SA-REST: BRINGING THE POWER OF SEMANTICS TO REST-BASED WEB SERVICES

by

JONATHAN DOUGLAS LATHEM

(Under the Direction of Amit Sheth and John Miller)

ABSTRACT

As the number of online applications increase so do sources of structured data repositories in the form of RSS, ATOM and lightweight Web services. These can all be covered by the larger umbrella of REST-based Web services. Many users want to bring together these discrete data to form new data sets that contain parts of the original services. This activity is referred to as building a mashup.

Until now almost all mashups have been built by hand, which requires a great deal of coding. This solution is of course not scalable since more and more sources are coming online. In this document we propose a way to add semantic metadata to REST-based Web services, called SA-REST services, and a way to create semantic mashups, called Smashups, that rely on the annotated services to remove the hand coding of mashups.

INDEX WORDS: Semantic Web Services, REST, SA-REST, SAWSDL, Mashup, SMashup

SA-REST: BRINGING THE POWER OF SEMANTICS TO REST-BASED WEB SERVICES

by

JONATHAN DOUGLAS LATHEM

Bachelors of Science, University of Georgia, 2005

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

Master of Science

ATHENS, GEORGIA

2007

SA-REST: BRINGING THE POWER OF SEMANTICS TO REST-BASED WEB SERVICES

by

JONATHAN LATHEM

Major Professor:	Amit Sheth John Miller
Committee:	David Lowenthal Hamid Arabnia

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2007

DEDICATION

This thesis is dedicated to the giants upon whose shoulders I now stand.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Two of the biggest promises of Web 2.0 are that it is a customizable Web and that it is a read/write Web. That is to say, the user can change what data are displayed and the format of those data while at the same time adding new content. One of the biggest phenomena of Web 2.0 is mashups. A mashup is the creation of a new service from two or more preexisting services. For example, Housing Maps is a mashup that combines descriptions of houses for rent or sale on Craig's List and the Google Maps of their locations to create a nice visualization of Craig's List. Mashups give the user a more integrated experience of the Web.

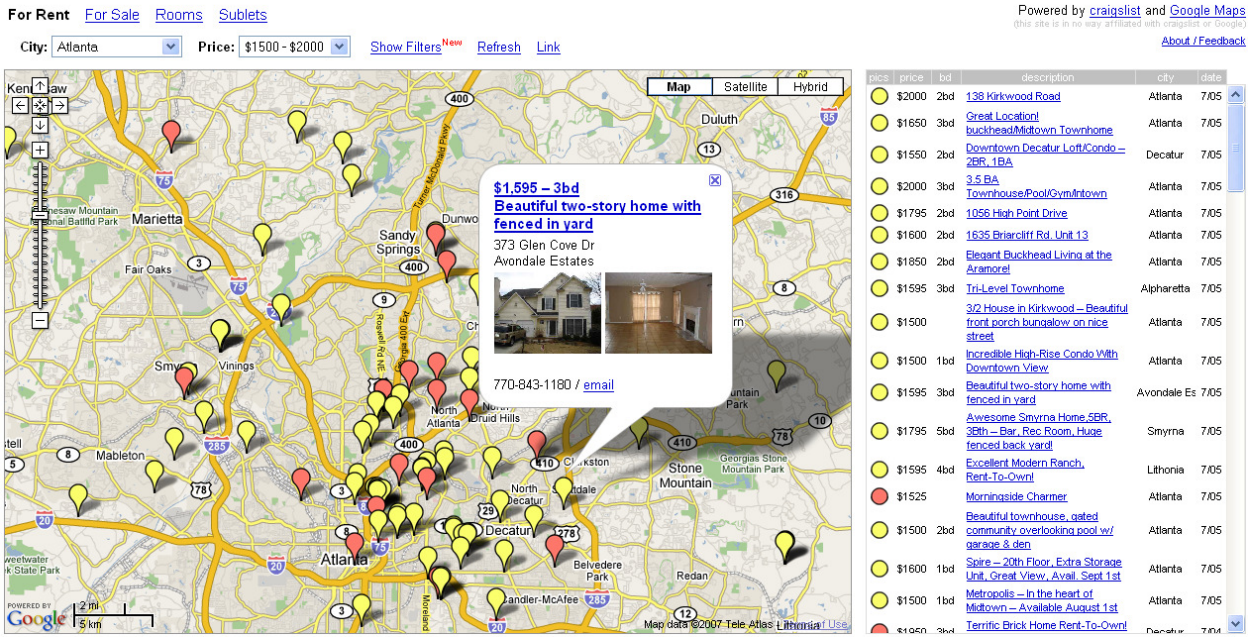

Figure 1: Screen shot of Housing Maps, an example of a mashup involving Craig's List and Google Maps

Mashups are a phenomenon of Web 2.0[26] and not of Web 1.0 partly because of the innovations of RESTful Web services and AJAX (Asynchronous JavaScript API and XML). RESTful Web services are an alternative to the traditional SOAP[1]-based Web services, and their use has significantly outpaced that of the more traditional counterpart. A RESTful Web service is a lightweight service. They differ from SOAP-based services in many ways. The main idea behind a RESTful service is simplicity. There is typically no WSDL[2] as in a SOAP service, and the input and output messages have no standardized format such as the one SOAP envelope provides. Most of the applications provided by the WEB 2.0 community are RESTful services or have APIs to the application that can be in invoked via a RESTful Web service. Because of their simplicity (at least for the basic usage), these services are easer to invoke while using scripting languages, such as javascript, which is the main coding language of the client-side Web. AJAX is a way to programmatically invoke a service or retrieve a resource via client-side code. The popularity of this technology has sparked the revolution of Web 2.0, the creation of RESTful Web services, and ultimately the creation of mashups. Before AJAX was widely used, the most common way for the client-side Web page to communicate with the server was by transitioning to a new page or by reloading the current page with different parameters. The difference between these two services can be seen by comparing classic Yahoo Maps [10] with the new Yahoo Maps [11]. In the new Yahoo Maps site the user can drag the map around; when a part of the map that was not previously visible comes into view, that section is programmatically downloaded and the visual interface updated. The classic Yahoo Maps

[1] SOAP initially stood for Simple Object Access Protocol, and lately also Service Oriented Architecture Protocol, but is now simply SOAP [Wikipedia].
[2] Web Services Description Language (WSDL) is a W3C recommendation/standard for describing Web services in an XML-based syntax. http://www.w3.org/TR/wsdl20/

approach was to display a static map to the user; when the map needed to be scrolled the entire Web page would reload with a new static map present at the scrolled position.

Mashups really embrace the idea of a customizable Web. A user is most likely not going to want to browse Craig's List and then look up the map for each location found on the list. On the other hand, mashups do not embrace the idea of a read/write Web. The reason for that is because of the difficulty for an average user without technical training to create a mashup. A lot of programming typically goes into creating a mashup; a user would need to understand not only how to write code but the APIs of all the services to be included in the mashup. For the typical Web user the task would be time consuming if not impossible. To solve this problem, leading companies are now actively developing tools that can be used to create a mashup and that require little or no programming knowledge. These tools typically facilitate the selection of some number of RESTful Web services or other Web resources and chain them together by piping one service's output into the next service's input while filtering content and making format changes. Three leaders in this field are Yahoo Pipes, Google Mashup Editor, and IBM's QEDWiki. We describe each of these tools in detail in Section 4.

The drawback of these tools is that they are limited in the number of services they can interact with. These tools normally deal with services that are internal to the company where the tool was developed (e.g., Google Mashup Editor can use Google Maps) or services that have standard types of outputs such as RSS or ATOM. A vast number of services therefore cannot be utilized via these tools for the creation of mashups. This limitation is understandable; it would be extremely difficult to work with a service that can take inputs in a nonstandard form and

produce arbitrary outputs without forcing users to handle all the data mediation themselves. It is true that that if one of these companies wanted to add a new service that did not have a standard output or was not an internal service to their tool, it would be possible by making modification to the tool and creating an interface to the new service. However, this solution is not scalable because of the rate at which new services are coming online. The need to change the tool itself also removes the ideal of a customizable Web. A user could not create a specialized service and utilize that service with one of the tools mentioned above.

