Grounding semantic web services with rules

Dave Lambert and John Domingue

Knowledge Media Institute, The Open University, Milton Keynes, United Kingdom
{d.j.lambert,j.b.domingue}@open.ac.uk

Abstract. Semantic web services achieve effects in the world through web services, so the connection to those services—the grounding—is of paramount importance. The established technique is to use XML-based translations between ontologies and the SOAP message formats of the services, but these mappings cannot address the growing number of non-SOAP services, and step outside the ontological world to describe the mapping. We present an approach which draws the service's interface into the ontology: we define ontology objects which represent the whole HTTP message, and use backward-chaining rules to translate between semantic service invocation instances and the HTTP messages passed to and from the service. We present a case study using Amazon's popular Simple Storage Service.

1 Introduction

Semantic web services are implemented on top of conventional, syntactic web services. The connection between the two—the *grounding*—must enable an actual web service to be invoked based on the content of a semantic description of a service invocation. The conventional approach taken by all the major semantic services frameworks [1–3] is to use an XML mapping language to translate between the two. This naturally doesn't work for non-XML services, such as many of the growing number of RESTful services [4] (section 2). Moreover, XML mapping requires the service engineer to use an use another language in addition to the knowledge representation language used for describing the ontologies, and to consider the XML serialisation of their ontology: three languages instead of one.

We present an alternative which models the target service's input and output (HTTP messages) as ontological objects, and uses backward chaining rules to direct the translations between the semantic service invocation and the low-level data formats required by the service (section 3). As a case study, we take the RESTful version of Amazon's commercial web-storage service, where Amazon's authentication process precludes its description by WSDL or WADL (section 4). We develop the ontologies for describing the HTTP protocol itself, cryptography, and the broker's ontological hooks for the grounding procedure, and put them together in a description of Amazon's service (section 5). The ontologies and grounding have been implmented in the Internet Reasoning Service broker [5]. We compare our scheme to current XML-centric approaches to groundings (section 6).

2 The web service menagerie

Semantic web services rely on normal web services to source and manipulate information and to effect change in the world. This requires that semantic service brokers translate between the world of domain theories, represented in some ontology language, and the on-the-wire data formats and protocols that are the language of the target web services. For semantic web services to be taken seriously by practitioners, groundings must be able to organise communication with a comprehensive subset of real, deployed web services.

The triumvirate of XML, SOAP, and WSDL [6–8] is the basis for the W3C's web services stack, colloquially known as 'WS-*'. As the complexity of the WS-* stack has increased, SOAP's hegemony has ebbed, with many services being offered using lighter weight alternatives. The genuinely simple protocol which inspired SOAP has re-emerged in its own right as XML-RPC [9]. Simultaneously, REST [4] has gained considerable mind-share: according to Amazon's web services evangelist Jeff Barr, around 80% of invocations of Amazon's services are done through the REST interface[1]. Yahoo! does not provide a SOAP interface at all, and has no intention of adding one[2]. Flickr, a popular photo-sharing website, provides its API in all three web service flavours: SOAP, XML-RPC, and RESTful. The SOAP interface does not have a WSDL description, and none of the the third-party bindings[3] for the most popular languages target the SOAP variant:

Flickr Binding	Language	API	Flickr Binding	Language	API
Flickcurl	C	REST	Flickr-Upload	Perl	REST
flickrj	Java	REST	phpFlickr	PHP	REST
jickr	Java	REST	flickr.py	Python	REST
FlickrNet	.NET	REST	flickr-ruby	Ruby	REST
Flickr-API	Perl	REST	rflickr	Ruby	XML-RPC

2.1 Invoking syntactic web services

Before considering how to connect the semantic invocation with the actual service, we should first look at how the various service types are invoked. In SOAP, the connection proceeds at two levels: the HTTP connection, and the XML message. The HTTP request must include the method (always POST), along with the URL of the service, and an HTTP header named `SOAPAction`, which the server uses to dispatch internally. With SOAP, most of the information required for service invocation is encoded in an XML message which comprises the content of an HTTP POST request[4]. This leads to the conception of groundings as simply translations at the XML level from some ontological representation to an XML

