactic language and doesn't lend itself well to Semantic Web Services.

Rei, the language we use in this article, draws on distributed policy work by Morris Sloman and Emil Lupu.[6–8] It has an RDF Schema representation and includes a Prolog-like notation for expressing rules on policy objects that exceeds what can be done in DAML+OIL and OWL.

References

1. J.M. Bradshaw et al., "Representation and Reasoning for DAML-Based Policy and Domain Services in KAoS and Nomads," *Proc. Conf. Autonomous Agents and Multiagent Systems* (AAMAS 03), ACM Press, 2003, pp. 835–842.
2. F. Gandon and N. Sadeh, "Semantic Web Technologies to Reconcile Privacy and Context Awareness," *Web Semantics J.*, vol. 1, no. 3, 2004, pp. 241–260.
3. A. Uszok et al., "KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement," *Proc. Policy Workshop*, IEEE Press, 2003, pp. 93–98.
4. I. Horrocks et al., *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, ver. 0.5, DAML, 19 Nov. 2003; www.daml.org/2003/11/swrl.
5. N. Damianou et al., "The Ponder Policy Specification Language," *Proc. Policy 2001: Workshop Policies for Distributed Systems and Networks*, LNCS 1995, Springer-Verlag, 2001, pp. 18–39.
6. E. Lupu and M. Sloman, "A Policy Based Role Object Model," *Proc. 1st Enterprise Distributed Object Computing Conf.* (EDOC 97), IEEE CS Press, 1997, pp. 36–47.
7. E. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Trans. Software Eng.*, vol. 25, no. 6, 1999, pp. 852–869.
8. M. Sloman, "Policy Driven Management for Distributed Systems," *J. Network and Systems Management*, vol. 2, no. 4, 1994, pp. 333–360.

it receives as input. A requester that values privacy might see this as an important requirement.

You can interpret a Web Service's privacy policies as an obligation and contract. For example, if after invocation, a service does provide a requester's details to a telemarketer, the person represented by the requester could take legal action against the service on the basis of the policy. As financial transactions become more common among Web Services and as Web Services start dealing with confidential information (such as names, addresses, social security numbers (SSNs), credit cards, and telephone numbers), more people will expect the enforcement of privacy policies.

Authorization policies constrain the provider to accept requests for service only from certain clients. For example, a service's authorization policy could state that a requester must act on behalf of a person who belongs to a certain organizational group and can prove membership with a digital certificate. Similarly, the requester could limit invocation to selected providers.

A motivating example

Consider a scenario in which a scientist is looking for an online computing service for her experimental data. Her privacy policy requires that any personal information provided to the service (such as name or SSN) stay confidential. So, she's only looking for Web Services that accept encrypted data and that don't release personal information to other services or agents.

The scientist finds a Web Service that can perform the necessary data computations. The service's authorization policy says that it allows access only to members of certain, selected organizations and that the scientist's registration must be authenticated.

In this article, we'll approach the formalization and processing of these privacy and authentication policies on two abstraction levels. On a more abstract level, we provide ontologies to annotate Web Service input and output parameters with security characteristics that state whether these parameters are encrypted or digitally signed, and we rely on Rei to formalize the privacy and authorization policies.

On a more concrete level, selecting Web Services that satisfy the requester's policies will be part of an extension of the OWL-S matchmaking process. Furthermore, cryptographic mechanisms such as encrypting or signing messages are enforced via integration into the OWL-S VM, a generic processor for the OWL-S process model and tool for automatic invocation of OWL services.

OWL-S markup

OWL-S is a set of ontologies that describes Web Services with the help of three modules: a *profile* that provides a general description of the Web Service, a *process model* that describes how the Web Service performs its tasks and the Web Service interaction protocol, and the *grounding* that specifies how the atomic processes in the process model map onto WSDL (Web Services Description Language)[5] representations.