In this thesis we propose a novel way to add semantic annotations to arbitrary RESTful Web services and then use these annotations to facilitate the mediation and composition of mashups while removing many of the limitations of current tools.

CHAPTER 2

ADDING SEMANTICS TO RESTFUL WEB SERVICES

Adding semantic annotations to a RESTful Web service yields many benefits and alleviates many of the problems associated with RESTful Web services. In this section, we discuss the why, what, and how of adding semantic annotation to RESTful services. We call this idea Semantic Annotation of RESTful Web services, or SA-REST.

There have been a number of efforts to add formal semantics to traditional Web services including OWL-S [7], WSMO [22] and WSDL-S [4]. Driven primarily by the W3C member input of our METEOR-S research group at LSDIS lab and Kno.e.sis Center, and in association with IBM [1], a W3C work group has recently released a candidate recommendation called Semantic Annotation of WSDL or SAWSDL [21]. We have developed SA-REST from many of the ideas that were first presented in WSDL-S and then adapted in SAWSDL. The idea behind SAWSDL is to add semantic annotations to the WSDL that describes service. The basic annotations that SAWSDL adds are inputs, outputs, operation, interfaces, and faults. In terms of SAWSDL, semantic annotations are simply bits of xml that are embedded as properties in the WSDL. These properties are URIs of ontology objects. This means that the annotation of a concept in SAWSDL or SA-REST is a way to tie together the concept out of the SAWSDL or SA-REST to a class that exists in ontology. An ontology is a conceptualization of a domain represented in terms of concepts and the relationships between those concepts. Furthermore, it

embodies an agreement among its users and provides a common nomenclature to improve interoperability. Since the adoptions of OWL (the Web Ontology Language, [25]) and RDF (Resource Description Framework, [24]) with the associated language of RDFS for RDF schemas as languages for ontology representation and semantic Web data, ontologies are most frequently represented in OWL or RDF. As with SAWSDL, SA-REST does not enforce the choice of a language for representing ontology or a conceptual model, but does allow use of OWL or RDF for representing ontologies or conceptual models. Since SAWSDL and SA-REST are more concerned with the data structure than with the relationships between objects and reasoning, RDF, as the simpler of the two languages, is likely to be used more frequently in the near future. For example, the output message in a WSDL may be annotated with a URI to an ontology class that logically represents that output. Taking the lead of SAWSDL, SA-REST annotates outputs, inputs, operations, and faults, along with the type of request that it needed to invoke the service. The latter is not required in SAWSDL because it deals with traditional SOAP-based services that primarily transmit messages via an HTTP Post because of the size of the payload, whereas RESTful Web services, being much lighter weight, can use either an HTTP POST request or an HTTP GET request. Later we will see that since SA-REST is in once sense a directive of SAWSDL, we can translate a SAWSDL service into a SA-REST service.

The point of adding annotations to the inputs and outputs of a service is to facilitate data mediation. Data mediation is effected in SAWSDL and in SA-REST not only by specifying the ontology concept that its message represents but also by specifying a lifting and lowering schema. What a lifting and lowering schema does is to translate the data structure that represents the input and output to the data structure of the ontology, which we call the grounding schema.

This grounding schema is needed is because it would be impossible to get everyone on the Web to agree on a single data structure for a concept. Furthermore, to force a data structure for a concept would hurt the flexibility of the service and the ideology of a read/write web. Lifting and lowering schemas are XSLTs or XQueries that can take an instance of an implementation-level data structure and turn it into an ontology-compliant data structure. By implementation-level data structure we mean the data structure that the service expects in the format that the service expects. This is a scalable data mediation solution because the service provider need provide only one lifting schema, which translates the output of the server to the grounding schema that logically represents the object, and one lowering schema that translates the grounding schema to the input of the service. This solution contrasts to having to provide a translation or mapping from one service to every other service that plans to utilize that service. In terms of an object-oriented style language, the lifting schema should be thought of as an "up cast" and the lower schema should be considered a "down cast," where the ontology class plays the role of the parent object and the service input and output plays the role of child object.

The first significant benefit of adding semantic annotation to RESTful Web services is data mediation. This is a key benefit since RESTful Web services have no way to specify the format of the inputs or outputs of the service. The following example illustrates the problem faced by standard (non-annotated) RESTful services: a user wants to plot the output of a service that tells the address of homes for rent on a map. Logically the real estate service returns a list of locations but the output is represented in XML. Assume that "address" constitutes an address line 1 field and an address line 2 field. The map service logically takes in a list of locations but the input needs to be represented in JSON and only has one address line. Logically these services should be able to work together with little effort, but they cannot; some work must go

mediating the data to be used between these two services. We propose to address this problem by annotating all inputs and outputs of the service with a URI to concepts related to "address" described in ontology. It is important to note that the annotation of the services inputs and outputs does not change the input or output of the service, nor does it place a restriction on the format. The annotation of a service is simply a way of attaching metadata to the service to be used later to glean information about the service. These metadata act as a loose standardization or normalization for data in the messages of the service.

Now assuming that the output of the real estate service points to a location object in an ontology and that the map service points to the same location object, data mediation could be done as follows: The first schema would not only point to the ontology for the output of the service but would also specify a lifting schema that could translate the output of the service into a form compliant to the ontology. It is important to note that the actual fields of the data structures are not annotated. The high-level concepts that the data structures represent are annotated. Likewise the mapping service will specify a lowering schema that would take data in the form of the ontology item and translate it into the form the service requires for input. In this way, the code for that mashup would invoke the real estate service, lift the output to the ontology form, lower the output the form for the mapping service, and invoke the mapping service. This bit of code would be general for all mashups. The solution is scalable, because for each service that is created the author need supply only a single lifting and lowing schema and not a mapping to all other services that may want to use this service in the future. By contrast the tools currently available require that all data mediation go on within the tool itself. The key role that ontologies play in our approach will be described later. Figure 2 demonstrates the process outlined above.

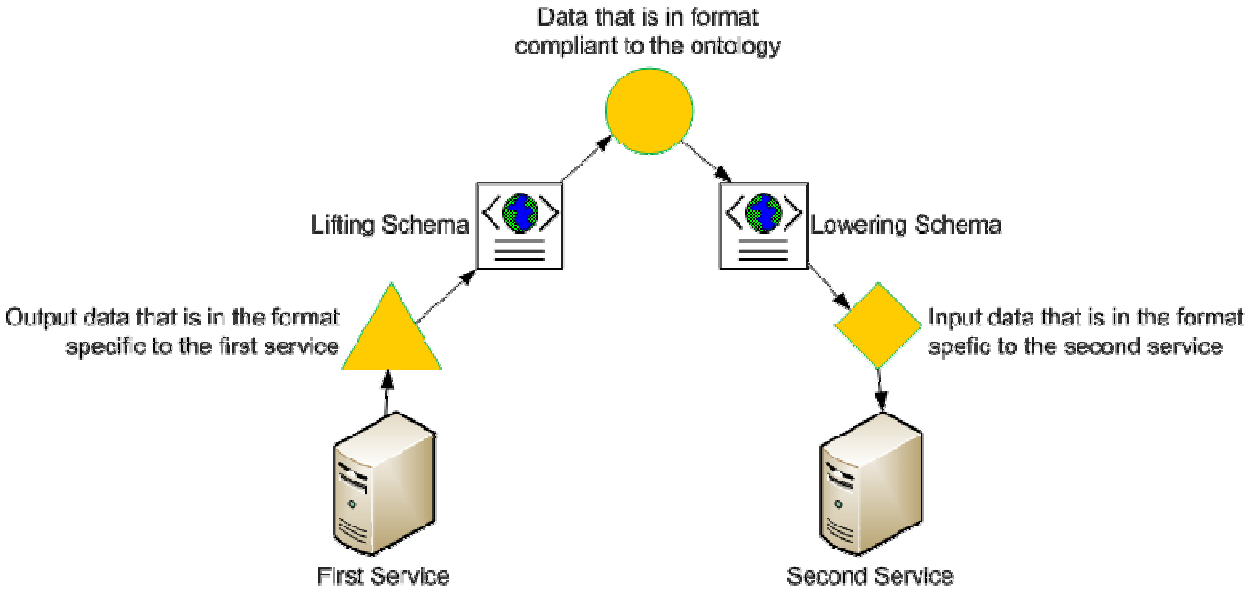

Figure 2: A demonstration of Lifting and Lowering Schemas

The second problem solved by adding semantic annotations to RESTful services is to determine automatically how services are invoked. Traditional SOAP-based services are invoked via an HTTP Post request because the amount of data present in an SOAP envelope can easily exceed the size limit of an HTTP Get request. Since RESTful Web services do not suffer from the same bloating as SOAP services, the style of invocation reflects a more traditional view of HTTP Get and HTTP Post. Traditionally, an HTTP Get request should be used when the request will not cause any change to the server, whereas an HTTP Post request might. The issue for a tool invoking a service is which type of request to choose. That is why in SA-REST we annotate the service with the type of request that should be used. The tools currently being used to create mashups circumvent this problem by always using an HTTP Get on external services. That is possible because the external services that are allowed are normally feeds such as RSS, and the retrieval of a feed does not cause a change on the server.