[1] `http://www.jeff-barr.com/?p=96`
[2] `http://developer.yahoo.com/faq/#soap`
[3] `http://www.flickr.com/services/api/`
[4] SOAP abstracts away from the transport, but in practise it is used almost exclusively with HTTP, and it is highly questionable whether a system not using HTTP can legitimately be called a 'Web' service [10].

serialisation to be passed to the service, where the URL is a fixed string, identified indirectly through the WSDL. The URL and `SOAPAction` values are dealt with at the HTTP level.

```
POST http://www.world-weather.org/soap
SOAPAction: WeatherReport

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:w="http://www.weather.ex/soap/">
  <soap:Body>
    <w:GetWeatherReport>
      <w:Country>Italy</w:Country><w:City>Rome</w:City>
    </w:GetWeatherReport>
  </soap:Body>
</soap:Envelope>
```

The XML-RPC equivalent is very similar, albeit with no `SOAPAction` header, nor the large number of XML namespace declarations. Unlike SOAP, XML-RPC relies on HTTP as more than a transport protocol. For example, authentication is handled at the HTTP level, so the `Authorization` header must be accessible to the grounding.

Representational State Transfer, better known as REST [4], emphasises resources over operations. It is most often implemented on top of HTTP, where it aligns web service operations with the 'methods' of the HTTP protocol. This means that information requests are performed using `GET`, changes of state are made with `POST`, while `PUT` and `DELETE` are used to manage the data stored at the specified URI. In practise, this means that many operations are performed simply by encoding an operation in a URI and having the client request it:

```
GET http://world-weather.org/forecast/italy/rome/20081216 HTTP/1.1
```

Note that the URL can be an arbitrarily complex amalgam of service's 'name', and some of the service parameters—country and city names, and forecast date. The returned content need not be XML: in this case, it might be plain text description of the weather, or a graphic illustration of the weather over Rome. The correct use of HTTP headers such as `Content-Type` and `Date` become essential. With RESTful and XML-RPC services, WSDL and XML schema descriptions are rarely, if ever, furnished by the service provider.

3 Approach

Given that SOAP is no longer the sole deployment route for web services, we need to reconsider the desiderata for a grounding scheme. For us, we needed a grounding which could address REST services, but which would also support SOAP and XML-RPC in the same framework. This led to the following aims:

1. Target HTTP, not SOAP.
2. Be agnostic about content type: HTTP carries XML, but also images, sound, and other data formats.
3. Do not require third-party cooperation.

4. Stay within the ontology language.

Principle 1 enables us to support the three web services approaches today, and gives us some degree of future-proofing. Principle 2 allows us to describe, for example, services that return images. Principle 3 frees the semantic web services description from the service provider. If a service's WSDL description is not available, or incorrect, we should not be prevented from creating a grounding if we can otherwise discover the interface, through reverse engineering or reading documentation. Nor should we expect the provider to link to our semantic description of their service: it is our description, and they have no obligation to trust or advertise it. Principle 4 is just the application of Ockham's razor: the fewer languages the engineer must deal with, the better. But it also retains the possibility of reasoning over the grounding, for instance in data mediation. More philosophically, if we are truly convinced of the power of ontologies to model such things, it seems odd to avoid their use at this point.

This thought leads to our approach: we model, in a series of ontologies, the various protocols and encodings needed to invoke web services. Each facet of the process is represented by its own ontology: a pure HTTP ontology; a cryptography ontology; and the IRS specific ontology which allows us to map from service invocations to data on the wire. While our ontologies were developed for the purpose of supporting groundings, they could in principle be general purpose ontologies developed for other uses in the respective domains. To invoke a web service, a WSMO broker follows this procedure:

1. A user invokes a semantic services goal by calling the broker with a goal name and the parameters.
2. After some processing by mediators, a web service invocation instance is created. The invocation object holds the service name, input parameters, and slots to hold the return values from the web service.
3. A rule from the service's ontology is called to create an HTTP message object based on the service invocation object, with its various slots' values set to reflect the parameters from the the service invocation object.
4. The HTTP message is passed to the broker, which then turns the HTTP object directly into a request on the network.