Information exchanged between the Web Service and its clients is controlled by the inputs and outputs (*I/O parameters*) defined

```
<foaf:Person rdf:ID="MarySmith">
    <foaf:name xml:lang="en">
        Mary Smith</foaf:name>
    <foaf:title>Dr.</foaf:title>
</foaf:Person>
```

(a)

```
<process:hasInput rdf:ID="PersonInf">
<process:parameterType rdf:resource=
    "EncPersonInfObj"/>
</process:hasInput>
where
<Class rdf:ID="EncPersonInfObj">
    <SubClassOf rdf:resource="#EncInfObject"/>
    <Restriction>
        <onProperty rdf:resource="baseObject"/>
        <allValuesFrom rdf:resource=
            "&foaf;Person"/>
    </Restriction>
</Class>
```

(b)

```
<process:hasInput rdf:ID="RegInf">
<process:parameterType rdf:resource=
    "SigRegInfObj"/>
</process:hasInput>
<Class rdf:ID="SigRegInfObj">
    <SubClassOf rdf:resource="#SigInfObject"/>
    <Restriction>
        <onProperty rdf:resource="baseObject"/>
        <allValuesFrom rdf:resource=
            "&foaf;Person"/>
    </Restriction>
</Class>
```

(c)

Figure 1. OWL encoding for (a) an instance Person using FOAF; (b) an OWL-S description of an input parameter with encryption of values; and (c) an authenticated sign-in requirement.

in the profile and process model. To support security and privacy information, we must provide a way to encrypt I/O parameters. Because encrypted data is just a byte string, it doesn't reveal its internal value or structure. So, we suggest a semantic markup that specifies the security characteristics of Web Services' I/O parameters, keeping information about the data's structure but without revealing its value. (You can find a basic ontology to handle the cryptographic details of Web Services I/O parameters at www.csl.sri.com/users/denker/owl-sec/infObj.owl.) Requestors or matchmaking services can use this metainformation for service selection.

To capture encrypted or signed I/O data, we define an InfObj class (information object) and subclasses EncInfObj (encrypted information object) and SigInfObj (signed information object). We'll use the InfObj class as a range for I/O parameters of OWL-S services. Information objects have a baseObject that describes the type or structure of the information that's encoded in it. For example, the base object of an I/O parameter of class EncInfObj might be a class such as SSN. This property provides knowledge about the data exchanged and could be used to determine whether a service parameter fits a client's requirements or whether two Web Services' I/O parameters match. Furthermore, an information object could have a property cryptoAlgUsed to indicate the specific cryptographic algorithm used for signing or encrypting data.

This basic ontology suffices to describe the cryptographic details necessary for our example. First, we'll look at service discovery and selection. Then, we'll proceed to service invocation.

We use a matchmaker to find a data computation service that satisfies the scientist's functional requirements (such as the type of data and turnaround time). (In this article, we omit the details about functional requirements and focus on the client and service security-related requirements.)

Privacy

Figure 1a shows a partial instance definition of class Person to describe our scientist using the Friend of a Friend (FOAF) ontology (see http://xmlns.com/foaf/0.1) that supports the description of people. Because the scientist doesn't want to reveal personal information to everyone, she looks for Web Services that accept encrypted personal information. To find such a Web Service, the scientist looks for OWL-S service descriptions containing input descriptions in the process:hasInput property that are consistent with the input parameter shown in Figure 1b.

Authorization

Assume that our data computation service grants authorization to the scientist based on the following:

- The scientist must belong to a certain group of selected organizations.
- The scientist must go through authentication to register with the service.

We'll treat the first condition as a Rei policy. The service expresses the second condition as a requirement in which the scientist must register personal information in a verifiable way to avoid impersonation attacks. Figure 1c shows how the Web Service could express its authenticated sign-in requirement. Because the client's policy can vary from the Web Service's policy, the client's requirements might not align with a Web Service's requirements. In our example, the client required encrypted personal information as input for the service, and the service required the client to sign the information. Nevertheless, a matchmaker can deduce from our markup that both the client and Web Service want personal information submitted.