So far we have shown only what kind of annotations need to be added to make a RESTful Web service into a SA-REST service; we have not stated how or where to attach the annotations. In SAWSDL the semantic annotations that described the service were added in the WSDL for that service. That is a very logical place because typically there is a one-to-one correlation between a WSDL and a traditional SOAP-based Web service and since WSDL describes the service as do the annotations it seems logical to group them together. Most RESTful Web services do not have WSDLs, since a main objective of REST is simplicity. WSDLs are complex and create another artifact of the service that must be kept up to date as the service changes. Furthermore, the simplicity of RESTful Web services is such that they do not require a WSDL, whose primary use is to facilitate tooling support. Most RESTful Web services have HTML pages that describe to the users what the service does and how to invoke it. This HTML is in a way the equivalent of a WSDL for RESTful Web services and thus would be an ideal place to add semantic annotations. The problem, however, with treating a HTML as a WSDL is that HTML is meant to be human readable whereas a WSDL is designed to be machine readable. Microformats come into the picture here. Microformats are a way to add semantic metadata to human readable text in such a way that machines can glean the semantics. Microformats come in many different and competing forms. Recently the W3C has worked on the standardization of two different technologies called GRDDL [20] and RDFa [14]. GRDDL (Gleaning Resource Descriptions from Dialects of Languages) is a way for the author of the human-readable text to choose any microformat and also specify a translation, normally an XSLT, that translates the human readable text into machine readable text. RDFa is a way to embed RDF triples in to an XML, HTML, or XHTML document. As a preferred implementation of SA-REST we recommend the use of RDFa as a microformat because it is standardized by the W3C and as it is

a subset of RDFa, it has built in support for URIs and namespaces. We will first discuss how annotations will be added to an HTML page using RDFa and then we will discuss the more general case of using GRRDL.

We embed our semantic annotations in RDFa into the HTML page that describes the service, thus making the page both a human-readable and machine-readable description of the service and at the same time creating a single place to do an update if the service ever changes. This in contrast to attaching a separate document, such as a WSDL, that contains the annotations so that when the service is updated both the HTML (the human-readable document) and the formal description (the machine-readable document) must also be updated. RDF triples can be extracted from the XML, HTML or XHTML which has been annotated with RDFa via parsers or XSLTs. SA-REST leaves it up to the user how and where to embed the triples—they could be intermingled into the HTML or clustered together and not rendered by the Web browser. The subject of the triple should be the URL at which you would invoke the service; the predicate of the triple should be either sarest: input, sarest:output, sarest:operation, sarest:lifting, sarest:lowering or sarest:fault, where "sarest" is the alias to SA-REST namespace. The object of the triple should be either a URI or a URL to a resource depending on the predicate of the triple. Figure 3 and 4 give a detailed example of a SA-REST document for a Web service to search for houses on Craig's List.

```
<html xmlns:sarest="http://lsdis.cs.uga.edu/SAREST#">
 …
       <meta about=" http://craigslist.org/search/">
             <meta property="sarest:input"
                   content=
                   "http://lsdis.cs.uga.edu/ont.owl#Location_Query"/>

             <meta property="sarest:output"
                   content=
                   "http://lsdis.cs.uga.edu/ont.owl#Location"/>

             <meta property="sarest:action" content="HTTP GET"/>

             <meta property="sarest:lifting" content=
                   "http://craigslist.org/api/lifting.xsl"/>

             <meta property="sarest:lowering" content=
                   "http://craigslist.org/api/lowering.xsl"/>

             <meta property="sarest:operation" content=
                   "http://lsdis.cs.uga.edu/ont.owl#Location_Search"/>
       </meta>
 …
```

Figure 3: An annotated Web page to search for houses on Craig's List. Annotations not mixed with content

```
<html xmlns:sarest="http://lsdis.cs.uga.edu/SAREST#">
 …
       <p about=" http://craigslist.org/search/">
             The logical input of this service is an
             <span property="sarest:input">
                   http://lsdis.cs.uga.edu/ont.owl#Location_Query
             </span>
             object. The logical output of this service is a list of
             <span property="sarest:output">
                   http://lsdis.cs.uga.edu/ont.owl#Location
             </span>
             objects.  This service should be invoked using an
             <span property="sarest:action">
                   HTTP GET
             </span>
             <meta property="sarest:lifting" content=
                   "http://craigslist.org/api/lifting.xsl"/>

             <meta property="sarest:lowering" content=
                   "http://craigslist.org/api/lowering.xsl"/>

             <meta property="sarest:operation" content=
                   "http://lsdis.cs.uga.edu/ont.owl#Location_Search"/>
       </p>
```

Figure 4: An annotated Web page to search for houses on Craig's List. Annotations mixed with content

To allow more flexibility and to lower the barriers to entry, we allow the user to use GRDDL to attach annotations. To annotate a HTML page with GRDDL the author first needs to embed the annotations in any microformat. To the "head" tag in the HTML document, a profile attribute must be added that is the URL of the GRDDL profile. The profile attribute tells agents coming to this HTML page that it has been annotated using GRDDL. The final step in adding annotations is to add a "link" tag inside the "head" element that contains the URL of the translation document. Although any format may be used to add annotations to this page, the data that are extracted after the translation is applied to the document must result in RDF triples that are identical to the ones that would be generated via RDFa embedding. That is, a page annotated with GRDDL still needs to produce triples whose subject is the URL that is used to invoke the service, whose predicate is the type of SA-REST annotation used, and whose object is the URI or URL for the resource to which the predicate refers.

GRDDL has the advantage that it is less intrusive than RDFa. GRDDL allows the user to embed annotations in any way that is convenient, which could be a preexisting microformat or a new microformat only known by the user. It would also be possible using GRDDL for the user to embed no extra data in the HTML page that is not required by GRDDL and to have the entire metadata specific to SA-REST be contained in the translation. The advantage of RDFa is that the annotations are self-contained in the HTML page and the user need only create and maintain one document. In contrast, GRDDL forces the user to create two documents, the HTML page and translation document. RDFa has the additional advantage that it is a standardized microformat. The standardization makes it simpler for a developer to maintain and understand a page that has been created by someone else.

From the fact that SA-REST is a derivative of SAWSDL it follows that a SAWSDL can be generated from an SA-REST annotated page. This idea would be useful because there are many tools that have been designed for SASWDL, including programs for workflow automation, service discovery, and service publication. All these tools rely heavily on the annotated WSDL to accomplish their task. This means that these tools could not cover REST services since they lack a WSDL. Given the above-mentioned annotation scheme, we can create a SAWSDL from a SA-REST page. The ability to create this mapping between SAWSDL and SA-REST shows that SA-REST has semantic support similar to that of SAWSDL and allows SA-REST services to be used with preexisting tools.

The main features of a SA-REST HTML page that are used for computation are the RDF triples. These triples can easily be translated in to a simple XML by the application of an XSLT, the use of an RDFa extractor, or a GRDDL agent. The WSDL 2.0 standard defines a REST binding, which for purposes of this manuscript is a normal WSDL binding but does not include the SOAP envelope. We can create a simple XSLT that is a template for a SAWSDL that has a WSDL 2.0 REST binding. If this XSLT is applied to the abstract XML document, the input, output, schema mappings, and operations would be filled in to create a complete SAWSDL document. The input and output messages in the SAWSDL describe the data structures, and since the abstract XML includes only the concepts about the messages, not the data structure information, we must pull this information from the grounding schema of the ontology class that represents the concept.