The process happens in reverse when the service replies. The stage addressed in this paper is step 3. The two rules we call `lower` and `lift`, which respectively 'lower' and 'lift' the service request object to an implementation level and back. Concretely, the two rule heads are:

```
(lower ?service-type ?service-instance ?http-request)
(lift ?service-type ?http-response ?service-instance)
```

The first argument, `?service-type`, allows each service to have its own implementation, dispatched on the value of the `?service-type` argument. The `lower` rule is a series of subgoals whose successful fulfilment leads to the instantiation of `?http-request`, which can then be interpreted by the broker to call the web service. When a response is received from the server, the `lift` rule runs on the same `?service-instance`, and the newly returned `?http-response`, modifying the original `?service-instance` frame to record the return value.

We implemented this approach in the Internet Reasoning Service (IRS) [5]. The IRS is based on the Web Service Modelling Ontology (WSMO) [11], and uses OCML [12] for knowledge representation and reasoning. OCML is a frame language with procedural attachment, and is comparable in expressiveness to the Ontolingua and Loom languages. In our implementation, we use OCML's ability to define the operational semantics of ontological components directly in Lisp.

4 Amazon's Simple Storage Service

Before looking at the ontologies in detail, we examine our motivating task: the targeting of a real-world web service in the context of our work in the Living Human Digital Library project (LHDL) [13]. LHDL is building a library of data about the human musculoskeletal system, along with web services to visualise and manipulate it. The medical imaging and motion data, from MRI scans, dissection and gait analysis, is stored in repositories accessed and managed by XML-RPC web services, and computational services accessed through SOAP. We use semantics to manage and integrate them with external services.

A key aim of LHDL is to enable the storing and sharing of the research data, but the data files commonly range in size from hundreds of megabytes to several gigabytes. It would be beneficial if we could outsource the storage and transfer of this data, and several major web services providers have recently begun offering such facilities. We decided to use Amazon's 'simple storage service' (S3) as our test-bed. Amazon's S3 is a commercial service, with charging being a function of storage and data transfer (in Gigabytes per month), and number of HTTP operations.

Storage at S3 is organised by 'buckets' and 'keys', which are analogous to directories and files in file systems. Buckets live in a global namespace, and users can create new ones provided the name is currently unused. Within a bucket, the owner has control over the keys and the objects they name, as well as access control for other users. A user is identified by an 'access key', and authorised by means of a 'secret key'. Buckets are created by executing a `PUT` to the bucket's corresponding URL. The following HTTP request instructs S3 to store the string '`Hello, world!`' in the object with key `hello` in the bucket `lhdl`:

```
PUT /lhdl/hello HTTP/1.1
Host: s3.amazonaws.com
Authorization: AWS 5EB1K7DR13JHNF92JV23:CwuLdze7uX4LLP+KGMe/5htHb20=
Date: Fri, 26 Sep 2008 21:33:39 GMT
Content-Type: text/plain
Content-Length: 13

Hello, world!
```

Note the `Authorization` header. The content of this field is specified by Amazon to be the pair of the user's access key, and a digital signature (using the HMAC-SHA1 hash) of the HTTP request itself, created with the user's secret key. Figure 1 shows the grammar provided by Amazon specifying the signature's content. Since a request's `Date` header must agree with that on the Amazon servers (within a narrow margin of error), the above request would not work now. Note that, because the signed string includes the date, the authorisation value cannot be generated before the date is decided. Since the date is only decided at the HTTP layer, they must both be done as part of the same process in the broker, and with one control procedure being responsible. These features make S3's interface impossible to capture in WSDL [8] or WADL [14], but straightforward to ontologise using rules.

```
Authorization =
    "AWS" + " " + AWSAccessKeyId + ":" + Signature;

Signature =
    Base64(HMAC-SHA1(UTF-8-Encoding-Of(StringToSign)));

StringToSign =
    HTTP-Verb + "\n" +
    Content-MD5 + "\n" +
    Content-Type + "\n" +
    Date + "\n" +
    CanonicalizedAmzHeaders +
    CanonicalizedResource;

CanonicalizedResource =
    [ "/" + Bucket ] +
    <HTTP-Request-URI, from the protocol name up to the query string>
```

Fig. 1. A grammar for the HTTP `Authorization` header for invoking an Amazon S3 REST service (From Amazon web services documentation.)

