Integrating heterogeneous web service styles with flexible semantic web services groundings

Conference or Workshop Item

Integrating Heterogeneous Web Service Styles with Flexible Semantic Web Services Groundings

Dave Lambert, Neil Benn, and John Domingue

Knowledge Media Institute, The Open University
Milton Keynes, United Kingdom

Semantic web services are touted as a means to integrate web services inside and outside the enterprise, but while current semantic web service frameworks—including OWL-S [1], SA-WSDL, and WSMO[1] [2]—assume a homogeneous ecosystem of SOAP services and XML serialisations, growing numbers of real services are implemented using XML-RPC and RESTful interfaces, and non-XML serialisations like JSON.[2] Semantic services platforms based on OWL-S and WSMO use XML mapping languages to translate between an XML serialisation of the ontology data and the on-the-wire messages exchanged with the web service, a process referred to as *grounding*. This XML mapping approach suffers from two problems: it cannot address the growing number of non-SOAP, non-XML services being deployed on the Web, and it requires the modeller creating the semantic web service descriptions to work with the serialisation of the service ontology and a syntactic mapping language, in addition to the knowledge representation language used for representing the semantic service ontologies and descriptions.

Our approach draws the service's interface into the ontology: we define ontology objects that 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. This novel approach is based on several principles:

1. Target HTTP, the Web's native protocol: all three Web services approaches in use today, namely SOAP, XML-RPC, and RESTful, use HTTP.
2. Be agnostic about content type: HTTP carries not only XML, but other text serialisations such as JSON, as well as images, sound, and other data.
3. Stay within the ontology language: the fewer languages the service semantics engineer must deal with, the better.

When a semantic broker invokes a web service, it translates an abstract ontological representation of the service invocation into an HTTP message that can be sent to the server. The server's response is, in due course, translated from the HTTP message to an ontological representation of the result. These two processes are respectively known as *lowering* and *lifting*. In our scheme we define two generic rules, which we call `lower` and `lift`:

```
(lower ?serviceType ?serviceInvocation ?httpRequest)
(lift ?serviceType ?httpResponse ?serviceInvocation)
```

[1] The WSMO specification deftly sidesteps the issue of grounding to any kind of Web service, current WSMO implementations are SOAP-centric.

[2] `http://www.json.org/`

Each web service description can define its own version of `lift` and `lower`, which the broker distinguishes by unifying on the `?serviceType` parameter that names the service being invoked. The `?serviceInvocation` represents the abstract invocation message, and `?httpRequest` and `?httpResponse` the ontological representations of the on-the-wire messages. The `lower` rule's successful fulfilment leads to the instantiation of `?httpRequest`, which can then be directly interpreted by the broker to call the web service. When a response is received from the server, the `lift` rule runs on the the newly returned `?httpResponse`, modifying the original `?serviceInvocation` frame to record the return values. We have fully implemented the approach in the Internet Reasoning Service (IRS),[3] a broker based on WSMO and using OCML[4] for its ontology language. In another paper [3] we showed how the approach integrates RESTful and XML-RPC services, and we now demonstrate the same approach cleanly extends to JSON serialisations.

JSON is a simple data format derived from Javascript, and is increasingly popular as a lightweight alternative to XML, particularly in RESTful services. Our lift and lower rules could directly manipulate the string representations of JSON, but this becomes cumbersome, prone to error, and fails to model in any meaningful way the transformations. Instead, we introduce a simple ontologisation of JSON. JSON is built on two main structures: objects and arrays. An object is an unordered set of key/value pairs, whereas an array is a sequence of values. Values are strings, numbers, booleans, other objects or arrays, or `null`. Our JSON ontologisation in OCML introduces a top-level class `Value`, which is then subclassed by `Object` and `Array`. An `Object` is made up of a list of `Members`, where each element of the list is a `Pair` consisting of a `key` (a string) and a `Value`. An `Array` is made up of a list of `Elements`, where each element of the list is a `Value`. Finally, the JSON value-types string, number, and boolean, are matched to OCML's equivalent built-in types, while JSON `null` values are treated as OCML `nil`. The OCML relation `serialiseJson` converts between JSON ontologisations in OCML, and the string containing the serialised JSON. The key definitions are:

```
(def-class Value () ?value
  :sufficient
  (or (String ?value)
      (Number ?value)
      (Boolean ?value)
      (= ?value nil)))

(def-class Object (Value)
  ((members :type Members)))

(def-class Array (Value)
  ((elements :type Elements)))
```

```
(def-class Elements () ?elements
  :iff-def (and (listp ?elements)
                (every ?elements Value)))

(def-class Members () ?members
  :iff-def (and (listp ?members)
                (every ?members Pair)))

(def-class Pair ()
  ((key :type String)
   (value :type Value)))
```

To illustrate how we can invoke, in a uniform way, Web services employing JSON and XML formats, we use two Flickr[5] services: *a*) getting a list of recently changed photos in a user's account (`flickr.photos.recentlyUpdated`), and *b*) getting a list of sizes in which those are available (`flickr.photos.getSizes`). We will ground