In a similar way we can move from a SAWSDL to an annotated HTML page for a SA-REST service. We could use a simple XSLT to translate from the SAWSDL to the abstract XML. With this XML we could use another XSLT to translate to HTML. It would be difficult to write a useful HTML page knowing only the RDF triples. For that reason we add more data to the XSLT from the ontology. If the ontology contains comments about the concept, they are added. If the concept is a subclass of a different concept we add this to the XSLT along with any comments it may have.

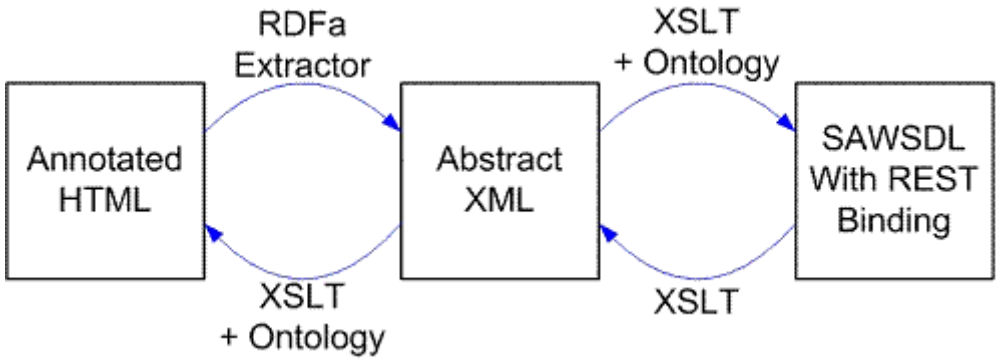

Figure 5: Translating from SA-REST to SAWDL and back

The newly created SAWSDL contains the generic data structures for input and output messages instead of the data structures that the service expects. The mismatch could cause validation errors when the service is invoked. For this reason the client invoking the service must know that data mediation via the lowering schema mapping must be done beforehand. As an alternative solution, a SA-REST service could validate the incoming message to see whether it is in the service-specific form or in a format compliant to the ontology. If the data is in the generic form the service itself could do the data mediation. The approach of the service itself checking to see if the data is in the correct format is a more flexible approach because the only

code that would need to be changed is the code for this service. This in contrast to having to change every client to first translate the input before invoke a service. The reason we propose both solutions is that the user may not always have access to the server code or the client code.

CHAPTER 3

BUILDING SEMANTIC MASHUPS

One of the most popular applications of RESTful Web services is mashups, which are basically Web sites that aggregate content from different providers. The popularity of mashups is partly due to the simplicity of RESTful Web services, since the language used for programming Web pages is typically a lightweight language or a scripting language. A mashup uses RESTful Web services to query the providers to get content typically in XML format. Differences between the definitions and representations of data from different providers necessitate a semantic approach for seamless integration of the data. Using semantics to integrate and coordinate mashups gives us SMashups (Semantic Mashups) [18]. As mentioned earlier, key difficulties in creating mashups are the amount of coding involved and in automating its creation. Much of the coding needed to create a mashup is required to mediate between the different data definitions and representations. A key difference between a mashup and a SMashup are the underlying services that are used. A traditional mashup would use RESTful Web services whereas SMashups use RESTful Web services with SA-REST annotations. The annotations give SMashups the ability to know more about what the service is going to do and what the inputs and outputs are, so that data mediation can be done automatically without human intervention. In this section we describe a system that we developed to help a user create mashups without having to know any programming language or do any programming. In these SMashups, we use semantics to do automatic data mediation.

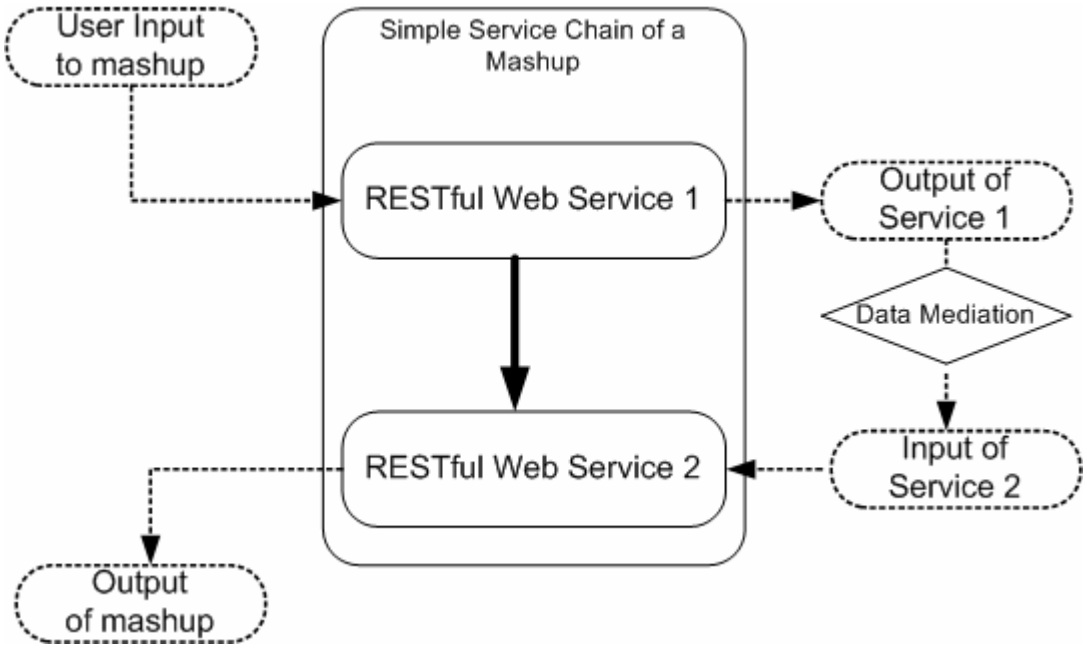

Figure 6: The typical architecture of a mashup that uses two services

In Figure 6 we show the typical architecture of a mashup that uses two RESTful web services. The key component of the system that is not the scaleable or flexible is the data mediation component. The data mediation component is the part in traditional mashups that is time consuming and requires human intervention. Our system attempts to automate that component of mashup architecture.

In our system we allow the user to first specify the chain of services that need to be invoked. That is, the user specifies the URLs of the annotated HTML pages that describe the SA-REST services to be invoked in the SMashup in the order in which they are to be invoked. The system then sends a request via an AJAX call to the proxy server, which in turn downloads the requested annotated Web page and sends it back to the client. If three SA-REST services were to be used in a particular SMashup, then three URLs to annotated HTML pages would have to be supplied by the user and three requests to the proxy server would be submitted. The proxy

server is used to circumvent the javascript security policy, which states that a Web page can not programmatically download a resource from a domain other than that from which the code itself was downloaded. That is, if code to create SMashups is downloaded by a user from one server, a Web page cannot be downloaded by that code from a different server. This is why we funnel all requests through the server from which we downloaded the SMashup code. The proxy server is used only to host the SMashup code and relay messages from the client, and for the creation of a SMashup only one proxy server can be used. In Section 6 we discuss how to remove the proxy server altogether. When the annotated HTML page is back on the client, an XSLT is applied to extract the RDF triples. The XSLT that is used to extract the RDF triples is either the XSLT that is used to extract RDF triples form a page annotated with RDFa or the XSLT that is specified via the GRDDL link tag. These triples are then used to create a description of the service and to display the description to the user. At this time the user can go through and specify where all the inputs should be gathered from. The user can specify that an input can be an input to the service or an output of any service that is higher up the chain as long as the objects semantically match. That is, that if the first service returns a location object as an output, the input to the second service can either be obtained as an input to the smashup or be the location object from the first service. When we say that two objects semantically match we are referring to the fact that objects have been annotated with the same ontology concept or that the service earlier in the chain uses a child concept of the object that is later in the chain. That is to say that one object is extended from another. Also at this point the user is allowed to specify which objects are the outputs of the SMashup. An output of a SMashup is one or more of the outputs of any of the services that is used in the SMashup.

