Model Problems in Technologies for Interoperability: OWL Web Ontology Language for Services (OWL-S)

Chris Metcalf
Grace A. Lewis

April 2006

Integration of Software-Intensive Systems Initiative

Technical Note
CMU/SEI-2006-TN-018

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

List of Figures

List of Tables

Abstract

In a services-oriented environment, services are constantly being added and removed. Application developers often do not have control over the services they utilize. What would happen if a service required by an application were removed from the environment or had its interface changed? What if a new and better service were introduced that an application might be able to utilize? Existing services-oriented frameworks do not protect application developers against these contingencies.

The OWL Web Ontology Language for Services (OWL-S) is a language to describe the properties and capabilities of Web Services in such a way that the descriptions can be interpreted by a computer system in an automated manner. This technical note presents the results of applying the model problem approach to examine the feasibility of using OWL-S to allow applications to automatically discover, compose, and invoke services in a dynamic services-oriented environment.

1 Introduction

Service-oriented architectures (SOAs) require a different perspective on software development than traditional application-centric methods.[1] SOAs support the decomposition of business processes into reusable elements implemented as services. These services provide support for tasks such as checking the inventory status of a particular item, obtaining the current location of a package, or checking the status of a customer. Developers can then build applications, such as order processing or customer account management, that take advantage of these services in new ways—reducing duplication of functionality within applications and making it simple to modify the internal behavior of a service without affecting existing applications.

To build those applications, developers query a service repository and obtain a list of addresses for those services that match the request. The developer selects a service from the list and includes calls to the service in the application code. The responses obtained from the call to the service can be used as input to additional services that were discovered in the same fashion. In this setting, if the service changes its interface or is not available at runtime, the application will receive an error. Currently, service discovery, composition, and invocation are done in a static manner at design time, rather than in a dynamic manner at runtime.

However, in a services-oriented environment, services are constantly being added and removed. Application developers[2] often do not have control over the services they utilize. What would happen if a service required by an application were removed from the environment or had its interface changed? What if a new and better service were introduced that an application might be able to utilize? Existing services-oriented frameworks do not enable application developers to deal with these changes.

The dynamic discovery, composition, and invocation of services is one potential solution to this problem. If applications were no longer restricted to the services they were developed to work with, they could discover new services based on the functionality they require at a particular time. Furthermore, dynamic service composition would allow applications to chain together diverse services automatically, based on their goals or required quality of service rather than a predefined process. Dynamic service invocation would allow applications to

[1] A *service* is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model. A collection of services with well-defined interfaces and shared communications model is called a service-oriented architecture (SOA). A system or application is designed and implemented using functionality from these services [Lewis 04].

[2] Even though *client application* is often used to refer to a consumer of services, all discussions in this document using that term also apply to the situation where the consumer of a service is another service.

invoke new services or sets of services without developer or user intervention. These dynamic environments also enable the flexibility being pursued by both industry and the U. S. Department of Defense (DoD) to respond to changing business and mission requirements.

The OWL Web Ontology Language for Services (OWL-S) provides developers with a strong language to describe the properties and capabilities of Web Services in such a way that the descriptions can be interpreted by a computer system in an automated manner [Martin 04a]. The information provided by an OWL-S description includes

- ontological description of the inputs required by the service
- outputs that the service provides
- preconditions and postconditions of each invocation

The goal of OWL-S is to enable applications to discover, compose, and invoke Web Services dynamically. Dynamic service discovery, composition, and invocation will allow services to be introduced and removed seamlessly from a services-rich environment, without the need to modify application code. If the information it needs to achieve its goals can be described in terms of an ontology that is shared with service providers, an application will be able to detect new services automatically as they are introduced and adapt transparently as the programmatic interfaces of services change. Although it is not the only technology being pursued to support dynamic environments, OWL-S is far enough along in its development to be used as a proof of concept, if not a potential solution.

This technical note describes the process that was followed to implement a model problem to examine the feasibility of OWL-S as a technology for discovering, composing, and invoking services in a dynamic services-oriented environment. Model problems are a very simple and cost-efficient way to understand what technologies can and cannot do within a specific context. The goal of this work was to discover the potential benefits and drawbacks of OWL-S and provide guidance for those wishing to explore and improve it for the future. This note is the second in a series of explorations of technologies for interoperability using the model problem process by the Integration of Software-Intensive Systems (ISIS) Initiative at the Carnegie Mellon® Software Engineering Institute (SEI). The first note examined claims for the benefits of Model-Driven Architecture (MDA) as an aid in achieving interoperability between systems.[3]

In Section 2 of this technical note, we define OWL-S. In Section 3, we explain the model problem approach. In Section 4, we detail the design and implementation of the model solution to examine OWL-S. In Section 5, we present the findings from the examination detailed in Section 4. In Section 6, we offer recommendations for the use of OWL-S. And in Section 7, we provide a brief summary of the note.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[3] To read the technical note on issues regarding MDA, go to http://www.sei.cmu.edu/publications/documents/05.reports/05tn022.html.

2 What is OWL-S?

OWL-S is a markup language that enables the description of Web Services in a way that they can be discovered automatically, composed into more complex services, and invoked with a high degree of automation [Martin 04a, Martin 04b].

The types of tasks that OWL-S supports can be seen in the example of a person making arrangements to travel to and attend a conference. Suppose a person enters travel dates and destination information into an "OWL-S enabled" application, choosing these criteria: (1) nonstop travel and lowest fare for a flight and (2) availability and lowest rate per day for a rental car. Behind the scenes and without additional intervention by the person, the application

1. discovers the set of airline services that provide rates for travel to the selected destination
   (This activity is the *automatic discovery of services*.)
2. obtains rates from the discovered airline services and selects the one that offers a nonstop option at the lowest fare
   (Here, the application performs an *automatic invocation of services*.)
3. discovers a set of rental car services at the selected destination
   (This task is accomplished by a second *automatic discovery of services.*)
4. queries the rental car services for availability and rates and selects the one with the lowest daily rate, given the travel dates and the arrival time of the selected flight
   (In this *automatic composition and invocation of services,* the output from the travel service is used as input to the rental car service.)
5. displays the selected itinerary to the user and asks for confirmation

The following sections describe the OWL-S elements that support the dynamic discovery, composition, and invocation of services like that illustrated by this travel arrangements scenario.

2.1 OWL Web Ontology Language

OWL-S is based on ontologies of objects and concepts defined using the OWL Web Ontology Language [W3C 04a, W3C 04b]. OWL ontologies describe the hierarchical organization of ideas in a domain in a way that can be parsed and understood by a software program. OWL ontologies are similar to an object-oriented class hierarchy in a software program. Consider the ontology of fruits and vegetables shown in Figure 1. The classification of *Berry* could be considered a subclassification of *Fruit*, and *Strawberry* could be introduced in the ontology as an instance of the *Berry* classification with characteristics such as its color

being red and its flavor being sweet. The ontology could further be extended to make *Fruit* and *Vegetable* subclassifications of *Food*.

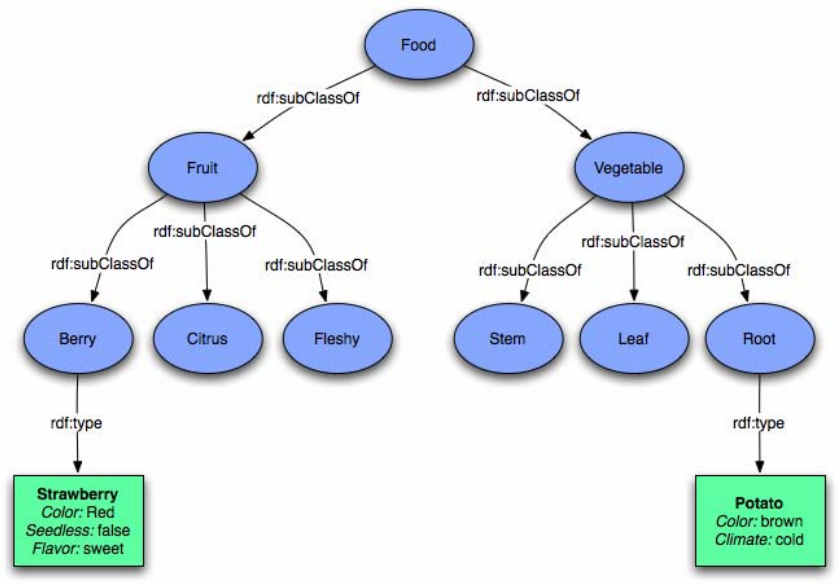

Figure 1: An Incomplete Example of an Ontology of Foods

2.2 OWL-S Ontologies

In its simplest form, an OWL-S ontology defines elements that describe the interface a service provides to the "outside world." Currently, OWL-S ontologies can only be defined for Web Services.[4] As specified in the WSDL document that describes them, the inputs and outputs of a service are mapped to classifications in an OWL ontology. For example, a service that provides online market pricing for berry farmers might take a berry name as input and produce a dollar-per-pound value as output. An OWL-S ontology for the berry pricing service could map the berry name input to the *Berry* classification from the *Fruit* ontology and the dollar value output to a *Currency* classification from a *Money* ontology. By doing so, any application that understands what berries and currency are in terms of their respective ontologies could make use of this service without specific knowledge of its service interface.

[4] The *Web Services* approach to implementing an SOA involves (1) service interfaces described using Web Services Description Language (WSDL), (2) content transmitted using Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP), and (3) Universal Description, Discovery, and Integration (UDDI) optionally used as the directory service for the discovery of services [Lewis 04].

2.3 OWL-S Description Elements

A service in OWL-S is described by means of three elements, as shown in Figure 2:

1. The *Service Profile* describes what the service does. It explains what the service accomplishes, details limitations on its applicability and quality of service, and specifies requirements the service requester must satisfy to use it successfully. This information is used by consumers during the discovery of the service.
2. The *Service Process Model* describes how to use the service. It details the semantic content of requests, the conditions under which particular outcomes will occur, and, where necessary, the step-by-step processes leading to those outcomes.
3. *The Service Grounding* specifies the details of how to access/invoke a service. It includes communication protocol, message formats, serialization techniques and transformations for each input and output, and other service-specific details such as port numbers used in contacting the service [Martin 04a].

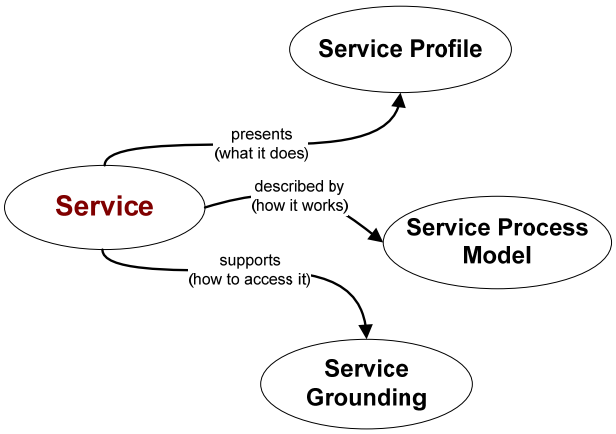

Figure 2: OWL-S Service Description Elements

2.4 OWL-S Discovery and Execution Elements

By itself, OWL-S is a language for the markup of Web Services. It becomes useful when there are tools to exploit Web Services described using OWL-S constructs.