[3] `http://technologies.kmi.open.ac.uk/irs/`
[4] `http://technologies.kmi.open.ac.uk/ocml/`
[5] `http://www.flickr.com/services/api/`

the first service in a RESTful way, consuming JSON output, and the second service via XML-RPC, consuming XML output, integrating the services at the ontological level. We begin with lowering rule for the `flickr.photos.recentlyUpdated` service:

```
(def-rule lower-for-photosRecentlyUpdated/RestJson
    ((lower photosRecentlyUpdated/RestJson ?invocation ?http-request) if
     (= ?account (wsmo-role-value ?invocation hasAccount))
     (= ?token (wsmo-role-value ?invocation hasToken))
     (= ?min-date (wsmo-role-value ?invocation hasMinimumDate))
     (hasKey ?account ?apikey)
     (hasValue ?apikey ?apikey-string)
     (hasValue ?token ?token-string)
     (= ?args (new-instance Arguments))
     (addArgument ?args "method" "flickr.photos.recentlyUpdated")
     (addArgument ?args "api_key" ?apikey-string)
     (addArgument ?args "auth_token" ?token-string)
     (addArgument ?args "min_date" ?min-date)
     (addArgument ?args "format" "json")
     (addArgument ?args "nojsoncallback" "1")
     (signArguments rest ?args ?account)
     (argsToRestRequest ?args ?http-request)))
```

The rule is straightforward, with the first half concerned with extracting from the `?invocation` the necessary argument values, and the second half constructing an `Arguments` instance using those values. The rule `signArguments` takes care of signing arguments with a valid Flickr account key, while `argstoRestRequest` takes care of converting the arguments set to URL parameters of the form

```
param1=value1&param2=value2...
```

and sets the `?http-request` fields appropriately (neither rule is shown due to space constraints, but they are relatively simple). The lifting rule is:

```
(def-rule lift-for-photosRecentlyUpdated/RestJson
    ((lift photosRecentlyUpdated/RestJson ?http-response ?invocation) if
     (rfc2616:get-content ?http-response ?http-content)
     (json:serialiseJson ?json ?http-content)
     (json:Object ?json)
     (json:members ?json ?objMembers)
     (member ?photosKeyValuePair ?objMembers)
     (json:key ?photosKeyValuePair "photos")
     (json:value ?photosKeyValuePair ?innerPhotosObject)
     (extractPhotoListFromJson ?innerPhotosObject ?photolist)
     (set-goal-slot-value ?invocation hasPhotoList ?photolist)))
```

In lifting the response, we extract the content from the `?http-response` into an ontological model of the JSON. This extraction is done through the two rules `extractPhotoListFromJson` and `extractPhotoFromJson` defined below:

```
(def-rule extractPhotoListFromJson
    ((extractPhotoListFromJson ?innerPhotosObject ?photolist) if
     (= ?photolist
        (setofall ?photo
            (and (json:Object ?innerPhotosObject)
                 (json:members ?innerPhotosObjectMembers)
                 (member ?photoKeyValuePair ?innerPhotosObjectMembers)
                 (json:key ?photoKeyValuePair "photo")
                 (json:value ?photoKeyValuePair ?photoObject)
                 (extractPhotoFromJson ?photoObject ?photo))))))

(def-rule extractPhotoFromJson
    ((extractPhotoFromJson ?photoObject ?photo) if
     (json:Object ?photoObject)
     (json:members ?photoObject ?photoObjectMembers)
```

```
    (member ?idKeyValuePair ?photoObjectMembers)
    (json:key ?idKeyValuePair "id")
    (json:value ?idKeyValuePair ?id)
    (member ?titleKeyValuePair ?photoObjectMembers)
    (json:key ?titleKeyValuePair "title")
    (json:value ?titleKeyValuePair ?title)
    (= ?slots ((id ?id) (title ?title)))
    (new ?photo Photo ?slots)))
```

With this list of photographs, represented in OCML, we are now ready to invoke the second service, `flickr.photos.getSizes`, using XML-RPC.

```
(def-rule lower-for-photosGetSizes/Xmlrpc
    ((lower photosGetSizes/Xmlrpc ?invocation ?http-request) if
     (argsForPhotosGetSizes ?invocation ?args)
     (= ?account (wsmo-role-value ?invocation hasAccount))
     (signArguments xmlrpc ?args ?account)
     (argsToXmlrpcRequest ?args ?http-request)))
```

This time, the conversion from the invocation object to the argument pairs used by Flickr is done in another rule `argsForPhotosGetSizes` (not shown). The use of a rule for this means we could share the logic between the XML-RPC version shown here, and a REST or SOAP version. The argument set is again signed using `signArguments`, and then passed to the rule `argsToXmlrpcRequest`:

```
(def-rule argsToXmlrpcRequest
    ((argsToXmlrpcRequest ?args ?http-request) if
     (getArgument ?args "method" ?method)
     (= ?nonmethodargs
        (setofall ?member
            (and (hasArgument ?args ?arg)
                 (hasName ?arg ?name)
                 (not (= ?name "method"))
                 (hasValue ?arg ?value)
                 (= ?member (xmlrpc:Member ?name
                                (xmlrpc:String ?value))))))
     (= ?xmlrpc (xmlrpc:MethodCall ?method
                    (xmlrpc:Param (xmlrpc:Struct ?nonmethodargs))))
     (xmlrpc:mapToXml ?xmlrpc ?xmlmodel)
     (xml:serialiseXml ?xmlmodel ?xmlstring)
     (rfc2616:set-content ?http-request ?xmlstring)
     (rfc2616:set-method ?http-request "POST")
     (rfc2616:set-url ?http-request "http://api.flickr.com/services/xmlrpc/")))
```

The `argsToXmlrpcRequest` rule performs a similar function to `argsToRestRequest`, but this time we create an XML-RPC message with a `Struct` to hold the pairs, rather than embedding them in a URL.

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. W3C member submission, World Wide Web Consortium (W3C) (November 2004)
2. Lausen, H., Polleres, A., Roman, D.: Web Service Modeling Ontology (WSMO). W3C member submission, World Wide Web Consortium (W3C) (June 2005)
3. Lambert, D., Domingue, J.: Photorealistic semantic web service groundings: Unifying RESTful and XML-RPC groundings using rules, with an application to Flickr. In: Proceedings of the 4th International Web Rule Symposium (RuleML 2010). (October 2010)