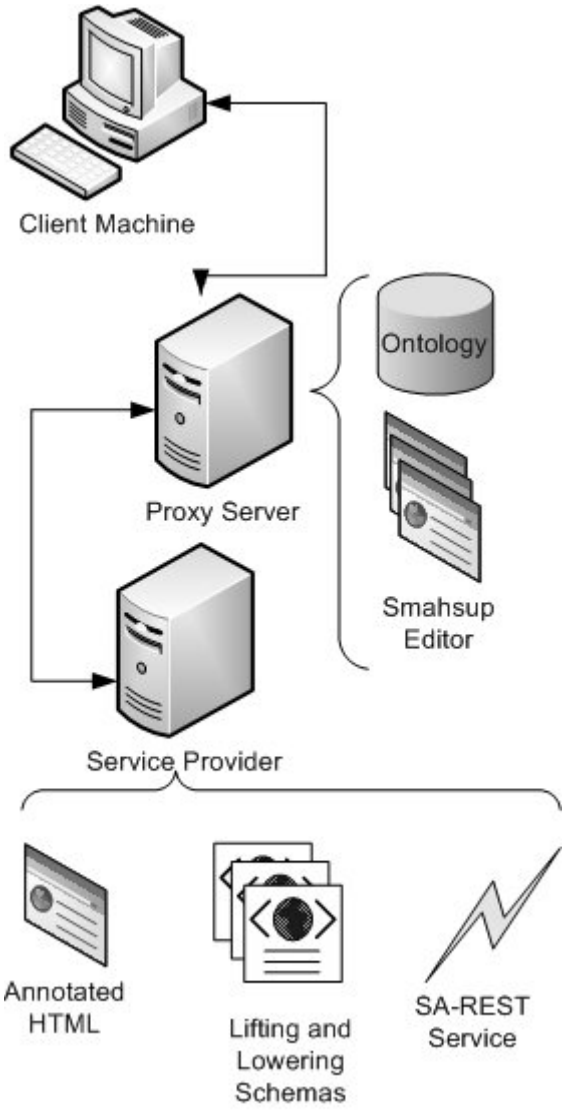

Figure 7: High-level preferred architecture of a SMashup

After all inputs and outputs have been specified, the user can prepare the service to be executed. At this step the system takes what objects are to be used as input to the service and creates an HTML form with which to prompt the user. The system knows what fields to create by going to the ontology, finding the attributes of the concept, and recursively going through the attributes until only primitive attributes are left. The names of these primitive attributes are what is presented to the user for inputs to the SMashup. It is important to note that these fields may not match up with any of the input fields for a service that will be invoked. After the user

has entered all the information, the user can start the execution of the service. At this point the HTML fields that were created will be turned into a XML that confirms to the grounding schema of the input objects as described by the ontology. These XML fragments— there is one for each input object— are placed in an intermediate object hash map and saved for later use.

After inputs are entered by the user and execution of the SMashup starts, each service in the execution chain will be executed in turn. This means that the engine will first look up the inputs that the service needs in the intermediate object hash map. The objects in the hash map are in the form of the grounding schema that reflects that ontology. The lowering schema is then used to convert the data from the grounding schema to the concrete-level format needed to invoke the service, which may be XML, JSON, or a query string. If the service is to be invoked via an HTTP Get request, the output from the translation from grounding schema to implementation schema is appended to the URL from which the service should be invoked. This string is then passed to the proxy sever, which in turn downloads the resource from the URL. The result is then passed back to the client where the lifting schema is applied to convert the output to the grounding schema. This grounding schema XML is then placed in the intermediate objects hash map. If the service is to be invoked via an HTTP Post, the lowered data will be posted to the proxy server who will in turn post the data to the appropriate service. The proxy server knows where to use an HTTP Post or an HTTP Get based on how the proxy server itself was invoked. Now this process is repeated for the next service in the execution chain.

After all services have been executed, the engine loops through all possible outputs of the service, finds all objects that user requested to be returned as outputs, and displays them to the

user. Note that at this time the output objects that are being displayed to the user are in the grounding schema format and not the implementation-level format. Possible output formats are HTML, XML, or even javascript. In Figure 8 we show the message flow that occurs during the execution of a SMashup that involves the client, proxy server and two service providers. The diagram does not include a request that occurs during the creation of a Smashup, only the execution of a SMashup.

The client side code that represents the execution of a SMashup can be saved in a file and then embedded into a preexisting web page. The only thing to note here is that the code file must reside on the proxy server and not the same server that is hosting the preexisting web page. This is again because of the javascript security policy. Some piece of code must reside on the proxy server so that the URL of the code can be embedded into the HTML by the page author. Since code is downloaded by the browser form the proxy server, the browser can be programmatically requested to download farther resources from proxy server which in turn can download resources from any other available server. This creates two models for the execution of SMashups. The first is force users to go to one central site that host all SMashups in order to use one. This model would be best if provider was trying to monetize this service or drive traffic to their site. The other is to allow users to embed a SMashup into there own pages. This model gives a more integrated look and fell to the SMashup creators own site and would most likely be more widely used then the first proposed model due to user demand. One approach is not more scaleable than another due to the fact that all the processing is done on the client side and not on the proxy server. The proxy server will need to process the same number of requests to forward invocations an resource requests in either model since both models still rely on the proxy server.

Because all the computation for SMashups is done on the client machine, the computation needed to process the request scales with the number of users.