An example of an OWL-S toolkit is CMU's OWL-S Development Environment (CODE), created by Carnegie Mellon University's Intelligent Software Agents Lab [Srinivasan 05, ISAL 04]. CODE supports the complete OWL-S Web Services development process—from the generation of OWL-S descriptions (from Java code or WSDL documents) to the

deployment and registration of the service. CODE is implemented as an Eclipse plug-in that supports activities for service providers and client application developers.[5]

In addition to tools for the description of services, CODE includes the OWL-S Matchmaker and the OWL-S Virtual Machine (VM) elements.

The OWL-S Matchmaker serves as a "catalog" of services defined using OWL-S. Service providers register OWL-S descriptions of services with the OWL-S Matchmaker. Client applications can query the OWL-S Matchmaker with an ontological description of the desired inputs and outputs. The OWL-S Matchmaker matches the request with its catalog of services and returns a ranked list of services that most closely match the request.

The OWL-S VM is used to invoke services using OWL-S. After the client application selects a service from the ranked list of services, it formulates its request using the format specified by the OWL ontology and sends the request to the OWL-S VM. Using Extensible Style Language Transformations (XSLT)[6] present in the Service Grounding element, the OWL-S VM reformats the request to match the format required by the service. Then, it invokes the service on behalf of the client. When it receives a response to that step, the OWL-S VM uses another XSLT transformation in the Service Grounding element to reformat the response into a format matching that of the ontology. Finally, the OWL-S VM sends the response back to the client application. In this manner, the client application does not need to know anything about how to interact with the actual service; the OWL-S VM acts as a mediator for the request and response.

Both the OWL-S Matchmaker and the OWL-S VM elements have an Application Programming Interface (API) for Java applications to discover and invoke services.

[5] Eclipse is an open source community engaged in providing a vendor-neutral software development platform. Read more about Eclipse at http://www.eclipse.org/.

[6] In its simplest definition, *XSLT* is a language used to transform XML documents into other XML documents [W3C 99b]. Currently, XSLT is the only type of transformation supported by the OWL-S VM.

2.5 Service Provider Perspective on the OWL-S Development Model

Using elements from CODE, we describe in this section the development process from the service provider perspective. The process is presented graphically in Figure 3. (A description of the service provider perspective in a dynamic discovery and invocation environment is provided in Section 2.7.)

1. *Create the OWL-S Profile*. The first step in the development process is the creation of an OWL-S description—called the OWL-S Profile—for the service. The OWLS-Profile can be generated directly from Java code or a WSDL document. The *Java2OWL-S Converter* component takes Java code containing the methods to be exposed as operations within the service and generates the OWL-S Profile. The *WSDL2OWL-S Converter* component takes a WSDL file for the service and generates the OWL-S Profile. In addition to the Service Profile, Service Process Model, and Service Grounding elements described in Section 2.3, Concept and Service files are generated as part of the OWL-S Profile. The Concept File is an OWL ontology that describes the concepts used by the inputs and outputs of the OWL-S processes and profile. The Service File is an OWL description of what the actual service does and what happens when the actions provided by the service are executed.
2. *Edit the OWL-S Profile*. Using the OWL-S Editor, the service provider can add details such as special data transformations in the Service Grounding element, control flow and data flow information in the Service Process Model element, and nonfunctional parameters (e.g., quality rating) in the Service Profile element.
3. *Validate the OWL-S Profile using the OWL-S Verifier component.* This step is optional.
4. *Deploy the service and its OWL-S Profile*. The actual service has to be deployed on a server where it is accessible to the client applications. Its OWL-S Profile has to be deployed on a public server where it is accessible by the OWL-S Matchmaker.
5. *Register the service with the OWL-S Matchmaker*.
    a. The OWL-S Profile is converted to its corresponding UDDI advertisement using the *OWL-S2UDDI Converter* component.
    b. The UDDI advertisement is registered with the OWL-S Matchmaker.

Additionally, CODE has the ability to perform test queries against the OWL-Matchmaker to make sure that the OWL-S Profile has been correctly created and registered.

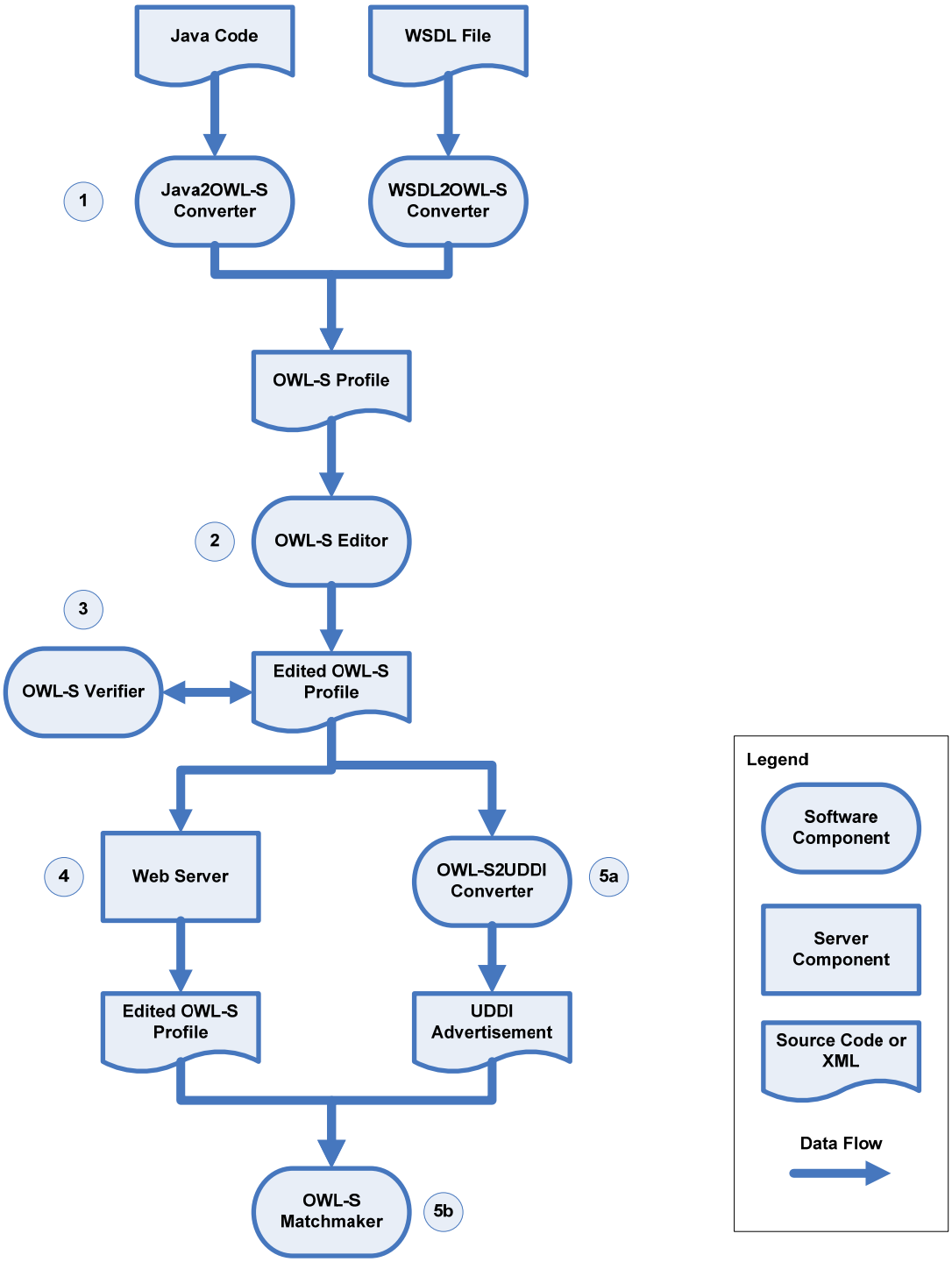

Figure 3: Service Provider Perspective on OWL-S Development using CODE

2.6 Application Developer Perspective on the OWL-S Development Model

From the client application developer perspective, the development process differs depending on whether the service discovery and invocation are done statically or dynamically. The steps for static discovery and invocation are listed below and presented graphically in Figure 4.

1. *Using the OWL-S Editor, create an OWL-S Request*[7] *that corresponds to the discovery query.* This OWL-S Request has the same format as the OWL-S Profile.
2. *Using the OWL-S Editor, register the OWL-S Request with the OWL-S Matchmaker*.
3. *Using the OWL-S Editor, query the OWL-S Matchmaker.* The OWL-S Request is used to perform the query against the OWL-S Matchmaker. The Matchmaker will return a list of OWL-S Profiles ranked according to how well they match the OWL-S Request.
4. *Select a service*. The client application developer selects the "best fit" service from the list of OWL-S Profiles.

At this point of the process, the client application developer has to write code that uses the OWL-S VM API to perform the following operations:

5. *Load the selected OWL-S Profile into the OWL-S VM.*
6. *Invoke the service.* The client application developer has to provide the proper OWL inputs and process the OWL outputs corresponding to the service response.

[7] CODE also calls this request or query a profile because it has the same format as the OWL-S Profile. To avoid confusion, we refer to it as the OWL-S Request in this report.