Policy representation and reasoning

We propose to integrate expressive policies relating to several security aspects, including authorization and privacy in Semantic Web Services. Policies are useful primarily during the discovery phase and for forming contracts.

Representing policies in Rei

Rei is an RDF Schema-based language for policy specification. It's modeled on deontic concepts of rights, prohibitions, obligations, and dispensations. These constructs have four attributes: *actor*, *action*, *provision*, and *constraint*. Constraint specifies conditions over the actor, action, and any other context entity that must be true at invocation, whereas provision describes conditions that should be true after invocation. Provisions are the actor's obligations. These basic constructs let us describe different kinds of policies, including authorization, privacy, and confidentiality.

We believe that in distributed environments such as those enabled by Semantic Web Services, the potential of conflicts between policies will be high as there will be several policies acting on a service. To enable dynamic conflict resolution, Rei also includes *metapolicy specifications*, namely setting the modality preference (negative over positive or vice versa) or stating the priority between

```
<!— Rei variables used —>
<entity:Variable rdf:ID="ProviderVar"/>
<entity:Variable rdf:ID="ProviderProject"/>
<entity:Variable rdf:ID="RequesterVar"/>

<!— Find provider of service —>
<constraint:SimpleConstraint rdf:ID="FindProviderOfService"
    constraint:subject="&dcs;profile"
    constraint:predicate="&process;contactInformation"
    constraint:object="#ProviderVar"/>

<!— Get Provider's project —>
<constraint:SimpleConstraint rdf:ID="GetProviderProject"
    constraint:subject="#ProviderVar"
    constraint:predicate="&foaf;currentProject"
    constraint:object="#ProviderProject"/>

<!—Is Requester in the same project as Provider —>
<constraint:SimpleConstraint rdf:ID="SameProjectAsProvider"
    constraint:subject="#RequesterVar"
    constraint:predicate="&foaf;currentProject"
    constraint:object="#ProviderProject"/>

<!— combine first two constraints —>
<constraint:And rdf:ID="FindProviderAndGetProject"
    constraint:first="#FindProviderOfService"
    constraint:second="#GetProviderProject"/>

<!— combine remaining constraint —>
<constraint:And rdf:ID="IsRequesterInSameProjectAsProvider"
    constraint:first="#FindProviderAndGetProject"
    constraint:second="#SameProjectAsProvider"/>

<!— permission to use data computation service —>
<deontic:Permission rdf:ID="ServicePermission">
    <deontic:actor rdf:resource="#RequesterVar"/>
    <deontic:action rdf:resource="&dcs;service"/>
    <deontic:constraint
rdf:resource="#IsRequesterInSameProjectAsProvider"/>
</deontic:Permission>

<sws:AuthorizationPolicy rdf:ID="AuthPolicy1">
    <policy:grants rdf:resource="ServicePermission"/>
 </sws:AuthorizationPolicy>
```

Figure 2. A section of authorization policy specified in Rei.

```
<!— Get process associated with service —>
<constraint:SimpleConstraint rdf:ID="GetProcess"
    constraint:subject="#ServiceVar"
    constraint:predicate="&service;describedBy"
    constraint:object="#ProcessVar"/>

<!— Get output value of process —>
<constraint:SimpleConstraint rdf:ID="GetOutputValue"
    constraint:subject="#ProcessVar"
    constraint:predicate="&process;Output"
    constraint:object="#OutputVar"/>

<!— Parameter type is Person described in foaf —>
<constraint:SimpleConstraint rdf:ID="IsPersonInfo"
    constraint:subject="#OutputVar"
    constraint:predicate="&process;parameterType"
    constraint:object="&foaf:Person"/>

<!— constraints combined —>
<constraint:And rdf:ID="GetProcessAndOutput"
    constraint:first="#GetProcess"
    constraint:second="#GetOutputValue"/>

<constraint:And rdf:ID="IsOutputPersonInfo"
    constraint:first="#GetProcessAndOutput"
    constraint:second="#IsPersonInfo"/>

<deontic:Prohibition rdf:ID="PrivacyRestriction1">
    <deontic:action rdf:resource="#ServiceVar"/>
    <deontic:constraint rdf:resource="#IsOutputPersonInfo"/>
</deontic:Prohibition>
```