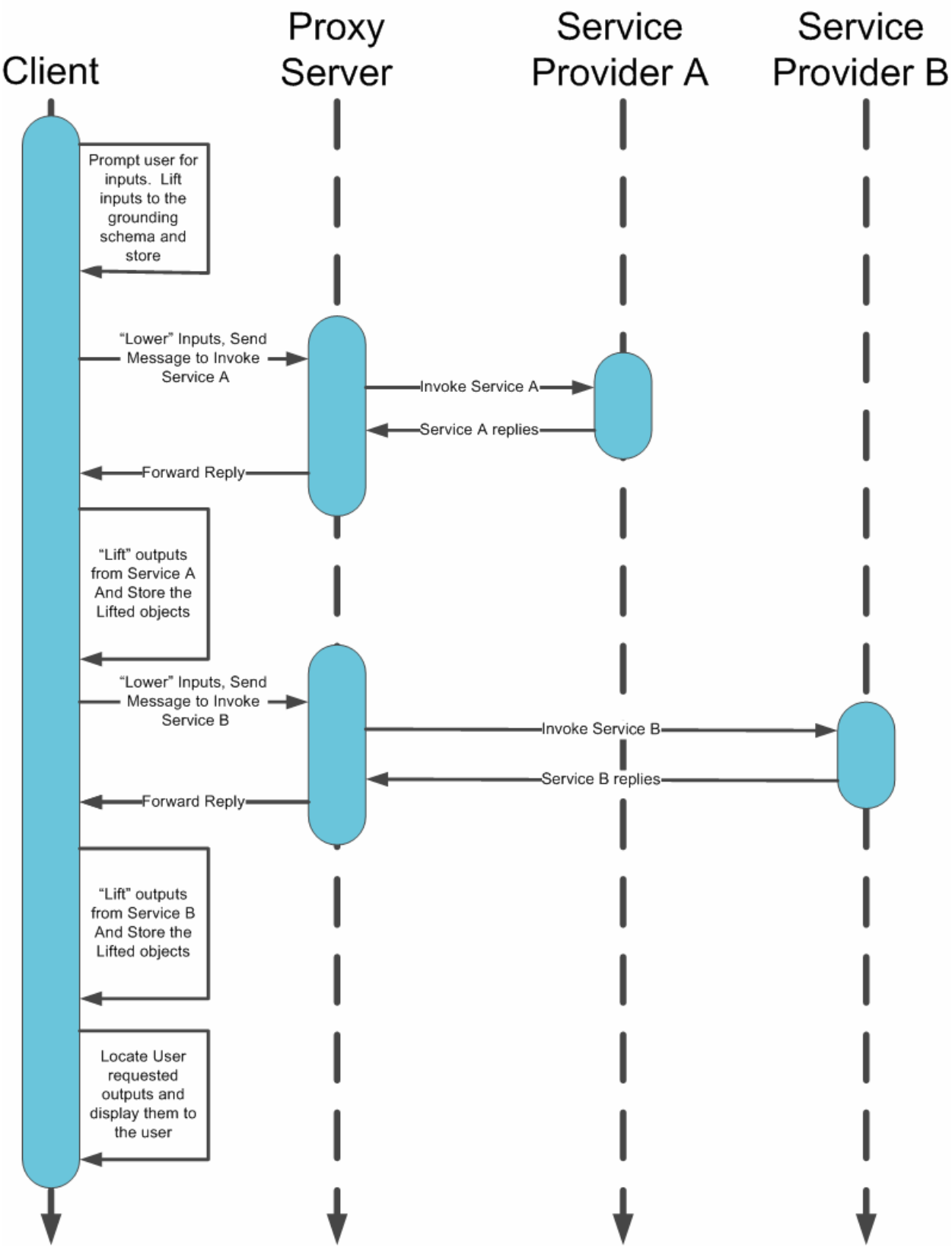

Figure 8: Message flow of a SMashup

CHAPTER 4

EVALUATION AND RELATED WORK

The idea of a user's being able to create mashups without having to write any code is a recently popular one and appears to be an application that will be commonplace on the Web in the future. This outcome is evident in the fact that most large Web development companies have recently developed applications that allow users to create mashups with graphical tools. Notably, Google, Yahoo, and IBM have developed tools to create mashups in the recent past, and Microsoft has a tool called Popfly[19] that is still very new.

Yahoo Pipes is a Web application that presents the user with a graphical interface that allows them to create mashups from feeds and a few other specialized services. Yahoo Pipes focuses on the aggregation of feeds from different providers. An example would be showing Flickr photos of keywords that appear in New York Times articles. That is to say, keywords are pulled from the New York Times RSS feed and then used to search Flickr. The search results from Flickr are presented as a feed aggregated with the original New York Times feed. Through a well-designed interface nontechnical users can employ Yahoo Pipes to create fairly complicated aggregate feeds. The drawback of Yahoo pipes is that it is limited to feeds and specialized services. That means integration with a mapping service would require coding outside of Yahoo Pipes. So the example of combining Craig's List and Google Maps is not possible using only Yahoo Pipes. In addition, since the focus is on feeds, it is not possible for

Yahoo Pipes to handle a service that has a complex input or output. For example, there is no way to post XML data to a service in Yahoo Pipes. Though Yahoo Pipes has controls to deal with XML, CSV, and JSON, the controls are not expressive enough to handle complex data manipulation.

The Google Mashup Editor is a new service from Google that is still in private beta. Unlike Yahoo Pipes, which focuses on feeds, Google Mashup Editor focuses on creating Web pages. The interface is a text window that allows the user to type in arbitrary HTML code along with a few proprietary tags that equate to a more complex object such as a Google Map widget or a RSS feed. These proprietary tags can interact with one another as long as the schemas match up. An RSS feed, for example, may return the location of homes for rent and could then be attached to a map widget to display the homes on the Google Map. It is important to know that the RSS needs to contain extra metadata that are not normally present in RSS so that the feed can become compliant to the map schema. The fact that outputs of services must have extra encoding or special tags so that the Google Mashup Editor knows how to handle the services is the weak point of this tool. A service for which the output schema does not contain the extra encoding or special tags cannot be used with Google Mashup Editor until the service provider rewrites the service. The available widgets for Google Mashup Editor are mainly Google Maps, RSS feeds, Google Base searches, and some HTML controls. The limited number of widgets and the forced output schemas make this tool weak in terms of built-in functionality. Nevertheless, any mashup can be created with this tool. Since the user input for the tool is simply a text box where the user enters arbitrary HTML code, the author of the mashup can use the built-in functionality to create some parts of the mashup and write custom javascript and HTML code for the parts that are missing. The fact that this tool allows the user to make

arbitrary changes to the underlying code of the mashup makes it very powerful through extensibility. At the same time, unfortunately, the user interface requires a steep learning curve, as the user needs not only to learn the new widgets included in the tool but also to have a firm understanding of HTML, javascript, and XML.

QEDWiki is a canvas on which the user designs a mashup. QEDWiki allows the user to use traditional SOAP-based Web services, RESTful Web services, RSS feeds, or data base connections. To combine services, QEDWiki depends on matching schemas or the user can parse out the data via tools or widgets provided in QEDWiki or enter raw code. There are very few restrictions on what the user may do in this tool. QEDWiki is more powerful and flexible than the other two tools mentioned but the flexibility and power come at a cost. QEDWiki needs matching schemes of input and output of services or needs the author of the page to add a good deal of programming logic. QEDWiki is more of a platform where the user can go and write code for a mashup with assistance of tools than a tool to create mashups.

In all of the current technologies for mashups, interoperability is only at a syntactic (keyword-matching) level; there is no support for semantics and hence no support for semantic data mediation as in the case of SA-REST and SMashups. To create advanced and complex mashups, current technologies are limited and require much human intervention. If a good deal of human intervention is needed to create a mashup there is little advantage in using a tool to begin with. Comparing Yahoo Pipes with QEDWiki, we can see that Yahoo Pipes has the simpler interface and does not require the user to have any extra knowledge such as XML or HTML to use the service. More powerful mashups can be created with QEDWiki based on the

amount of knowledge the user has in the area of SOAP, XML, and HTML. Yahoo Pipes can create only very simple and limited mashups, while QEDWiki can create very complex mashups. By contract to both of these tools, SMashups can use a simple interface and create complex mashups. Simplicity is achieved by hiding the data mediation part of the mashup from the user. Data mediation can be done automatically in SMashups because the services are not being paired through syntactic matching of input and output schemas but through semantic matching of the schemas. This semantic matching is possible because of the semantic annotations added to the service via SA-REST.

Figure 9 is a chart comparing Google Mashup Editor, listed as GME in the chart, Yahoo Pipes, listed as pipes, and QEDWiki. We attempt to compare the services on built-in functionality rather than functionality that is possible through extravagant workarounds. We compare the mashup tools on the following attributes: RSS—does the tool have a way to retrieve a RSS feed and parse its XML-based output; Google Maps— does the tool have a way to plot locations on and display a Google Map; Yahoo Maps—does the tool have a way to plot locations on and display a Yahoo Map (we show both Yahoo Maps and Google Maps in the chart to show whether the tool really can interact with such services or is capable of interacting only with internal services); Non XML Output—if the service returns an output in a format other than XML, is the service able to parse and make use of the data that it may contain; Non Query String Inputs—can the service be invoked with parameters if the parameters are not to be passed in the query string (an example of this is if a service needs to have XML sent as a input); HTTP Post—can the tool invoke a service that requires an HTTP Post message; HTTP Get—can the tool invoke a service that requires an HTTP Get message; Extensible—can a service that was not

included in the tool and not of a standard type such as RSS or ATOM be invoked; and User Knowledge Need—what are the requirements of a user's knowledge before they can use the tool.

Feature	**SMashups**	**GME**	**Pipes**	**QEDWiki**
RSS	X	X	X	X
Google Maps	X	X		X
Yahoo Maps	X			X
Non XML Output	X		X	X
Non Query String Inputs	X			X
HTTP Post	X			X
HTTP Get	X	X	X	X
Extensible	X	X		X
User Knowledge Needed	Low	Medium	Low	High

Figure 9: Table comparing the feature sets of SMashups with Google Mashup Editor, Yahoo Pipes, and QEDWiki

CHAPTER 5

WALKTHROUGH EXAMPLE

In this section we walkthrough the creation and execution of two example SMashups using a combination of real word and imaginary RESTful web services in order to bring out some features that are possible while using SMashups. We use a simple prototyped system for these examples. The system has the same architecture as described in section 3. Due to the fact that SA-REST is not yet adopted the annotated HTML pages along with the ontologies the pages are annotated with reside on the proxy server. The proxy server is running Apache Tomcat. The appendices contain a install guide and user manual for this prototype.