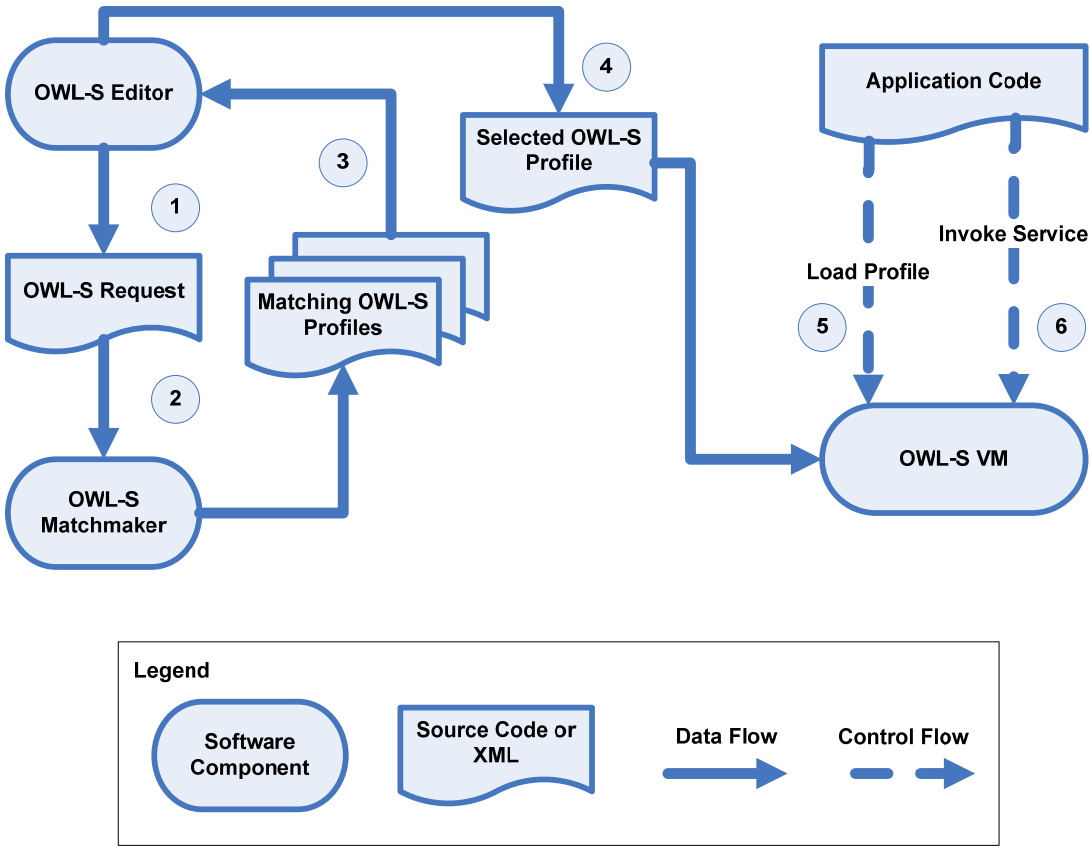

Figure 4: Application Developer Perspective on OWL-S Development using CODE (Static Discovery and Invocation)

In a dynamic service discovery and invocation setting, both the discovery (steps 1–4) and invocation (steps 5 and 6) tasks listed above are performed by the application without any user intervention. That is, the OWL-S Matchmaker API is used to perform steps 1–4, instead of the OWL-S Editor, and the OWL-S VM API is used to perform steps 5 and 6.

An application that can dynamically discover and invoke services would be quite complex, as will be explained in Section 5.3. The OWL-S runtime model explained in the next section also helps to illustrate the complexity of such an application.

2.7 The OWL-S Runtime Model

Figure 5 depicts the sequence of events that would occur at runtime in a dynamic discovery and invocation environment of services using OWL-S and registered with the OWL-S Matchmaker. In that environment, all of the following steps would be performed by the client application at runtime.[8]

1. *Create the OWL-S Request and load it into the OWL-S Matchmaker*.
2. *Query the Matchmaker.* The client application queries the OWL-S Matchmaker to find services that best match the OWL-S Request. The OWL-S Matchmaker returns a list of ranked matching OWL-S profiles.
3. *Select a service from the list of matching OWL-S profiles.*
4. *Load the selected OWL-S Profile into the OWL-S VM.*
5. *Invoke the Web Service.*
    a. The client application formats its request using OWL and sends it to the OWL-S VM.
    b. The OWL-S VM invokes the service on behalf of the client application, using the XSLT contained in the OWL-S Service Grounding element to convert it into the correct form.
    c. The OWL-S VM then translates the response returned from the service invoked into an OWL representation that is returned to the client application.

[8] As indicated in Section 2.5, steps 1 through 4 would occur during development time and not at runtime in a static discovery and invocation environment.

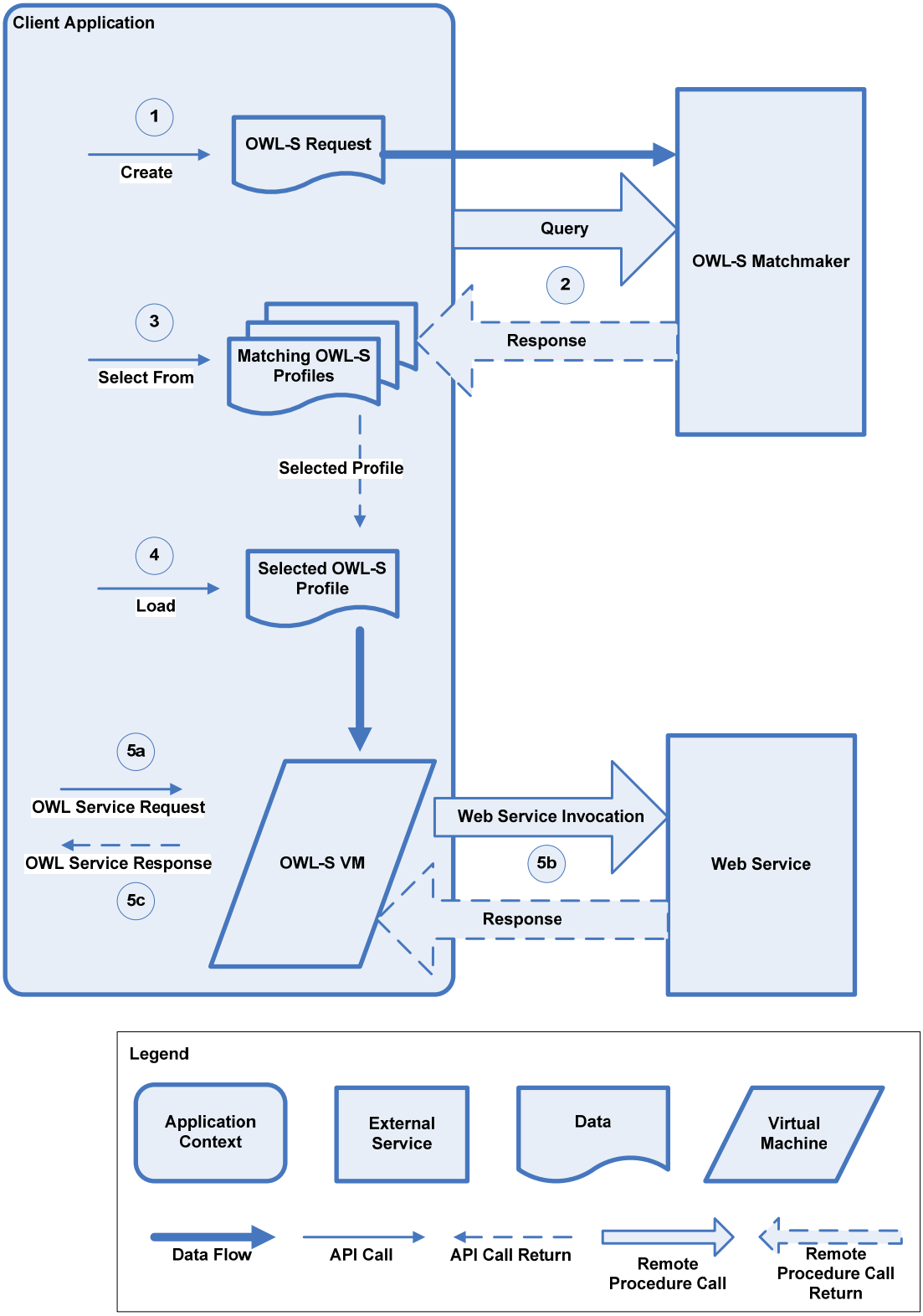

Figure 5: OWL-S Runtime Model

3 Using the Model Problem Approach

The model problem approach is a technique for evaluating software technologies. This approach involves (1) formulating hypotheses about the technology and (2) examining these hypotheses against very specific criteria through experimentation. The outcome of this two-stage approach is that the hypotheses are either sustained or refuted. The model problem approach has the advantage of producing very efficient and representative experiments that not only evaluate technologies within the context of their future use but also generate hands-on competence with the technologies [Wallnau 01].

A graphical representation of the model problem process is presented in Figure 6. The model problem process is part of a larger process for context-based technology evaluation. In this larger process, the context for the model problems is established and the expectations from the technology are captured [Lewis 05]. This model problem approach was used to evaluate the potential of OWL-S for the dynamic discovery, composition, and invocation of services.

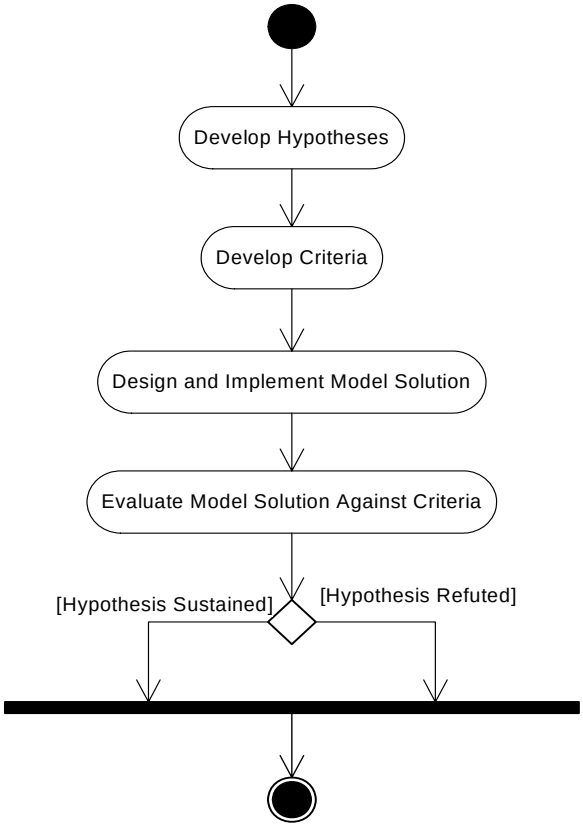

Figure 6: Model Problem Process for Technology Evaluation

3.1 Model Problem Context

The context selected for the model problem used in this study was map services, because they provide a valuable example of a real-world application of OWL-S for commercial and DoD organizations. Maps are available in a diverse selection of classifications and formats, and the coordinates and measurements used to describe them are distinct and meaningful. This level of information allowed us to create a deep and specific ontology to describe the knowledge space for the mapping services.

3.2 Evaluation Hypotheses

The first step in developing a model problem is to define hypotheses—claims about the technology in question that will be supported or refuted by the model problem. For OWL-S, the following hypotheses were defined:

1. OWL-S will allow the dynamic discovery of services by expected input/output and service description and provide the location of the service that constitutes the best match.
2. OWL-S will allow the dynamic composition of services when there is not a single service that can satisfy the query, but rather a sequence of services.
3. A client application that uses OWL-S for locating services will be able to invoke the highest ranked service (or set of services in the case of composition) automatically.

3.3 Evaluation Criteria

As Table 1 shows, each hypothesis was paired with the criteria (i.e., measurable statements of capability) used to determine whether the model solution sustains or refutes it.

Table 1: Hypotheses and Criteria for the OWL-S Model Problem

Hypothesis	Evaluation Criteria
1. OWL-S will allow the dynamic discovery of services by expected input/output and service description and provide the location of the service that constitutes the best match.	• When searching by input/output, the OWL-S Matchmaker will rank higher those services that are a closer match to the given input/output. • When searching by service description, the OWL-S Matchmaker will return only those services within the same domain (in this case, mapping).
2. OWL-S will allow the dynamic composition of services when there is not a single service that can satisfy the query, but rather a sequence of services.	The OWL-S Matchmaker will be able to find a sequence of two services where the output of one service is the input to another service, and the services as a set will match the given query.
3. A client application that uses OWL-S for locating services will be able to invoke the highest ranked service (or set of services in the case of composition) automatically.	If the client application finds a match, it will invoke the highest ranked service or set of services without any user intervention.