Figure 3. A privacy policy specified in Rei.

rules within a policy or between policies themselves.

The Rei reasoning engine interprets and reasons over Rei policies, domain information, and context and answers queries about the current permissions and obligations of entities in the environment. It can answer several types of queries including,

- Does X have permission to perform Y on resource Z?
- What are X's current obligations?
- What actions can X perform on resource Z?
- What are all of X's permissions in the current policy domain?
- Under what conditions does X have permission to perform Y on resource Z?

While answering these queries, the Rei engine takes into account speech acts and tries to resolve any conflicts it might find using the defined metapolicies.

The class Policy is at the root of the Rei ontology. Furthermore, Rei defines three subclasses of Policy, PrivacyPolicy, AuthorizationPolicy, and ConfidentialityPolicy, to specify the different types of policies we can support. In our implementation, we relate the class Policy with the OWL-S ontology by defining a new OWL-S description property, called policyEnforced, of which Policy is the range (see www.csee.umbc.edu/~lkagal1/rei/examples/sws-sec/swspolicy.owl).

For example, we can define in Rei an authorization policy such as (in natural language) "Permit everyone to access the data computation service who is in the same group as the provider of the service." To specify this policy, we exploit the OWL-S property contactInformation, which we specialize to have the range foaf:Agent. We can use this property to describe the service provider. We assume that the OWL-S description of the data computation service exists at some namespace http://www.somenamespace.com/dcs. Moreover, we assume that information

```
<!— Parameter type is Person encrypted as per
inf ontology —>
<constraint:SimpleConstraint rdf:ID=
        "IsPersonInfoEnc"
    constraint:subject="#OutputVar"
    constraint:predicate=
        "&process;parameterType"
    constraint:object="&inf;EncPersonInfObj"/>

<!— constraints combined using IsOutputPerson-
Info constraint described earlier —>
<constraint:And rdf:ID="IsOutputEncPerson"
    constraint:first="#IsOutputPersonInfo"
    constraint:second="#IsPersonInfoEnc"/>

<deontic:Permission rdf:ID="PrivacyRestriction2">
    <deontic:action rdf:resource="#ServiceVar"/>
    <deontic:constraint rdf:resource=
        "#IsOutputEncPerson"/>
</deontic:Permission>

<sws:PrivacyPolicy rdf:ID="PrivacyPolicy1">
    <policy:grants rdf:resource=
        "PrivacyRestriction1"/>
    <policy:grants rdf:resource=
        "PrivacyRestriction2"/>
</sws:PrivacyPolicy>
```

Figure 4. A privacy policy stating that all shared personal information must be encrypted.

```
<profile:Profile rdf:ID=
        "DataComputationServiceProfile">
    <profile:textDescription>
    This data computation service requires
        authorization.
    </profile:textDescription>
    …
    <policyEnforced:rdf:resource=
        "#AuthPolicy1"/>
</profile:Profile>

(a)

<foaf:Person rdf:ID="MarySmith">
    <foaf:name xml:lang="en">
        Mary Smith</foaf:name>
    <foaf:title>Dr.</foaf:title>
    <policyEnforced: rdf:resource=
        "#PrivacyPolicy1"/>
</foaf:Person>

(b)
```

Figure 5. Annotation of an OWL-S profile with (a) authorization and (b) requester policies.

exists about the groups the scientist belongs to as well as information about the groups to which the service provider belongs. See Figure 2 for a section of the authorization policy specified in Rei.