5 Ontologies

Having seen our overall approach, and our target web service, we are now in a position to present the ontologies: one each for the IRS broker, HTTP protocol, cryptographic operations, and the Amazon service description itself. In OCML, namespaced symbols are written `#_namespace:localName`. A symbol `#_localName` without a namespace refers to the current namespace. In this paper, the namespace prefixes map to ontologies as follows:

`irs` → `irs`	The broker's grounding interface
`rfc2616` → `rfc2616`	HTTP protocol
`crypt` → `cryptography`	Cryptography
`store` → `storage`	Amazon S3 services

The IRS ontology contains the `lift` and `lower` rules. Our HTTP ontology is named `rfc2616`, for the IETF standard for HTTP/1.1 [7]. We modelled only a subset of HTTP, but it is sufficient to perform invocations described here. A message is represented by the class `#_rfc2616:http-message`. A message has a method, URL, and set of headers. To provide security, the S3 services require the cryptographic signing of the HTTP requests using the HMAC-SHA1 hash. Figure 2 shows the cryptography ontology developed to handle this. The SHA1 hash function is represented by the `hmac1-sha1` function.

```
(def-function #_hmac-sha1 (data key)
  "Return the SHA-1 digest of DATA, using KEY."
  :lisp-fun #'%hmac-sha1)

(defun %hmac-sha1 (data key)
  "Compute an RFC 2104 HMAC-SHA1 digest on DATA using KEY."
  (let ((hmac (ironclad:make-hmac
               (ironclad:ascii-string-to-byte-array key) :sha1)))
    (ironclad:update-hmac hmac (ironclad:ascii-string-to-byte-array data))
    (ironclad:hmac-digest hmac)))

(def-function #_encode-base64 (octets)
  "Encode OCTETS into base 64 ASCII."
  :lisp-fun #'base64:usb8-array-to-base64-string)
```

Fig. 2. Cryptography ontology fragment.
Here we use OCML's ability to define, within the ontology itself, entry points to functionality implemented in the host language, Lisp. This is only only a convenience: our approach requires only that the broker can somehow operationalise the low-level elements of the ontology. The `encode-base64` function actually lives in the MIME ontology.

Figure 3 shows the key parts of the `storage` ontology which holds the description of the S3 service for getting objects from the server. The class `amazon-get-object-service` models the web service, where it is characterised by input and output roles. Those are typed by the five domain classes which relate to the S3 concepts of buckets, object keys, access keys, and secret keys, with the latter two being tied together in an account structure. The description shown omits several WSMO elements related to choreographies, capabilities and so on which are not relevant here. Figure 3 contains the lifting and lowering rules. Note that the `lift-for-get-object` recovers not just the content of the HTTP message, but the `Content-Type` header, too. For S3, this is important because arbitrary data types can be stored, and S3 records the type specified at upload time and returns when the object is downloaded.