4 Designing and Implementing the Model Solution

The model solution was to be developed as a Java application that retrieved a specific type of map, given location coordinates expressed using different units. This problem allowed us to test discovery based on different map types and composition, when there is difference in the location coordinate systems.

4.1 Setting Up the OWL-S Development Environment

The first step in implementing the model solution was to select the development environment. It was essential to select tools for which direct and personal support was available during the development of the model problem.

Fortunately, the Intelligent Software Agents Lab at Carnegie Mellon University has played a critical role in the development of OWL-S from its very inception [ISAL 01]. This group developed CODE, described in Section 2.4. Thus, CODE was selected as the development environment, and support was graciously provided by Naveen Srinivasan of the Intelligent Software Agents Lab.

4.2 Selecting Potential Web Services

The next step was to select Web Services that could potentially be described and accessed using OWL-S. While doing so, there were several considerations kept in mind:

- The service should provide similar capabilities through different interfaces. Operations with different interfaces but similar ontological meanings would provide for more meaningful queries and results.
- The service should provide a number of simple operations that can be used to test the basic capabilities of OWL-S. These simple operations would be composable into more complex processes.
- A WSDL definition must be available for the service.
- There must be no cost to access the service.

Based on these criteria, Microsoft's TerraServer was selected as the service provider [Microsoft 05].[9] TerraService Web Service is freely available and provides a number of

[9] The Microsoft TerraServer Web site "stores aerial, satellite, and topographic images of the earth in an SQL database available via the Internet." TerraService is a "Web Service that provides programmatic access to the TerraServer database" [Barclay 02].

operations that perform simple tasks, such as converting between location coordinate units, and more complex operations to retrieve maps and photos of specified areas.

4.3 Creating the OWL Ontology

The next step in implementing the model solution was to create an OWL ontology to represent the mapping knowledge space. The ontology had to describe not only the different types of maps that a client might wish to access but also the different concepts—such as geographic locations and map scales—that describe the maps themselves.

Fortunately many complete and easy-to-use tools exist for creating OWL ontologies. The Protégé Ontology Editor was selected to develop the ontology [Stanford 06]. Protégé is an open-source ontology editor that allows knowledge providers to create knowledge bases easily via a graphical editor.

Using Protégé, an ontology was created that defines some of the core concepts related to mapping services—such as map types, datum to define map orientation, and types of locations. A simplified graphical representation of the ontology is presented in Figure 7 below. The complete OWL ontology can be found in Appendix A, along with a short explanation of some of the terms used.

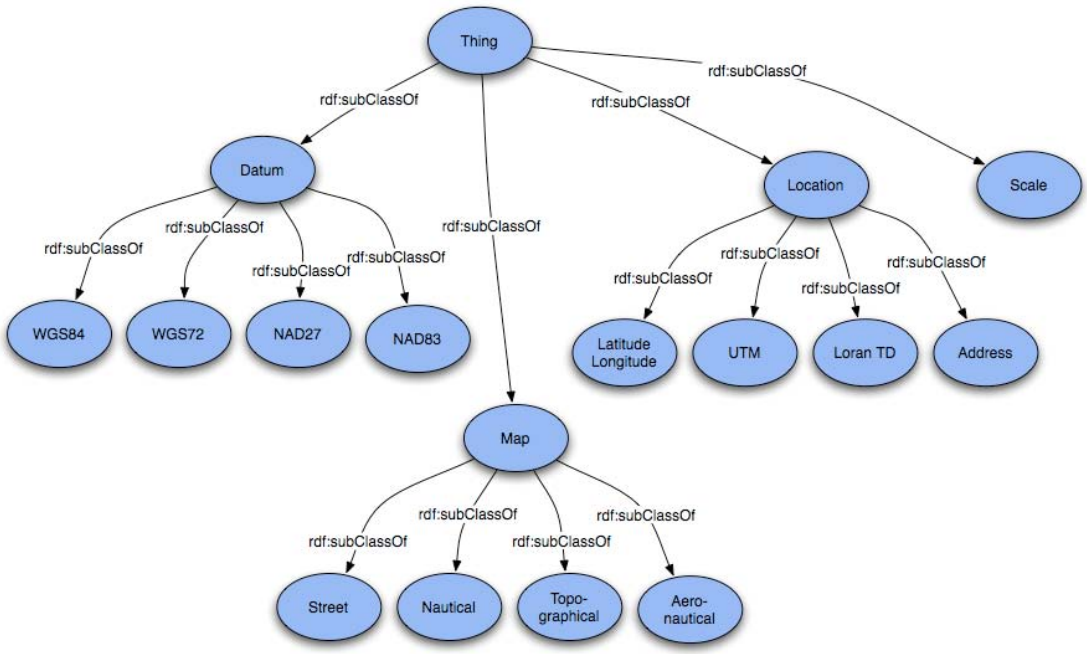

Figure 7: A Simplified Representation of an OWL Ontology for Maps[10]

[10] OWL ontologies can be thought of as object-oriented hierarchies. The *rdf:subClassOf* relationship, as defined in the Resource Description Framework (RDF) specification, is analogous to an inheritance relation. Classes that are subclasses of a particular parent class share a common semantic root with each other [W3C 99a].

4.4 Generating the OWL-S Profile for the Selected Web Service

Using the WSDL2OWL-S component included with CODE, the OWL-S Profile was generated for the TerraService Web Service; the result of that process is shown in Figure 8. Generating the OWL-S Profile involved several simple processes that performed operations, such as converting between location formats, and more complex processes that composed multiple service invocations. (For the Service Profile element generated, see Appendix B; for the Service Process Model element generated, see Appendix C; and for the Service Grounding element generated, see Appendix D.)

Using the OWL-S Editor to modify the OWL-S Profile, we mapped the input and output parameters of the operations to their counterparts in the OWL ontology created earlier. The Service Grounding element was modified to use XSLT style sheets to transform the inputs and outputs between their OWL representations and their service counterparts, because there was not a direct mapping.[11]

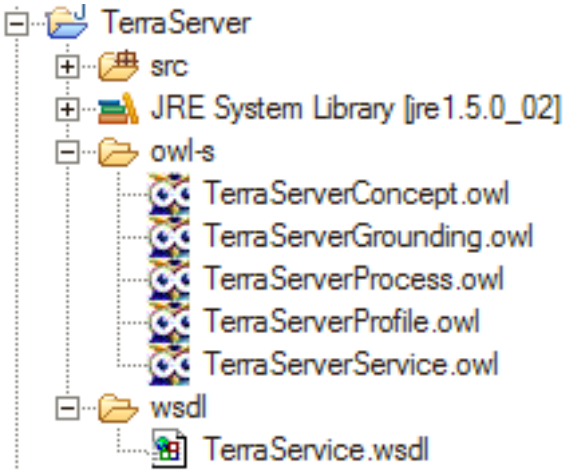

Figure 8: OWL-S Files Generated from a WSDL Service Definition

4.5 Publishing the OWL-S Profile on the Internet

In order to advertise it with the OWL-S Matchmaker, the OWL-S Profile must be accessible on the Internet. As a result, we published it on a Web server for the OWL-S Matchmaker to access.

4.6 Advertising the OWL-S Profile with the OWL-S Matchmaker

Using the OWL-S2UDDI component included with CODE, the OWL-S Profile was converted to a UDDI advertisement and published to an OWL-S Matchmaker maintained by Carnegie Mellon University. The OWL-S Matchmaker then registered the OWL-S Profile in

[11] As will be explained in Section 6.1, this modification was the consequence of creating services while unaware of the existence of an ontology.

its database and made it available for discovery. A screenshot from CODE is presented in Figure 9.

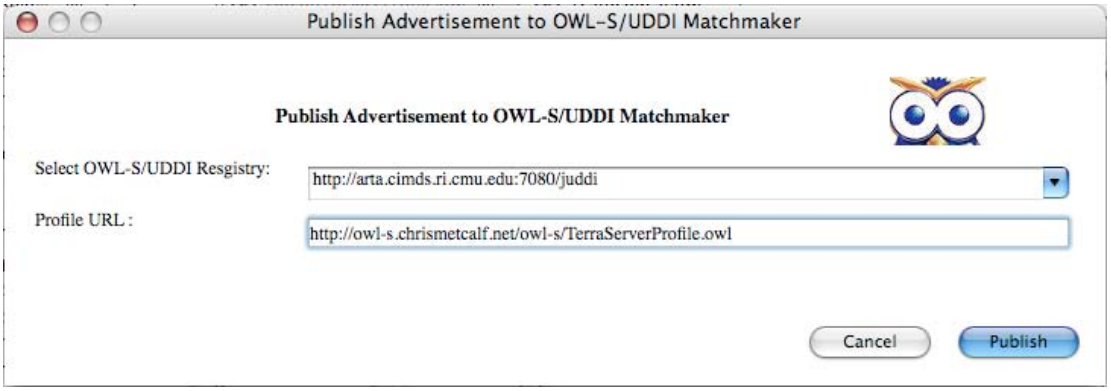

Figure 9: Publishing an Advertisement of an OWL-S Profile on the Matchmaker

4.7 Querying the OWL-S Matchmaker to Discover Services

Using the OWL-S Editor, an OWL-S Request was created, including the ontological descriptions of the inputs and outputs desired from a potential service. Like the OWL-S Profile, this OWL-S Request must be published to a Web server where it will be accessible to the OWL-S Matchmaker.

Using the client application included in CODE, the OWL-S Matchmaker was queried for services that most closely matched the OWL-S Request. The OWL-S Matchmaker successfully returned a set of services ranked by how well they matched the request. The OWL-S Matchmaker was able to return simple as well as composite services.

Now that we verified that the ontology had been correctly defined and that the OWL-S Matchmaker worked with the created requests and profiles, the next step was to initiate requests from a client application.

4.8 Developing a Java Application to Test the Hypotheses

Both the OWL-S Matchmaker and the OWL-S VM have APIs that can be used by Java applications to discover and invoke services. Unfortunately, it was impossible to build an application because of the many problems we encountered with the OWL-S VM.[12]

One initial problem was that the current implementation of the OWL-S VM does not support the Base64Binary datatype[13] that is necessary to retrieve a map image using the TerraService

[12] As a side note, the main developer for CODE left during this work and support was no longer available.

[13] The XML base64Binary data type represents Base64-encoded arbitrary binary data. This data type is used to embed binary data such as images into XML documents.

Web Service. This shortcoming limited the initial scope of the model solution simply to the return of a location's coordinates in different units.

Another problem involved dynamic service invocation. At runtime, the OWL-S VM uses the OWL-S Service Grounding element and the WSDL description of the service to transform the OWL Request into a SOAP-XML message that is understood by the TerraService Web Service. However, the OWL-S VM was unable to successfully transform the SOAP-XML response into a valid OWL entity.