However, a requester might have a privacy policy of never sharing personal information. Figure 3 shows how you could express this in Rei. Specifically, this privacy policy assumes that the FOAF ontology concepts specify all of the scientist's personal information. The policy prohibits any service that has as output personal information described using FOAF Person. The privacy policy acts as a template for allowed or prohibited services based on output parameters. Additionally, the requester might want to specify that any personal information, if shared, must be encrypted (see Figure 4).

Finally, Rei provides a metapolicy prioritization mechanism to resolve policy conflicts. So, a requester could state, for example, that PrivacyRestriction2 holds priority over PrivacyRestriction1, ensuring that services meeting PrivacyRestriction2 are checked first.

Extending OWL-S with policies

All three modules of OWL descriptions need security information: the profile is where you specify the Web Service security requirements for discovery, and the process model and grounding need a specification of the security requirements for invocation and messages exchanged between the Web Service and its requester. No explicit place for security policies exists in OWL-S, but you can naturally link to the profile because policies specify the Web Service's general properties rather than properties that are specific to any process.

Based on our earlier work,[1] we propose that policies are an extension of services' security requirements and suggest adding a property called policyEnforced, defined as a subproperty of securityRequirement (see www.csl.sri.com/~denker/owl-sec/serviceSecurity.owl). PolicyEnforced describes the different policies that must be enforced for the service to execute correctly.

Figure 5a shows how we annotate a Web Service requiring the authorization policy shown in Figure 2.

Similarly, we envision annotations of requesters' policies. In earlier work, we suggested a property securityRequirement with domain Agent, a general class for clients and requester.[1] Property policyEnforced is also a subproperty of an agent's securityRequirement, and we define the foaf:Person class to be a subclass of Agent. So, in Figure 5b we show how we define a scientist who requires that her personal information be transmitted as encrypted data and that it never appear as a service's output.

Using policies to select providers

During the discovery process, the requester must select the best provider. To do this, the requester must verify the compatibility of its policies with the provider's. In this article, we aim to integrate Rei reasoning on policies within the Matchmaker, a capability-based matching engine.[2] To begin, we present the algorithm for privacy constraints:

1. The Matchmaker fetches the OWL-S description of a Web Service that matches the requester's functional requirements.
2. The Matchmaker does the capability matching to extract the Web Services that perform the requested task.
3. The Matchmaker retrieves the requester's privacy policy and extracts the privacy policies from the provider's profile.
4. The Matchmaker sends the OWL-S description and the privacy policies to the Rei reasoner.
5. As the privacy policy defines the prohibited service templates, the Rei reasoner verifies that the matched service is not prohibited. It checks that the service doesn't have as output any information that the client wants to keep private. It also checks that the provider and requester's privacy policies don't contradict each other.
6. If a privacy policy isn't satisfied, the Rei reasoner returns false and the Matchmaker continues to check the next service for privacy compatibility. Otherwise, the Rei reasoner returns true and the Matchmaker returns this service to the client.

Similarly, we present the algorithm for authorization policies:

1. After matching capabilities, the Matchmaker extracts the precondition of the

service that is of type AuthorizationPolicy.
2. It gathers all relevant information about the user and sends this and the authorization policy to the Rei reasoner.
3. If the Rei reasoner returns true, the authorization policy is satisfied and the Matchmaker can return the service to the client. Otherwise, the Matchmaker continues checking the next service for authorization compatibility.

Verifying policy adherence

You can declare policies in the profile, but they should be enforced in the process model that's responsible for the provider and requester interactions. Furthermore, the grounding module provides a mapping from the process model to the messaging specification, and specifically to WSDL and SOAP (Simple Object Access Protocol).

The emerging specifications for Web Services security assume that message security is specified at the WSDL and SOAP levels.[6] If the requester wants to check whether the policies will be enforced in the interaction, it must verify the constraints placed by the provider on message passing.