The lowering rule itself is natural and straightforward. It begins with the creation of a new frame instance, the `?http-request`, and the date and host fields being set up. The target bucket and key objects are recovered from the service invocation with the `slot-value` relation, and are used to create the URL. The HTTP object's URL and method slots are set appropriately. Finally, the keys are recovered from the account, and used to sign the request. The signature itself is controlled by the rule `sign-amazon`, whose structure parallels that of the grammar provided in Amazon's documentation as shown in figure 1.

```
(def-class amazon-get-object-service (web-service)
  ((has-input-role :value has-account :value has-bucket
                   :value has-key :value has-data)
   (has-output-role :value has-content :value has-content-type)
   (has-account :type #_amazon-account)
   (has-bucket :type #_amazon-bucket)
   (has-key :type #_amazon-object-key)
   (has-content :type octets)
   (has-content-type :type string)))


(def-rule #_lower-amazon-get-object
    ((#_irs:lower amazon-get-object-service ?service-instance ?http-request) if
     (= ?http-request (#_rfc2616:new-instance #_rfc2616:http-request))
     (= ?date (#_rfc2616:format-http-time (#_irs:current-time)))
     (#_rfc2616:set-header ?http-request "Date" ?date)
     (= ?host "s3.amazonaws.com")
     (#_rfc2616:set-header ?http-request "Host" ?host)
     (= ?bucket (slot-value ?service-instance 'has-bucket))
     (= ?key (slot-value ?service-instance 'has-key))
     (= ?url (make-string "http://~A/~A/~A" ?host ?bucket ?key))
     (= ?canonical-url (make-string "/~A/~A" ?bucket ?key))
     (#_rfc2616:set-url ?http-request ?url)
     (#_rfc2616:set-method ?http-request "GET")
     (= ?account (#_get-account (slot-value ?service-instance has-account)))
     (#_has-amazon-access-key ?account ?access-key)
     (#_has-amazon-secret-key ?account ?secret-key)
     (#_sign-amazon ?http-request ?canonical-url ?access-key ?secret-key)))

(def-rule #_lift-for-get-object
    ((#_irs:lift get-object-web-service ?http-response ?invocation) if
     (#_rfc2616:get-content ?http-response ?http-content)
     (set-slot-value ?invocation #_hasContent ?http-content)
     (#_rfc2616:header-value ?http-request "Content-Type" ?content-type)
     (set-slot-value ?invocation #_hasContentType ?content-type)))


(def-rule #_sign-amazon
    "Sign the ?HTTP-REQUEST in the manner of Amazon S3."
    ((#_sign-amazon ?http-request ?canonical-url ?access-key ?secret-key) if
     (= ?signature
        (#_compute-signature ?http-request ?canonical-url ?secret-key))
     (#_has-value ?access-key ?access-key-string)
     (= ?signature-header
        (make-string "AWS ~A:~A" ?access-key-string ?signature))
     (#_rfc2616:set-header ?http-request "Authorization" ?signature-header)))

(def-function #_compute-signature (?http-request ?canonical-url ?secret-key)
  "Return the cryptographic signature for ?HTTP-REQUEST for Amazon S3."
  :constraint (and (#_http-request ?http-request)
                   (#_amazon-secret-key ?secret-key))
  :body (the ?signature
          (and (#_has-value ?secret-key ?secret-string)
               (#_rfc2616:has-method ?http-request ?method)
               (#_rfc2616:get-header ?http-request "Date" ?date-header)
               (#_rfc2616:field-value ?date-header ?date)
               (#_rfc2616:get-header ?http-request "Content-Type"
                                     ?content-type-header)
               (#_rfc2616:field-value ?content-type-header ?content-type)
               (= ?to-sign
                  (make-string "~A~%~%~A~%~A~%~A" ?method ?content-type ?date
                               ?canonical-url))
               (= ?signature (#_crypt:encode-base64
                              (#_crypt:hmac-sha1 ?to-sign ?secret-string))))))
```

Fig. 3. The `get-object` service, `lower` and `lift` rules, and digital signature function.

6 Related work

WSDL [8] has been the de facto specification means for web services since the birth of web services. Both OWL-S [1] and WSMO [11] define their groundings by pointing at the WSDL of their targets, but the mapping to the syntactic content of the messages is something of a grey area. The OWL-S WSDL document [15] suggests that OWL-S services should require web services to use an OWL specific encoding in their implementation. The semantic annotation extensions for WSDL—WSDL-S and then SA-WSDL (Semantic Annotations for Web Service Description Language) [3]—make it possible to have WSDL descriptions link to semantic descriptions in various frameworks. SA-WSDL uses mapping schemas to handle the lifting and lowering, but again it is XML specific. The IRS [5] used XPath expressions to generate OCML relations which performed the lifting and lowering. Another WSMO based broker, the Web Services Execution Environment (WSMX) uses service-specific 'adaptors', written in Java, to connect to services.