With the help of the OWL-S VM developer, we discovered that the problem concerned the deserializing of XML results returned by the TerraService Web Service. The OWL-S VM uses a tool called Java Record Object Model (JROM) to serialize/deserialize XML documents "on the fly" [IBM 02]. In order to function properly, JROM requires all XML data elements to have the `xsi type` attribute; however, the TerraService Web Service does not return the type attribute of an element. Therefore, JROM fails to convert XML data elements automatically into Java objects that would then transformed into the respective OWL outputs. Unfortunately, given the time and resources available, we could not solve this problem. Consequently, it was not possible to invoke the TerraService Web Service from the Java client application. More detail on this experience is provided in Section 5.3.

In addition, we were unable to successfully query for a matching OWL-S Profile from our OWL-S Matchmaker client. Although we were able to successfully publish OWL-S Profiles for our TerraService Web Service, we were unable to retrieve any matching services from the Matchmaker itself. We suspect that the OWL-S Matchmaker client Java API was built upon a different version of the OWL-S Matchmaker libraries than the one used by the OWL-S Editor.

By this point in our investigation, the funding for OWL-S had been stopped, and the CODE developer was no longer available to help us debug the OWL-S Matchmaker client.

5 Evaluation

The last step in the model problem process is to evaluate the model solution against the criteria in order to determine whether the hypotheses are sustained or refuted. The results of this process are provided in the following sections. As stated previously, OWL-S is simply a markup language for Web Services; the tools that apply OWL-S make it useful. The observations that follow pertain to OWL-S as well as the tools that exploit its benefits.

5.1 Hypothesis 1 Results

As stated in Section 3.2, this was our first hypothesis: *OWL-S will allow the dynamic discovery of services by expected input/output and service description and provide the location of the service that constitutes the best match*.

This hypothesis is partially supported: OWL-S does allow the dynamic discovery of services based on the ontological classifications of expected input and output. Given a collection of properly defined OWL-S Service Grounding elements and a request describing the classifications of the desired inputs and outputs, an OWL-S Matchmaker can return an ordered list of services that most closely match a request. A client application can then select the service that best provides the desired capabilities.

OWL-S does not provide support for queries against service descriptions, however. In OWL-S, the functional description of the service is expressed in terms of the transformation produced by the service (inputs and outputs plus preconditions and postconditions). The Service Profile element information that is closest to service description is the category of a given service. Even so, the OWL-S Matchmaker does not yet provide support for querying for services based on their service descriptions. Therefore, it does not take advantage of this part of the Service Profile element while performing a match.

A more detailed description of additional findings follows.

5.1.1 Effective Dynamic Discovery Depends Upon a Well-Defined Ontology

In OWL-S, services are matched based on the ontological classifications of their inputs and outputs. For example, an application can specify that it wants to discover a service that takes a *Fruit* type as input and a *Price* as its output, and the OWL-S Matchmaker will help it find a service that does just that. In order to correctly classify services for use with OWL-S, an ontology must be established that sufficiently covers all of the types of inputs and outputs that might be available in the domain. Small variations in the type of input or output that a service provides can drastically change the ontological meaning. A weakly defined ontology

that only contains the loosely defined type *Foodstuff* would not help the above application find its fruit pricing service. On the other hand, ontologies that are too complicated will be more difficult to map efficiently to the "concrete" interface of a service. Ontologies must be deep enough (i.e., have enough levels) to cover these variations without sacrificing the quality of the fit of an ontological classification to the interface of a service.

5.1.2 Ideally, Service Providers within the Same Domain and Their Client Applications Should Share a Common Ontology

In a real-world context, (1) an ontology is created for the domain that the OWL-S Matchmaker is going to serve, (2) Web Services providers register their services by providing a description of the service in terms of this ontology in the Service Profile element and any necessary transformations between elements of the ontology and operation parameters in the Service Grounding element, and (3) client applications query the OWL-S Matchmaker in terms of this ontology to find matching Web Services.

In our model problem, we performed all of those tasks because we were acting in all the roles. Even though the ontology was as generic as possible for any mapping service, the truth is, to reduce the complexity of the transformations, we had more flexibility to adjust the ontology to be a good fit with the TerraService Web Service. Even though we did not try to extend the model problem to incorporate other mapping services, we anticipate that an extension would be difficult because we would need to

- provide potentially complex transformations for parameters of the Web Services that are not an exact match with terms in the ontology
- compensate for parameters that are not part of the ontology

These accommodations are more difficult for a client application working in a dynamic environment, as will be explained in Section 6.1.

In order to find better matches for services, developers for service providers and consumers in a domain should share a common ontology. In that way, services with common ontological inputs and outputs can be discovered using the same queries, even if the services have different interfaces. As a result, matches between requests and profiles would be more precise. Also, the amount of work required of service provider and client application developers to adapt to an existing ontology could be reduced. Creating a common ontology for service providers and consumers in a domain sounds utopian, but it might be the only path to dynamic invocation of services.

5.1.3 Categorization of Services by Service Description or Category May Not Enable Effective Service Discovery

Classifying services by description and/or category alone will not allow applications to use OWL-S effectively to perform dynamic service discovery. Indeed, it will certainly limit dynamic service invocation. Within a single type or classification of service, there are a vast

number of different combinations of input and output types. For example, an application cannot expect to simply query for *Map Provider* services and find a perfect fit for its particular need. If the only *Map Provider* service that was discovered took an address as input and the only knowledge available to the application is a latitude/longitude coordinate, that service is certainly not a good fit. What is the desired map type? What are the desired quality, scale, and resolution? What type of input can the application provide? If dynamic invocation is intended, then the application would have to deal with all possible input and output mappings and translations.

In order to find effective matches using OWL-S as a discovery mechanism, the ontological classifications of the input and output still need to be taken into consideration. Service description or category can be used as an additional qualifier to improve the results of a service query, but neither one can stand alone as a way to discover services.

5.2 Hypothesis 2 Results

The second hypothesis proposed in Section 3.2 was this: *OWL-S will allow the dynamic composition of services when there is not a single service that can satisfy the query, but rather a sequence of services.*

Like hypothesis 1, this second premise is partially supported. Using OWL-S, a client application can discover a sequence of independent services that could ultimately help satisfy its information goal. However, this dynamic composition of services is not provided by OWL-S itself—it is a capability that must be implemented at the client level. OWL-S simply provides for the discovery of services, not their automatic composition. In order to enable the applications themselves to compose collections of services into a more complicated operation, automatic composition would require more planning logic at the application layer. For example, an application that wants to find the price of a bushel of apples in Euros (instead of in U.S. dollars) could discover one service to retrieve the price of a bushel of apples in U.S. dollars and another to convert those dollars into Euros. But the OWL-S Matchmaker would not create that process flow automatically. The application would have to contain enough planning logic to discover those two independent services and compose them on its own.

However, the OWL-S Process Editor does support the manual composition of services into composite processes. When defining the process model of a service, composite services that perform multiple operations in a predefined order, including sequential execution and branching, can be created. *Dataflows* can also be defined to transfer some or all of the data from one operation to the next. In this manner, distinct services can be manually composed into complex sequences of operations to reach a predefined goal.

Figure 10 presents an example of a composite service *ConvertUTMToNearestPlace* being edited in the CODE Process Editor, along with its data flow and the simple services it comprises. Two simple services—*ConvertUTMToLatLon* and

ConvertLatLonToNearestPlace—are composed into the *ConvertUTMToNearestPlace* complex process flow. The latitude and longitude output of *ConvertUTMToLatLon* (*LatLon_Out*) is redirected to the input parameter of *ConvertLatLonToNearestPlace* (*LatLon_In*) in order to create one process flow that converts a Universal Transverse Mercator (UTM) location into the nearest place name to that geographic point.

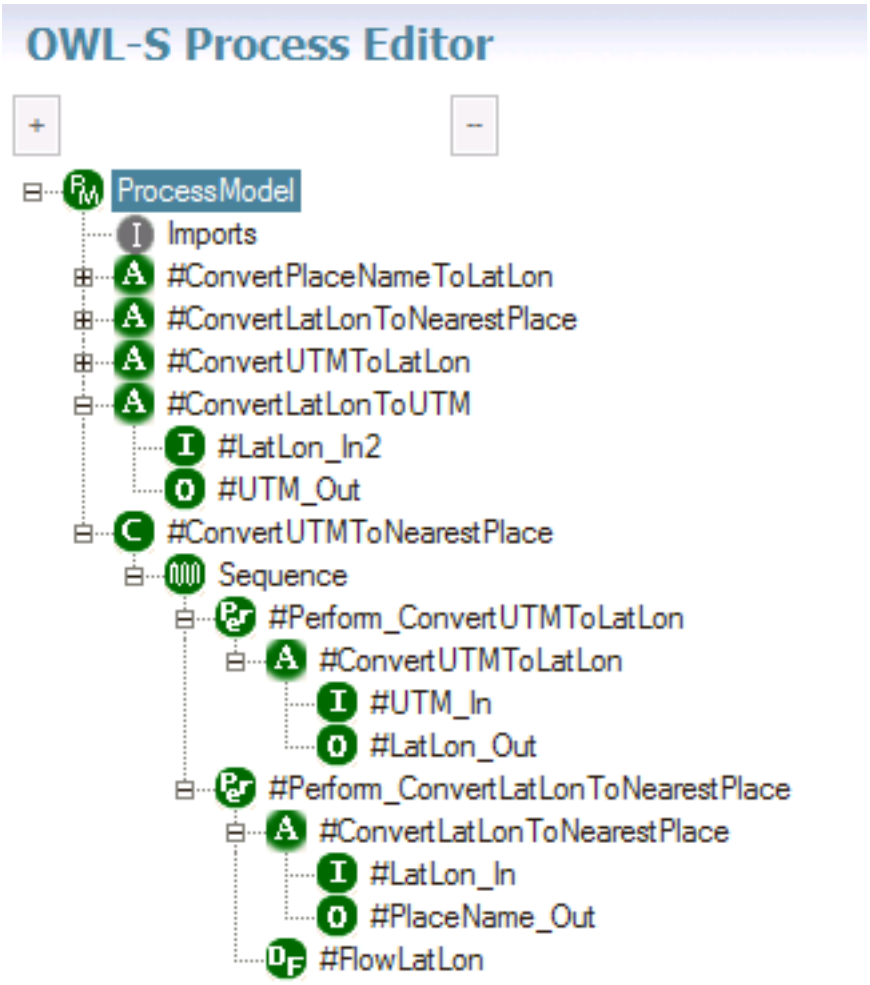

Figure 10: Simple and Composite Services Represented in the OWL-S Process Editor

5.3 Hypothesis 3 Results

The third hypothesis introduced in Section 3.2 was this: *A client application that uses OWL-S for locating services will be able to invoke the highest ranked service (or set of services in the case of composition) automatically.*

Again, this hypothesis is partially supported. An application using OWL-S for locating services dynamically will be able to invoke the highest ranked service, as long as it is capable of providing all of the necessary information in terms of the ontology. As an example,

suppose there is a client application that wants to find a mapping service that takes a latitude/longitude coordinate as input and provides a weather map as output. If there are no perfect matches to this request, the best match might actually be a service that requires both a latitude/longitude coordinate *and* a desired forecast time as its input. If the client application is unable to respond to this slightly different input or does not have access to this additional information, it will not be able to invoke the service automatically.