If the requester wants to verify that the provider adheres to the published policies, it must analyze all specifications for the message passing. The requester also needs to do this because the provider might not expose its policies completely, but it could compile some aspects directly in the interaction specifications.

The following algorithm is a first attempt to enable the requester to verify the provider's adherence to policies:

1. The requester gathers the process model, grounding, and WSDL and SOAP specifications from the provider and its own and the provider's policies.
2. The requester uses the provider's process model, grounding, and WSDL and SOAP specifications to detect what encryption is adopted for the different types of information.
3. The reasoner verifies that
   a. The requester's policies are satisfied
   b. The provider enforces its own policies
4. If the first test fails, the requester doesn't use the provider. If the second fails, the requester makes its own decisions about using the provider.

Steps 1 and 2 are achieved by exploiting the grounding and WSDL, which describes how this information is encoded in the message. In the future, we envision implementing Step 3 of this algorithm in the Rei reasoner. If the results of the reasoning about policies aren't consistent with the requester's policies, the requester knows that it will incur a violation if it pursues the interaction. If the reasoning reveals an inconsistency between the policies specified and the actual interaction management, the requester may decide whether or not to select the provider depending on whether it can satisfy the additional requirements and its own judgment on the provider's failure.

In general, it behooves the provider to be explicit and honest about its policies. If a provider isn't honest and it specifies a policy that it doesn't enforce, it loses all the requesters that don't want to adhere to the policy and loses the trust of the requesters that realize the policies aren't enforced. Similarly, if the provider doesn't explicitly specify some of its policies, it could interact with requesters that can't deal with those policies and fail in the interaction.

Enforcing privacy and authentication

We mentioned that you can fulfill privacy or authentication through encrypting or signing I/O parameters. We propose to keep the work involved with cryptographic operation transparent to the requester by extending the tool that invokes the Web Service (in our case, the OWL-S VM) with features for encrypting or signing data exchanged between client and server.

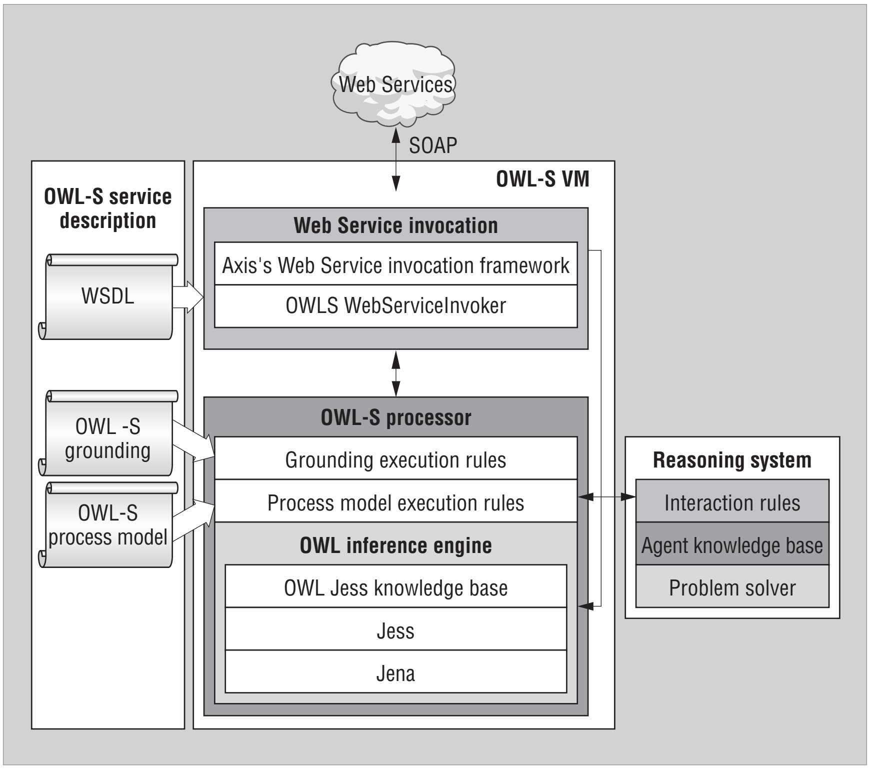


Figure 6. The OWL-S Web Service architecture.