In this first SMashup example we show the classic Craig's List and Google Map mashup. Map mashups are the most prevalent type of mashup on the web and are the general thought when the term mashup is mentioned. Map mashups are slightly different then pure mashups. The general idea of a mashup is to call a number services and combine the output of the services to form a new service. Map services are not callable services in the since that a RESTful Web services are callable. Map services do use RESTful protocols to keep the client view of the map update by constantly sending new images to display but the author of the web page that contains the map does not explicitly invoke the RESTful Web services. Instead of explicitly invoking the RESTful Web service the author of the Web page makes programmatic calls to an API, typically written in javascript, which resides on the client which in turns makes calls to the appropriate

RESTful Web service. Due to this type of interface mashup tools need to call a wrapper Web service in order to access the map service in a consistent way. The wrapper service would take in parameters and return executable code that could be ran on the client. This means for example if the user wanted to plot points on a map the user would have to call the wrapper service passing the points that are to be plotted as parameters and the return of the service would be executable code that would generate, display, and plot the given points. This is slightly unintuitive but allows a tool that generates mashups to work solely with RESTful web services and not need to create special logic for every map application. If a tool created special logic to handle the creation of javascript code for a particular map service then the tool would loose flexibility and scalability due to the fact that only map interfaces that had special logic added to them could work and if a new map interface was needed, the designer of the tool would have to create special logic for the new interface. It is also important to note that since the wrapper service returns executable code that is designed to create GUI elements there is no semantic meaning to the output. Since there is not semantic meaning of the output, the output can not be used as the input of another service.

Figure 10: Screen shot of design phase of SMashup creation

This SMashup, as mentioned earlier, combines Craig's List and Google Maps. Craig's List logically receives a "house query" as an input and a list of "locations" as an out put. These objects, "house query" and "location", can be annotated with any ontology but one of the large barriers of wide spread adoption for this approach is the creation, acceptance, and wide spread adoption of ontologies that describe common data structures needed in mashups. For this example we are using a temporary ontology that was developed in house but was designed to resemble an ontology that could appear in the real word as much as possible. Google Maps logically takes a list of "locations" in as a parameter and returns "executable code" as an output. Since the output of the first service in the chain, Craig's List, and the input of the second service in the chain, Google Maps, match the two services can be mashed up. To get the metadata loaded about the service the user enters the URL of the first service's annotated Web page into

the appropriate input field. Now the user sees the metadata for that service. This metadata includes invocation URL, input objects, and output objects. Next the user will enter the URL for the wrapper service of Google Maps and the metadata for that service will be fetched. The user should then link the two services together by specifying the input to the Google Maps service should come from the output of the Craig's List service. The user also needs to specify that the output of the entire SMashup should be the executable code that the Google Maps service returns. Figure 10 shows the processes for this example. This is the part that makes SMashups different form normal mashups, the user does not need to specify any kind of data mediation form one service to the other; data mediation is done automatically.

After the service chain is specified for the SMashup the user runs the command to create the SMashup. This brings up the a screen where the user has the opportunity to run the newly created SMashup. There is a HTML form that allows the user to enter values that make up a "house query". It is important to know this HTML form matches the ontology version of "house query" and does not match the fields of the Craig's List API. After this form is filled out and the SMashup started, a Google Map will appear below the HTML form with location of houses to rent showing on the map. This is seen in Figure 11.

A real advantage of SMashups over traditional mashups is that services can be interchanged for semantically similar services without any extra work. A service is semantically similar to another service if the operation, output, and input map to the exact same ontology object or to a super class of the original ontology object. This is an advantage because after this mashup up is created Google Maps can be replaced with Yahoo Maps because both services do

the exact same thing. In traditional mashups a change like this would require much code change because now the data mediation would have to change. Also the first service in the chain, Craig's List, can be changed also. Since the HTML form that was generated to coincide with the ontology and not the service the user does not need to reenter their data before re-executing the new service. An example of why services would want to exchange would be that Google Maps may not be accurate for a particular neighborhood or Yahoo Maps may show more detail for a particular area. Likewise there could be a real estate service that has different or more listings than Craig's List for a particular area.

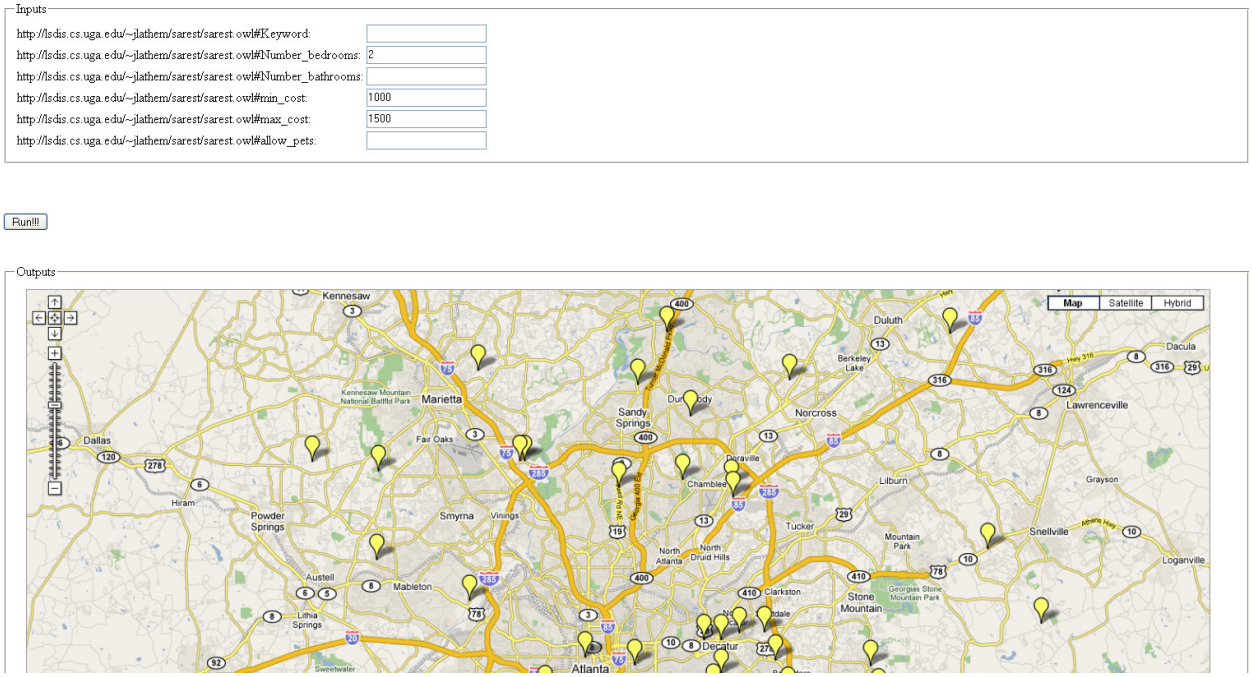

Figure 11: Screen shot of a SMashup after execution

The second example of a SMashup we use two made up services in order to bring out some unique features of SMashups that the first example did not bring out. This mashup consist of a stock portfolio service that takes login credentials as an input and returns a RSS feed that list

out owned stocks as the first service. For the second service we use a simple stock quote service that receives an stock symbol as a input and returns the current stock quote. This SMashup would be built in a very similar manner as the first service was built. Likewise, this SMashup will be executed in a similar manner as the first service. There difference between this SMashup and the first example is the output. In the first example the output was executable code. In this example the output is a object that represents a list of stock quotes. This also is an important difference than traditional mashups. Traditional mashups will return feeds, GUI code, or bits of arbitrary XML. These outputs are not very useful except for the purpose they were intended. The output of SMashups is a object that has semantic meaning which among other things makes the SMashup reusable. This means that the created SMashup could be reused as a SA-REST service in another SMashup. The input and output of this SA-REST service would be the same as the input and output of the SMashup. A very trivial annotated HTML page could be created from the SMashup by the user only supplying the operation of the SMashup.

The reusability of SMashups could create a new type of social network. Where current social networks are united via common interests such as music, art, or friendship this new type of social network would be united though use of common services or a common goal of some end service. Each user would create his or her SMashup and publish it then other users could come along and build or extend on that service. Such a social network would lead to an obvious way to measure quality of service for mashups. A user may only trust services from his friends or only trust services that the majority of his friends use.