Although the principle of 'lifting and lowering' the XML serialisation is well established, it does not address aspects of the HTTP protocol like the `Authorization` header. Naturally, XML translation precludes the use of many services that do not use XML. Although WSDL and SOAP are products of the W3C standards, there is significant disquiet amongst developers about their complexity, interoperability, and their lack of Web nature. Personal experience has made us skeptical of the quality of WSDL and XSD descriptions, even, or perhaps especially, machine generated ones. Finally, using an XML mapping scheme like XSLT forces the ontology engineer to leave the semantic realm to work on the groundings, and to consider the domain objects in terms of their XML serialisation.

In contrast, our groundings unify the lifting and lowering with the management of the HTTP protocol, and are declarative and wholly within the ontology language, modulo the small number of operational primitives such as the cryptography functions. Since there are relatively few data encoding and cryptography schemes—many orders of magnitude fewer than there will be web services—it makes sense to embed them in semantic brokers, and make them available to ontology rules. With these hooks in place, we can encode groundings to a large number of important, real-world web services in a unified, ontology-based manner.

7 Conclusion

Semantic web services are about web services as well as semantics. In this paper, we introduced an approach to groundings rooted in ontologies and rules. A set of ontologies for modelling aspects of web service implementation was introduced, allowing us to use the ontological language itself to express the grounding, including the 'lifting and lowering' to string serialisations of data structures. We used these to describe a popular commercial REST web service, Amazon's S3. We have implemented the ontologies discussed, and used them in the IRS. They currently support the automated transfer of file resources in the LHDL project (`http://lhdl.open.ac.uk:8080/irs`).

We see this approach as a useful low-level implementation platform: it is sufficiently powerful to connect to any kind of HTTP service, and yet is fully accessible from the ontological level. It is general enough, for instance, that it could support multi-part MIME messages. We intend to add ontologies for reasoning directly with XML, XML-RPC, and SOAP messages, which currently must be accessed by direct manipulation of the XML string representation. On top of this, we expect to layer generic rules which will handle any services which have WSDL or WADL definitions, as well as semantic approaches like SA-WSDL.

Acknowledgements This research was funded by the Living Human Digital Library project, under European Union grant FP6-026932. We thank Barry Norton and Marta Sabou for helpful discussions.

References

1. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic Markup for Web Services (2004) W3C Member Submission 22 November 2004.
2. Lausen, H., Polleres, A., Roman, D.: Web Service Modeling Ontology (WSMO). Technical report, World Wide Web Consortium (W3C) (June 2005)
3. Farrell, J., Lausen, H.: Semantic Annotations for WSDL and XML Schema. W3C Recommendation, World Wide Web Consortium (W3C) (August 2007)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
5. Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Norton, B., Tanasescu, V., Pedrinaci, C.: IRS-III: A Broker for Semantic Web Services based Applications. In: Proceedings of the 5th International Semantic Web Conference (ISWC2006), Athens, Georgia, USA (2006)
6. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: Simple Object Access Protocol (SOAP) 1.1. Technical report, World Wide Web Consortium (W3C) (May 2000)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol — HTTP/1.1. Technical report, Internet Engineering Task Force (June 1999)
8. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001)
9. Winer, D.: XML-RPC Specification (June 1999) Online at `http://www.xmlrpc.com/spec`.
10. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Inc (2007)
11. Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J.: Enabling Semantic Web Services. Springer (2006)
12. Motta, E.: An Overview of the OCML Modelling Language. In: 8 th Workshop on Knowledge Engineering: Methods & Languages KEML 98. (1998)
13. Viceconti, M., Taddei, F., Van Sint Jan, S., Leardini, A., Clapworthy, G., Galizia, S., Quadrani, P.: Towards the multiscale modelling of musculoskeletal system. (2007)
14. Hadley, M.J.: Web Application Description Language (WADL) (November 2006)
15. Martin, D., Burstein, M., Lassila, O., Paolucci, M., Payne, T., McIlraith, S.: Describing Web Services using OWL-S and WSDL