The OWL-S VM[3] implements a general purpose Web Service client that relies on the OWL-S process model and grounding to automate interactions between Web Services, minimizing human intervention. The OWL-S VM architecture, shown in Figure 6, is organized in three columns: on the left are the inputs to the OWL-S VM, specifically the process model, grounding, and WSDL description of a Web Service. The center column describes the OWL-S VM proper, while the box on the right describes the *reasoning system* of the agent that uses the OWL-S VM.

Upon receiving the process model and grounding of a Web Service, the OWL-S VM activates the *OWL-S processor* module, which implements the semantics of OWL-S and OWL, to control the interaction of the Web Service through the execution of the process model. In addition, the Web Service's WSDL description is used to parameterize the *Web Service invocation* module that manages the actual message passing between the OWL-S VM and the Web Service. Whenever the process model requires a message passing between the client and the Web Service, the OWL-S processor asks the reasoning system for the content of the messages to send and

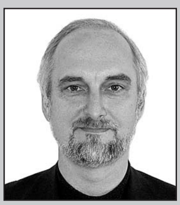
The Authors

Lalana Kagal is a PhD candidate and research assistant with Tim Finin at the Department of Computer Science and Electrical Engineering in the University of Maryland, Baltimore County. She is part of UMBC's eBiquity group (http://research.ebiquity.org), which explores the interactions between mobile computing, multiagent systems, and AI. Her research interests include pervasive computing, application-level security, knowledge representation, agent systems, and the Semantic Web. She received MS degrees in computer science from the University of Poona, India, and from UMBC. Contact her at Computer Science and Electrical Eng., Univ. of Maryland Baltimore County, 1000 Hilltop Cir., Baltimore, MD 21250; lkagal1@cs.umbc.edu; www.cs.umbc.edu/~lkagal1.

Massimo Paolucci is a principal research programmer at Carnegie Mellon University. His research interests include the Semantic Web, Web Services and their relations to multiagent systems, and multiagent planning. He received his MS in computational linguistics from Carnegie Mellon University and his MS in intelligent systems from the University of Pittsburgh. He is a member of the OWL-S Coalition, UDDI (Universal Description, Discovery, and Integration) Technical Committee, Semantic Web Services Initiative architecture committee, and AAAI. Contact him at Robotics Inst., 1604D NSH, Carnegie Mellon Univ., 5000 Forbes Ave., Pittsburgh, PA 15213; paolucci@cs.cmu.edu.

Naveen Srinivasan is a research programmer at Carnegie Mellon University. His research interests include the Semantic Web, Web Services, and tools for multiagent systems. He received his MS in computer science from the University of Maryland, Baltimore County. Contact him at Robotics Inst., 1604D NSH, Carnegie Mellon Univ., 5000 Forbes Ave., Pittsburgh, PA 15213; naveen@cs.cmu.edu.

Grit Denker is a computer scientist in the Computer Science Laboratory at SRI International. Her research interests include formal specification and verification of cryptographic security protocols, security for the Semantic Web and Semantic Web Services, and logic-based approaches for distributed system analysis. She received her PhD in computer science from the Technical University of Braunschweig. Contact her at Computer Science Lab., M/S EL284, SRI Int'l, 333 Ravenswood Ave., Menlo Park, CA 94025; denker@csl.sri.com; www.csl.sri.com/~denker.