Overall, dynamic invocation of services is a difficult task that seems almost impossible for several reasons. As explained previously, in order to automatically invoke a service, a client application must have at its disposal all the information that it needs to provide to the service or the means by which to acquire that information. This need becomes more important with OWL-S services. In order to invoke an OWL-S service, the application must not only have access to that data but also must be able to provide it in the proper OWL format before querying the service.

Although this formatting task can be done in a static way, much like what is now done for Web Services, a better way is to make an application aware of the data it has access to and the ontological meaning of that data. If done properly, this method will enable an application to access services that were not available when it was originally developed, since it will be able to build new types of queries to access new services. In addition, through goal-based planning, a service could compose—"on the fly"—other, simple services into more complex sequences of services to work towards a particular information goal. However, perfect goal-based planning requires an artificial intelligence engine and constant probing of the services available in the OWL-S Matchmaker, which is not practical or efficient in most applications.

The problem encountered in the development of the Java application regarding the incompatibility of the TerraService WSDL file and the expectations of the JROM tool used by the OWL-S VM, as explained in Section 4.8, is another issue that often complicates the implementation of Web Services. Most of the standards used in Web Services are emerging—they contain elements that are subject to multiple interpretations, causing interoperability problems between client applications and Web Services.

That incompatibility makes automatic invocation even harder. In the case of static invocation, incompatibility could be easily resolved—by asking the service provider to run the WSDL file against the WS-I Basic Profile, for example.[14] As a result, the Basic Profile Testing Tool would have pointed out the inconsistency that caused the problems in the OWL-S VM. Obviously, this fix is unacceptable in a dynamic invocation environment because the OWL-S VM has expectations on the WSDL files that describe the Web Services. In a dynamic invocation environment, a potential overall solution to the incompatibility problem would be to test the WSDL file before the service is advertised in the OWL-S Matchmaker. While this would add an additional step for the service provider, it would make the process easier for the

[14] The *Basic Profile* is a set of nonproprietary Web Services specifications, along with clarifications, refinements, interpretations, and amplifications of those specifications that promote interoperability [WS-I 04].

application developer. This test would have to be added to the OWL-S Matchmaker registration process.

6 Experience with OWL-S

Experimentation with OWL-S was a positive experience in the exploration of OWL-S itself, particularly the use of ontologies as a means to enable dynamic discovery, composition, and invocation of services. The immaturity of the OWL-S toolkit has proven frustrating and accentuates the need for more funding and research in this area, because the dynamic use of Web Services is a very complicated task. What follows in this section is a sample of the knowledge gained from this experience.

6.1 Successful Integration Using OWL-S Will Require a Change in Development Paradigms

Currently, Web Services implementations are often a direct result of adding an additional interface to existing application functionality, as shown in Figure 11. Typically, a system with a capability at the local level is made available as a service through Web Services for distributed applications. Application developers use the WSDL document that describes the service in order to use its functionality. Therefore, services are developed using an “inside-out" approach: functionality is developed for specific applications, and later it is decided that this functionality should be a service available through Web Services.

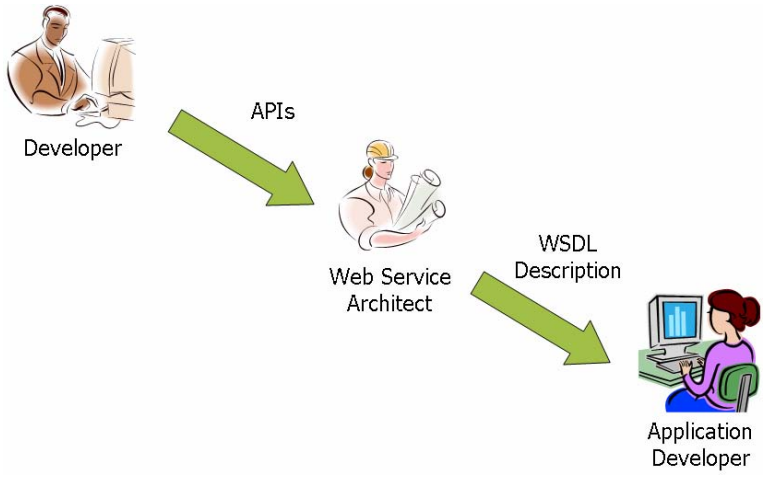

Figure 11: The Code-Driven Approach

If the organization then wishes to use OWL-S to represent its Web Services, it would fit a process model to the Web Services interfaces and eventually create an ontology to represent the information services provided. This approach is called the code-driven approach [Srinivasan 05]. However, the code-driven approach is tedious and fraught with errors. Because the Web Services interfaces were initially created without consideration of an

ontology or the way in which the services might be used, the task of mapping service inputs and outputs to their counterparts in the ontology is very difficult. The mappings are often very fragile and depend on assumptions about the expected data. In addition, there are often parameters—such as user names and passwords—that do not fit with the rest of the ontology. That leaves the application and Web Services developers with few options—sacrifice the quality of the ontology by modifying it to include the new information or lose flexibility by hard-coding the values into the Service Grounding element.

Moving to a semantic representation of services requires thinking and working in another way—that is, taking a model-driven approach [Srinivasan 05]. As Figure 12 illustrates, developers must first work with *Domain Experts* to create an ontology of the knowledge domain in which the services will reside. Then, the *OWL-S Architect* can create a process model that describes how clients will interact with the services. Only then can the programmatic interfaces for the Web Services be defined and registered with an *OWL-S Provider*. The OWL-S Provider is responsible for maintaining a OWL-S Matchmaker service to which Web Services providers can publish OWL-S descriptions of their services. The OWL-S Provider would then provide access to the OWL-S Matchmaker for valid consumers of those Web Services. Application developers would use the same ontology to build the requests for matching Web Services.

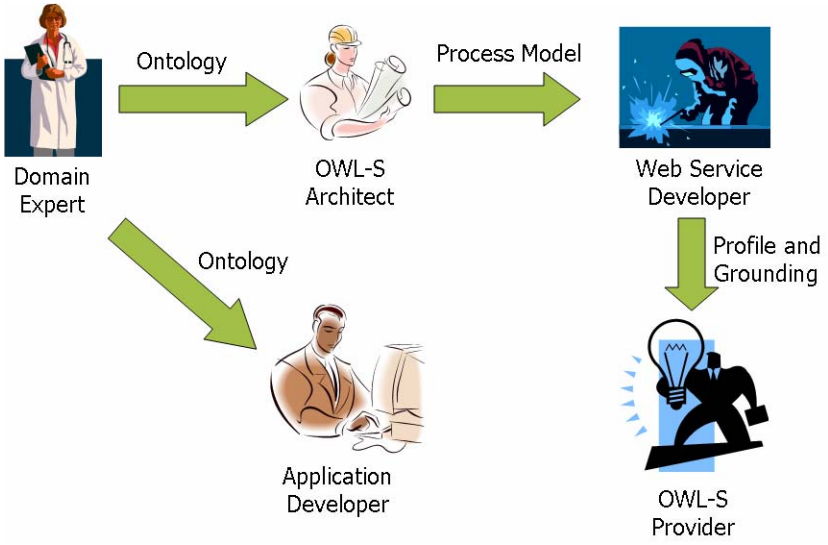

Figure 12: The Model-Driven Approach

By working in this way, Web Services interfaces can be designed that map more directly to the ontologies of their OWL-S descriptions. A deep, complete, and extensible ontology can be created that accurately represents the knowledge space in which the service resides, rather than simply the interface of the service. The model-driven approach will allow not only for better matches to service queries, but will also make it easier to add new services in the future.

6.2 Tool Support for OWL-S is Immature

OWL-S is no longer an “infant,” but it is still growing. Although many tools are currently under development, they are immature for the most part. Most of the technologies are designed primarily as proof-of-concept research tools and are not meant for production use. CODE is the only cohesive toolset for OWL-S development. With time and industry support, it is likely that this situation will improve. But at the current moment, OWL-S tool support is lacking in both depth and quality.

This problem is the usual “chicken-and-egg” situation between research and tools. If there are no tools that can be used with OWL-S, its adoption will be very difficult. If there are no opportunities to use OWL-S in real environments, research to develop tools cannot make real progress. As an example, whenever CODE was taken out of the demonstration context, problems arose. If the CODE developers had more access to real users, they would benefit from a feedback loop, helping OWL-S and its potential users.

6.3 OWL-S is not a “Drop-In” Solution

Creating a semantics layer based on OWL-S to enable dynamic systems composition requires that client applications and services be designed with OWL-S and the service ontology in mind.

Pursuing OWL-S as a solution for interoperability requires careful examination of whether the system needs really warrant the use of OWL-S. Introducing OWL-S into an existing system requires that significant resources be expended to convert client applications, develop ontologies, and map services. Therefore, if the context for the system is not one in which diverse services need to be dynamically added, removed, and discovered—which is where the greater benefits of OWL-S lie—it may not be cost-beneficial to introduce OWL-S into the system.

6.4 Semantics is Not Exempt from the Abundance of Standards

These four specifications related to Semantic Web Services were submitted to the World Wide Web Consortium (W3C) in 2004–2005.

1. OWL-S [W3C 04c]
2. Semantic Web Services Framework (SWSF) [W3C 05a]
3. Web Service Semantics (WSDL-S) [W3C 05b]
4. Web Service Modeling Ontology (WSMO) [W3C 05c]

For the most part, it is unclear now whether these specifications compete against, complement, or supersede one another. There are ongoing integration efforts. For instance, WSMO uses WSDL-S as its Service Grounding mechanism. Also, there is a W3C Semantic

Web Services Interest Group that is chartered with discussing the integration of this work [W3C 03].

A clear understanding of the relationship among these four submissions and the types of problems in which they can be applied is necessary for the adoption of Semantic Web Services in general. We do not believe it is a matter of deciding who the winner is, but rather how these different efforts can collaborate in producing a standard that encompasses the benefits of each or provides clear instructions for use and integration into semantic solutions.

6.5 The Use of Semantics is Valuable—if Only for Service Classification and Description

Although fully dynamic discovery, composition, and invocation of services is still not possible (or at least extremely difficult) with current technologies, the use of semantics is still of value. The classification and description of services using ontologies can enhance service repositories used within an organization or community of interest. An organization can require service developers to ontologically classify their services before placing them in the service repository. Application developers can then query the repository for desired functionality. This can greatly promote reuse and reduce development time because service discovery is easier and efficient, even if not fully automatic.

An intermediate scenario between dynamic and static is also feasible with current technology, where services are discovered at runtime, but user intervention is required to select a service and provide the required information that is not accessible by the application.