CHAPTER 6

FUTURE WORK

There are several flaws in the system as it is now. Though a service may specify a map of an input or an output object via a lifting or lowering schema to an ontology concept, the mapping could be one-to-one but not onto. As ontologies grow in richness and detail the frequency with which data mapping cannot be onto will surely go up. That is, the service-level data structure of a service may have four fields, where as the ontology data structure may have five fields. The four fields of the service-level data structure may map directly to four of the five fields of the ontology data structure, leaving nothing to map to the fifth element. If a service later on in the execution chain relies on this object and the fifth element that does not have a value assigned to it, the service call will fail even though the inputs has the correct data structure. Similarly, the mapping from the ontology data structure to the implementation-level data structure could be onto but not one-to-one. In the example above, the ontology would have to map two elements on one element in the service-level data structure. This would cause loss of precision. These problems are present to in many object-oriented languages that allow up casting and down casting of objects and could be addressed by adding much lower level annotation to the Web service. That would mean, instead of annotating a message only with a class object in an ontology, annotating the message with the class object and the attributes of the class that is actually being taking advantage of. Another solution would be use less general ontologies. That would mean, if a message has only four fields, finding an ontology that

contains a class that closely resembles the message with only four attributes. A negative impact of this solution would be the appearance of many ontologies that would not be reusable.

In future work we plan to develop many of the tools that users are accustomed to seeing with traditional semantic SOAP-based Web services, including a discovery tool, an annotation tool, and an abstract processes composition tool. These tools exist for SAWSDL already. Lumina[23] is a semantic discovery tool that is built on top of UDDI. There is also a runtime engine that can take an abstract process description as an input that exists as part of the METEOR-S project and is built on top of BPEL4WS. The discovery tool would be able to find concrete RESTful Web services from an abstract description of the service. For example, a user may describe a service that has operation that "FindsBusiness," takes in as input a "Keyword" object, and returns a list of "Location" objects. There are many services that could do such a look-up, but if the user does not know any of them or can not remember a URL this would be a helpful feature. To create such a service there would need to be a registry where SA-REST services could publish their annotated HTML pages. The registry would also need a search interface where the user could choose ontology objects that specified any of the following: input objects, output objects, and operation concepts. If the process mentioned above were used to turn SA-REST-annotated HTML pages into SAWSDL, tools such as Lumina could be used. More important than a simple convenience, this registry could be used in the creation of an abstract process composition tool that would allow users to build a mashup up specifying only descriptions and order of services that should appear in the mashup, without the need to specify any concrete services. Then at runtime the tool would discover concrete services that met the abstract description and invoke them. This system could also try to optimize the choice of services based on some metric such as time or cost. These tools are possible for SA-REST

services and not traditional RESTful Web services due to SA-REST services have a semantic description of the service in the annotated HTML page where as traditional RESTful Web services have no standardized description.

The main reason to have the proxy server is to circumvent the javascript security policy. As a feature work we will try to implement the design tool and execution engine as a browser plug-in. This would remove the need to have a proxy server because the browser plug-in would have more permissions and less security restrictions than code running in a web page. The browser plug-in could be used to design the mashup and save the design as a javascript code snippet that could then be embedded into a Web page.

Currently, for an output of one service to be used as the input of another the objects must logically be the same or one must be inherited from the other. This implies that both services use the same ontology to annotate the same logical object. The need for the entire Web to agree on a single ontology or a group of ontologies is one of the main hindrances to the adoption of semantic Web and would likewise hinder universal adoption of such a system as we have described in this paper. Therefore, as future work, we would like to consider two objects to be logically equal if they are both represented by the same ontology object in the same ontology or if the are represented by equivalent ontology objects in two different ontologies having a mapping from one ontology to the other.

CHAPTER 7

CONCLUSIONS

In this thesis we have shown a novel way to make a RESTful Web service semantic while not altering the service itself and not introducing a new resource that would have to be maintained along with the service. This innovation gives RESTful Web services much of the power that traditional SOAP-based services have, while not taking away any of the flexibility and simplicity for which RESTful Web services were adopted. We have also shown that using these semantic RESTful Web services we can create arbitrary mashups without many of the limitations that plague current mashup editors. In addition, our mashup system is scalable with the number of services.

REFERENCES

[1] R. Akkiraju, J. Farrell, J.Miller, M. Nagarajan, M. Schmidt, A. Sheth, K. Verma, "Web Service Semantics -- WSDL-S," a W3C member submission, Nov. 7, 2005. http://www.w3.org/Submission/WSDL-S/

[2] A. Sheth, K. Verma, K. Gomadam, “Semantics to energize the full Services Spectrum,” Communications of the ACM (CACM) special issue on Services Science, 49 (7), July 2006, pp. 55-61.

[3] M. Nagarajan, K. Verma, A. Sheth, J. Miller, and J. Lathem, Semantic Interoperability of Web Services - Challenges and Experiences, Proc. of the 4th IEEE Intl. Conference on Web Services, Chicago, IL, September 2006, pp. 373-382.

[4] WSDL-S, http://www.w3.org/Submission/WSDL-S/

[5] Web Services Modeling Ontology, http://www.wsmo.org

[6] Sivashanmugam, K., Verma, K., Sheth, A., Miller, J., Adding Semantics to Web Services Standards, ICWS 2003

[7] OWL-S, http://www.daml.org/services/owl-s/

[8] Verma K., Gomadam K., Lathem, J., Sheth A., Miller, J., Semantically Enabled Dynamic Process Configuration, LSDIS Lab Technical Report 2006.

[9] Google Maps, http://maps.google.com

[10] Classic Yahoo Maps, http://maps.yahoo.com/

[11] Yahoo Maps, http://maps.yahoo.com/broadband

[12] Craig's List, http://www.craigslist.com

[13] Anupriya Ankolekar, Markus Krötzsch, Thanh Tran, Denny Vrandecic, The two cultures: mashing up Web 2.0 and the semantic Web, Proceedings of the 16th international conference on World Wide Web

[14] RDFa, http://www.w3.org/2006/07/SWD/RDFa/syntax/

[15] Google Mashup Editor, http://editor.googlemashups.com/

[16] Yahoo Pipes, http://pipes.yahoo.com

[17] QEDWiki, http://services.alphaworks.ibm.com/qedwiki/

[18] A Sheth, K Verma, K Gomadam, Semantics to energize the full services spectrum, Communications of the ACM, 2006

[19] Microsoft Popfly, http://www.popfly.ms/

[20] Gleaning Resource Descriptions from Dialects of Languages (GRDDL), http://www.w3.org/TR/grddl/

[21] SAWSDL, http://www.w3.org/2002/ws/sawsdl/

[22] John Domingue, Liliana Cabral, Farshad Hakimpour, Denilson Sell, Enrico Motta, IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services

[23] Lumina, http://lsdis.cs.uga.edu/projects/meteor-s/downloads/Lumina/

[24] Ora Lassila, Ralph R. Swick, Resource Description Framework (RDF) Model and Syntax Specification

[25] I Horrocks, PF Patel-Schneider, F van Harmelen, From SHIQ and RDF to OWL: The making of a web ontology language, Journal of Web Semantics, 2003

[26] Tim O'Reilly, What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software, O'Reilly Network

APPENDIX A – INSTALL GUIDE FOR SMASHUP ENGINE

1. Download and install the Java Runtime Environment version 1.5 or greater. The Java Runtime Environment can be found at http://java.sun.com. There version is important.
2. Download and install Apache Tomcat version 6 or greater. Apache Tomcat can be found at http://tomcat.apache.org/. For the rest of this install guide we will assume that Apache Tomcat is running at the following address http://localhost:8080.
3. Download the SMashup war file. This file can be found at http://lsdis.cs.uga.edu/~jlathem/sarest.
4. Place the war file in Apache Tomcat "webapp" directory.
5. Restart Apache Tomcat
6. At this point the SMashup engine should be installed and ready to use. You can view the main page for the SMashup engine at the following address http://localhost:8080/sarest.

At this point this tool is only guaranteed to work for Mozilla Firefox version 1.5. The use of this tool with any other browser may yield unexpected results.