Tim Finin is a professor in the Department of Computer Science and Electrical Engineering at the University of Maryland, Baltimore County. His research interests include intelligent interfaces, robotics, software agents, the Semantic Web, and mobile computing. He received his PhD in computer science from the University of Illinois. Contact him at Computer Science and Electrical Eng., Univ. of Maryland Baltimore County, 1000 Hilltop Cir., Baltimore, MD 21250; finin@umbc.edu; http://umbc.edu/~finin.

Katia Sycara is a research professor in the School of Computer Science at Carnegie Mellon University and director of the Advanced Information Technology Laboratory. Her research interests include autonomous agents; planning; learning and coordination of multiple agents in open, uncertain, and dynamic environments; Web Services; and case-based reasoning. She received her PhD in computer science from the Georgia Institute of Technology. She is a member of the OWL-S Coalition and UDDI (Universal Description, Discovery, and Integration) Technical Committee, and is the US chair of the Semantic Web Services Initiative executive committee. She is a fellow of the AAAI, founding editor in chief of the Journal of Autonomous Agents and Multi-Agent Systems, and the 2002 recipient of the ACM Autonomous Agent Research Award. Contact her at Robotics Inst., 1604D NSH, Carnegie Mellon Univ., 5000 Forbes Ave., Pittsburgh, PA 15213; katia@cs.cmu.edu.

then asks the Web Service invocation module to send the message.

We intend to extend the OWL-S VM to enforce authorization and privacy policies. We'll implement the required security transformation on the I/O parameters in the OWL-S VM. So, upon executing an atomic process, the OWL-S VM uses the semantic parameter annotation in the corresponding process model to enforce the privacy and authorization constraints that cryptographic techniques (using encryption and digital signatures) can implement. We use SOAP security annotations to implement the actual message encryption or signing. By using the security mechanisms proposed in this article, Web Services implementing the OWL-S VM are guaranteed to maintain secure communication with their partners.

In the future, we plan to expand our work to address negotiation protocols. Policies that don't match require some form of negotiation. Let's assume that a Web Service requires another Web Service's authentication, but the credential provided doesn't suffice. The service could enter a negotiation phase, following certain communication protocols, to resolve this problem. Furthermore, the policy language's abstraction level or expressiveness also determines the problem's complexity. Consider a policy stating that a client never wants to reveal information that someone can use to deduce his home address. Depending on the information exchanged with the service and additional context information, this could mean that the service could never release the client's phone number because a reverse lookup could compromise the address. This shows that a broad range of policies apply. In the future, we'll also look at more complex policies that address combinations of these security notions and other user-defined policies.

Acknowledgments

DARPA supports this work through the Air Force Research Laboratory under contract F30602-00-C-0168 to SRI, contract F30602-00-2-0592 to Carnegie Mellon University, and contract F30602-97-1-0215 to the University of Maryland, Baltimore County.

References

1. G. Denker et al., "Security for DAML Web Services: Annotation and Matchmaking," *The Semantic Web - ISWC 2003*, LNCS 2870, Springer-Verlag, 2003, pp. 335–350.
2. M. Paolucci et al., "Semantic Matching of Web Services Capabilities," *Proc. 1st Int'l. Semantic Web Conf.*, 2002, pp. 333–347.
3. L. Kagal, T. Finin, and A. Joshi, "A Policy Based Approach to Security on the Semantic Web," *The Semantic Web - ISWC 2003*, LNCS 2870, Springer-Verlag, 2003.
4. M. Paolucci et al., "The DAML-S Virtual Machine," *The Semantic Web - ISWC 2003*, LNCS 2870, Springer-Verlag, 2003, pp. 290–305.
5. E. Christensen et al., "Web Services Description Language (WSDL) 1.1," W3C, 15 Mar. 2001; www.w3.org/TR/wsdl.
6. B. Atkinson et al., "Specification: Web Services Security (WS-Security)," ver. 1.0, 5 Apr. 2002; www-106.ibm.com/developerworks/webservices/library/ws-secure, 2002.