7 Conclusions and Request for Feedback

The model problem experience has proven that regardless of its contributions to the semantics community, OWL-S is not ready to support the dynamic discovery, composition, and invocation of services, mostly due to the scarcity of tool support. Nonetheless, despite its problems, OWL-S has tremendous potential if given the proper resources and opportunities. Being able to define the inputs and outputs of a service in terms of an ontology is a huge step towards dynamic discovery, composition, and invocation without user intervention. Unfortunately, funding—which is what makes technologies real—has stopped for OWL-S tool development. Additional investment in completing and debugging CODE (or other tools that are as further along as CODE) would demonstrate the feasibility of the technology and encourage potential adoption from industry that in turn would provide feedback for further improvements.

With development, OWL-S could enable applications to dynamically discover, compose, and invoke new services to solve problems and gain information in ways that were not available when those applications were created. OWL-S has the capability to embed semantic meaning into the collections of services available in services-oriented computing environments, which will allow applications to be developed without knowledge of specific services that may or may not be available. That capability could also make SOAs more robust and flexible. Collections of services that are defined using OWL-S would allow applications to be tolerant of faults in a dynamic and transparent way by simply accessing other services with similar semantic meanings. OWL-S could also enable services to be registered and removed at runtime dynamically without causing downtime in client applications.

The ISIS team that is investigating OWL-S and other technologies using the model problem approach is interested in feedback from and collaboration with the communities that are considering technologies for interoperability. In addition to OWL-S and MDA, the ISIS team is looking at Web services, Open Grid Service Architecture (OGSA), and other standards and technologies. Write to the ISIS team at isis-sei@sei.cmu.edu.

Appendix A OWL Ontology for the Mapping Domain

The OWL ontology detailed in this Appendix represents the subset of the mapping knowledge space covered by the model problem, as represented in simplified form in Figure 7. The ontology describes not only the types of maps that a client might wish to access but also the different concepts that describe the maps themselves, such as geographic locations and map scales. The ontology would be used by the service provider to describe the inputs and outputs of a service and by the application developer to construct queries against the OWL-S Matchmaker.

Definition of Navigational Terms

A map's *datum* is the grid of reference used when a map was surveyed or created. The subclasses of Datum (WGS and NAD) refer to four of the most common map datum used for the 1972 and 1984 World Geodetic System surveys and the 1927 and 1983 North American Datum Surveys. The subclasses of *Location* also refer to four common location coordinate systems: (1) UTM, which is commonly used by the military; (2) Loran Time Differential (Loran TD), a grid system sometimes still used in nautical navigation; (3) Latitude/Longitude coordinates; and (4) street address coordinates.

Description of Code Example

The code example below contains these sections:

- *general information*
  This information includes such items as the namespaces used in the file and a description of the ontology.
- *an area resembling an object-oriented hierarchy (under the* <!-- **Class Hierarchy** --> *heading)*
  The elements in this section start with the `owl:Class` tag. A class represents a concept within the ontology. As an example, the `owl:Class rdf:about="#Map"` tag represents the class declaration for the Map type. The `Nautical` map type represented by `owl:Class rdf:about="#Nautical"` is a subclass of `Map` and therefore inherits all its properties.
- *description of the relationships between classes (under the* **<!-- Object Properties Representing Relationships Between Classes -->** *heading)*
  The elements in this section start with the `owl:ObjectProperty` tag. An `ObjectProperty` represents relationships between classes. For example, the

`owl:ObjectProperty rdf:ID="has_scale"` tag defines a relationship called `has_scale` between the class `Map` and the class `Scale`.

- *description of the relationships between a class and an XML Schema Datatype value or an RDF literal (under the* <!-- **Datatypes Properties Associating Data Types to Classes** *--> heading)*
  The elements in this section start with the `owl:DatatypeProperty` tag. Each element defines the type of data that can be associated to a particular class, such as strings, dates, numbers, specific values, and ranges. For example, the `owl:DatatypeProperty rdf:ID="zip_code"` defines a property called `zip_code` that is shown by individuals in the class `Zip_Code` and is of type `int`.

Simplified Code Example of OWL Ontology

A full OWL ontology would be much richer and more complete than that represented by this simple example.[15]

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns="http://www.owl-ontologies.com/unnamed.owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology rdf:about="Map Ontology for OWL-S Model Problems">
    <owl:imports rdf:resource="http://purl.org/dc/elements/1.1/"/>
  </owl:Ontology>

  <!-- Class Hierarchy -->
  <owl:Class rdf:ID="Bathymetric">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Nautical"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Satellite">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Map"/>
    </rdfs:subClassOf>
  </owl:Class>
```

[15] This example was created using Protégé (with OWL plug-in 2.2, Build 307). For more information on the Protégé-OWL Editor, go to http://protege.stanford.edu. All URLs listed in the code example were valid when the example was prepared.

```
<owl:Class rdf:ID="Ground">
  <rdfs:subClassOf rdf:resource="#Map"/>
</owl:Class>
<owl:Class rdf:ID="NAD_83">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Datum"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="#Nautical">
  <rdfs:subClassOf rdf:resource="#Map"/>
</owl:Class>
<owl:Class rdf:ID="Resolution"/>

<owl:Class rdf:ID="NAD_27">
  <rdfs:subClassOf rdf:resource="#Datum"/>
</owl:Class>
<owl:Class rdf:ID="Scale"/>
<owl:Class rdf:ID="Cloud">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Weather"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Surface">
  <rdfs:subClassOf rdf:resource="#Nautical"/>
</owl:Class>
<owl:Class rdf:ID="US_Street_Address">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Location"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Wind">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Weather"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Heightmap">
  <rdfs:subClassOf rdf:resource="#Satellite"/>
</owl:Class>
<owl:Class rdf:ID="Topographic">
  <rdfs:subClassOf rdf:resource="#Ground"/>
</owl:Class>
<owl:Class rdf:about="#Weather">
  <rdfs:subClassOf rdf:resource="#Map"/>
```

```
</owl:Class>
<owl:Class rdf:ID="Street">
  <rdfs:subClassOf rdf:resource="#Ground"/>
</owl:Class>
<owl:Class rdf:ID="Military_Grid_Reference_System">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:ID="Loran_Time_Differential">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:ID="WGS_84">
  <rdfs:subClassOf rdf:resource="#Datum"/>
</owl:Class>
<owl:Class rdf:ID="Thermographic">
  <rdfs:subClassOf rdf:resource="#Satellite"/>
</owl:Class>
<owl:Class rdf:ID="Zip_Code">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:ID="Temperature">
  <rdfs:subClassOf rdf:resource="#Weather"/>
</owl:Class>
<owl:Class rdf:ID="WGS_72">
  <rdfs:subClassOf rdf:resource="#Datum"/>
</owl:Class>
<owl:Class rdf:ID="Flight_Paths">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Aeronautical"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Barometric">
  <rdfs:subClassOf rdf:resource="#Weather"/>
</owl:Class>
<owl:Class rdf:about="#Aeronautical">
  <rdfs:subClassOf rdf:resource="#Map"/>
</owl:Class>
<owl:Class rdf:ID="Latitude_Longitude">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:ID="Photographic">
  <rdfs:subClassOf rdf:resource="#Satellite"/>
</owl:Class>
<owl:Class rdf:ID="Radar">
```

```
  <rdfs:subClassOf rdf:resource="#Weather"/>
</owl:Class>
<owl:Class rdf:ID="Universal_Transverse_Mercator">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>

<!-- Object Properties Representing Relationships Between Classes -->
<owl:ObjectProperty rdf:ID="has_scale">
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Map Scale</rdfs:comment>
  <rdfs:domain rdf:resource="#Map"/>
  <rdfs:range rdf:resource="#Scale"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="represents">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="represented_by"/>
  </owl:inverseOf>
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >A map represents a location</rdfs:comment>
  <rdfs:range rdf:resource="#Location"/>
  <rdfs:domain rdf:resource="#Map"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#represented_by">
  <rdfs:domain rdf:resource="#Location"/>
  <owl:inverseOf rdf:resource="#represents"/>
  <rdfs:range rdf:resource="#Map"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="has_resolution"/>

<!-- Datatypes Properties Associating Data Types to Classes -->
<owl:DatatypeProperty rdf:ID="zone_number">
  <rdfs:domain rdf:resource="#Universal_Transverse_Mercator"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="zone_quadrant">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >north</rdf:first>
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >south</rdf:first>
```

```
          <rdf:rest rdf:parseType="Resource">
            <rdf:rest rdf:parseType="Resource">
              <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax
               -ns#nil"/>
              <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >west</rdf:first>
            </rdf:rest>
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >east</rdf:first>
          </rdf:rest>
        </rdf:rest>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
  <rdfs:domain rdf:resource="#Universal_Transverse_Mercator"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="zip">
  <rdfs:domain rdf:resource="#US_Street_Address"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="longitude_east_west">
  <rdfs:domain rdf:resource="#Latitude_Longitude"/>
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:rest rdf:parseType="Resource">
          <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
               ns#nil"/>
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >west</rdf:first>
        </rdf:rest>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >east</rdf:first>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="latitude_degrees">
  <rdfs:domain rdf:resource="#Latitude_Longitude"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="map_unit">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
```

```
  <rdfs:domain rdf:resource="#Scale"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="street_Address">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#US_Street_Address"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="city">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#US_Street_Address"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="longitude_degrees">
  <rdfs:domain rdf:resource="#Latitude_Longitude"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="real_world_unit">
  <rdfs:domain rdf:resource="#Scale"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="easting">
  <rdfs:domain rdf:resource="#Universal_Transverse_Mercator"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="northing">
  <rdfs:domain rdf:resource="#Universal_Transverse_Mercator"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="depth_unit">
  <rdfs:domain rdf:resource="#Surface"/>
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >fathoms</rdf:first>
          <rdf:rest rdf:parseType="Resource">
            <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
             ns#nil"/>
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >meters</rdf:first>
          </rdf:rest>
        </rdf:rest>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >feet</rdf:first>
```

```
        </owl:oneOf>
      </owl:DataRange>
    </rdfs:range>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="longitude_minutes">
    <rdfs:domain rdf:resource="#Latitude_Longitude"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="street_name">
    <rdfs:domain rdf:resource="#US_Street_Address"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="zip_code">
    <rdfs:domain rdf:resource="#Zip_Code"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="latitude_north_south">
    <rdfs:domain rdf:resource="#Latitude_Longitude"/>
    <rdfs:range>
      <owl:DataRange>
        <owl:oneOf rdf:parseType="Resource">
          <rdf:rest rdf:parseType="Resource">
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >south</rdf:first>
            <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax
             -ns#nil"/>
          </rdf:rest>
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >north</rdf:first>
        </owl:oneOf>
      </owl:DataRange>
    </rdfs:range>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="latitude_minutes">
    <rdfs:domain rdf:resource="#Latitude_Longitude"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="state">
    <rdfs:domain rdf:resource="#US_Street_Address"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <Zip_Code rdf:ID="Map_RDFResource_56"/>
</rdf:RDF>
```

Appendix B Service Profile for the TerraServer OWL-S Service

The OWL-S Service Profile element for Web Services is the ontological description of the services provided. The OWL-S Matchmaker uses the Service Profile element to match requests with services. This Service Profile example in this Appendix describes a service with a single operation `ConvertLatLonToUTM` that has a single input `Latitude_Longitude` and a single output `UTM`. This Service Profile example maps these inputs and outputs to their counterparts in the OWL ontology, in this case `LatLon` and `UTM` respectively.

This Service Profile example also includes the service provider's contact information (`profile:contactInformation`) and a human-readable text description of the service provided (`profile:textDescription`), as the code example below illustrates.[16]

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rss="http://purl.org/rss/1.0/"
    xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
    xmlns:shadow-rdf="http://www.daml.org/services/owl
      -s/1.1/generic/ObjectList.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:swrl="http://www.daml.org/services/owl-s/1.1/generic/swrlx.owl#"
    xmlns:expr="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
    xmlns:jms="http://jena.hpl.hp.com/2003/08/jms#"
    xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#"
    xmlns:actor="http://www.daml.org/services/owl-s/1.1/ActorDefault.owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://owl-s.chrismetcalf.net/owl-s/LatLonToUTMProfile.owl#">

<owl:Ontology rdf:about="">
```

[16] All URLs listed in the code example were valid when the example was prepared.

```
    <owl:imports rdf:resource="http://owl-s.chrismetcalf.net/owl/Map.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl
      -s/1.1/Profile.owl"/>
    <owl:versionInfo>
            $Id: OWLSServiceProfileEmitter.java,v 1.1 naveen
        </owl:versionInfo>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Service.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl"/>
    <rdfs:comment>
            Add Comment
        </rdfs:comment>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-
     s/1.1/ActorDefault.owl"/>
  </owl:Ontology>

  <rdfs:Resource rdf:about="http://owl-s.chrismetcalf.net/owl
   -s/LatLonToUTMProfile.owl#LatLonToUTM">

    <profile:contactInformation>
      <actor:Actor rdf:about="http://www.daml.org/services/owl
       -s/1.1/ActorDefault.owl#ChrisMetcalf">
        <actor:title>Developer</actor:title>
        <actor:physicalAddress>123 Developer Lane.</actor:physicalAddress>
        <actor:webURL>http://chrismetcalf.net</actor:webURL>
        <actor:phone>555-1212</actor:phone>
        <actor:email>chris@chrismetcalf.net</actor:email>
        <actor:name>Chris Metcalf</actor:name>
        <actor:fax>555-1213</actor:fax>
      </actor:Actor>
    </profile:contactInformation>

   <profile:serviceName>LatLonToUTM</profile:serviceName>

   <profile:textDescription>
        Converts a Latitude/Longitude point a UTM point
    </profile:textDescription>

   <!-- Service Input —->
   <profile:hasInput>
      <process:Input rdf:about="http://owl-s.chrismetcalf.net/owl
       -s/LatLonToUTMProfile.owl#LatLon">
```

```
        <process:parameterType
          rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://owl-s.chrismetcalf.net/owl/Map.owl#Latitude_Longitude
        </process:parameterType>
      </process:Input>
    </profile:hasInput>


  <!-- Service Output —->
  <profile:hasOutput>
     <process:Output rdf:about="http://owl-s.chrismetcalf.net/owl
      -s/LatLonToUTMProfile.owl#UTM">
       <process:parameterType
         rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://owl-s.chrismetcalf.net/owl/Map.owl#Universal_Transverse_Mercator
       </process:parameterType>
     </process:Output>
    </profile:hasParameter>

  <rdf:type rdf:resource="http://www.daml.org/services/owl
    -s/1.1/Profile.owl#Profile"/>

  <rdf:type rdf:resource="http://www.daml.org/services/owl
    -s/1.1/Service.owl#ServiceProfile"/>

 </rdfs:Resource>
</rdf:RDF>
```

Appendix C Service Process Model for the TerraServer OWL-S Service

The Service Process Model element describes both simple processes and preconstructed complex process flows. In the code fragment shown below, one simple process is defined for the `ConvertLatLonToUTM` service described in the Appendix B.[17]

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uridef[
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
<!ENTITY owl "http://www.w3.org/2002/07/owl">
<!ENTITY xsd  "http://www.w3.org/2001/XMLSchema">
<!ENTITY service "http://www.daml.org/services/owl-s/1.1/Service.owl">
<!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
<!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl">
<!ENTITY concept "http://owl-s.chrismetcalf.net/owl-s/LatLonToUTMConcept.owl">
]>
<rdf:RDF
    xmlns:rdf= "&rdf;#"
    xmlns:rdfs= "&rdfs;#"
    xmlns:owl= "&owl;#"
    xmlns:xsd= "&xsd;"
    xmlns:service= "&service;#"
    xmlns:process= "&process;#"
    xmlns:profile= "&profile;#"
    xmlns:concept= "&concept;#"
    xml:base= "http://owl-s.chrismetcalf.net/owl-s/LatLonToUTMProcess.owl"
>

<owl:Ontology about="">
        <owl:versionInfo>
            $Id: OWLSProcessModelEmitter.java,v 1.1 naveen
        </owl:versionInfo>
        <rdfs:comment>
            Add Comment
```

[17] All URLs listed in the code example were valid when the example was prepared.

```
        </rdfs:comment>
        <owl:imports rdf:resource="&service;" />
        <owl:imports rdf:resource="&process;" />
        <owl:imports rdf:resource="&profile;" />
        <owl:imports rdf:resource="&concept;" />
        <!-- WSDL2OWL-S :: Add More Imports If needed -->
</owl:Ontology>

<!-- WSDL2OWL-S  :: Add composite processes if needed-->

<!--WSDL 2 OWL-S Generated code-->

<!--**List of Atomic Processes**-->
<!--ProcessName :LatLonToUTM-->
<!--******************************-->

    <!--Definitions for Atomic Process : LatLonToUTM-->

    <!--Inputs-->
    <process:Input rdf:ID="LatLon">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
            &concept;#ConvertLonLatPtToUtmPtTypeDeclaration
        </process:parameterType>
    </process:Input>

    <!--Outputs-->
    <process:Output rdf:ID="UTM">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
            &concept;#ConvertLonLatPtToUtmPtResponseTypeDeclaration
        </process:parameterType>
    </process:Output>

    <!--Process-->
    <process:AtomicProcess rdf:ID="LatLonToUTM">
        <process:hasInput rdf:resource="#LatLon"/>
        <process:hasOutput rdf:resource="#UTM"/>
    </process:AtomicProcess>

</rdf:RDF>
```

Appendix D Service Grounding for the TerraServer OWL-S Service

The Service Grounding element maps an OWL-S description of a service to its real-world Web Services counterpart. In the instance shown below, the OWL-S representation of the `LatLonToUTM` service is mapped to the actual TerraService WSDL file. Each of the OWL-S inputs and outputs is directly mapped to parameters defined in the service's WSDL definition. If the mapping were not direct, any required XSLT transformations would also be included in this file.[18]

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uridef[
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
<!ENTITY owl "http://www.w3.org/2002/07/owl">
<!ENTITY xsd  "http://www.w3.org/2001/XMLSchema">
<!ENTITY service "http://www.daml.org/services/owl-s/1.1/Service.owl">
<!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
<!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl">
<!ENTITY grounding "http://www.daml.org/services/owl-s/1.1/Grounding.owl">
<!ENTITY pm_file "http://owl-s.chrismetcalf.net/owl-s/LatLonToUTMProcess.owl">
]>
<rdf:RDF
    xmlns:rdf=     "&rdf;#"
    xmlns:rdfs=    "&rdfs;#"
    xmlns:owl=    "&owl;#"
    xmlns:xsd=     "&xsd;"
    xmlns:service= "&service;#"
    xmlns:process= "&process;#"
    xmlns:profile= "&profile;#"
    xmlns:grounding= "&grounding;#"
    xml:base= "http://owl-s.chrismetcalf.net/owl-s/LatLonToUTMGrounding.owl"
>
<owl:Ontology about="">
        <owl:versionInfo>
            $Id: OWLSGroundingModelEmitter.java,v 1.1 naveen
```

[18] All URLs listed in the code example were valid when the example was prepared.

```
        </owl:versionInfo>
        <rdfs:comment>
            Add Comment
        </rdfs:comment>
        <owl:imports rdf:resource="&service;" />
        <owl:imports rdf:resource="&process;" />
        <owl:imports rdf:resource="&profile;" />
        <owl:imports rdf:resource="&grounding;" />
        <owl:imports rdf:resource="&pm_file;" />
        <!-- WSDL2OWLS :: Add More Imports If needed -->
</owl:Ontology>

<grounding:WsdlGrounding rdf:ID="WsdlGrounding">
    <service:supportedBy rdf:resource=" ---Add INFO---" />
    <grounding:hasAtomicProcessGrounding rdf:resource="#LatLonToUTM_Grounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="LatLonToUTM_Grounding">
    <grounding:owlsProcess rdf:resource="&pm_file;#LatLonToUTM"/>

        <!-- Mapping to Web Service operation -->
        <grounding:wsdlOperation>
            <grounding:WsdlOperationRef>
                <grounding:portType rdf:datatype="&xsd;#anyURI">
                    http://terraservice-usa.com/#TerraServiceSoap
                </grounding:portType>
                <grounding:operation rdf:datatype="&xsd;#anyURI">
                    ConvertLonLatPtToUtmPt
                </grounding:operation>
            </grounding:WsdlOperationRef>
        </grounding:wsdlOperation>

        <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
            http://terraservice-usa.com/#ConvertLonLatPtToUtmPtSoapIn
        </grounding:wsdlInputMessage>

        <grounding:wsdlInput>
            <grounding:WsdlInputMessageMap>
                   <grounding:owlsParameter rdf:resource="&pm_file;#LatLon"/>
                   <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
                       http://terraservice.net/TerraService2.asmx?WSDL#parameters
                   </grounding:wsdlMessagePart>
            </grounding:WsdlInputMessageMap>
```

```
        </grounding:wsdlInput>


        <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
            http://terraservice-usa.com/#ConvertLonLatPtToUtmPtSoapOut
        </grounding:wsdlOutputMessage>

        <grounding:wsdlOutput>
            <grounding:WsdlOutputMessageMap>
                <grounding:owlsParameter rdf:resource="&pm_file;#UTM"/>
                <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
                    http://terraservice.net/TerraService2.asmx?WSDL#parameters
                </grounding:wsdlMessagePart>
            </grounding:WsdlOutputMessageMap>
        </grounding:wsdlOutput>

        <grounding:wsdlDocument  rdf:datatype="&xsd;#anyURI">
            http://terraservice.net/TerraService2.asmx?WSDL#
        </grounding:wsdlDocument>

        <grounding:wsdlReference rdf:datatype="&xsd;#anyURI">
            http://www.w3.org/TR/2001/NOTE-wsdl-20010315
        </grounding:wsdlReference>

</grounding:WsdlAtomicProcessGrounding>

</rdf:RDF>
```

References

URLs are valid as of the publication date of this document.

[Barclay 02] Barclay, Tom; Gray, Jim; Strand, Eric; Ekblad, Steve; & Richter, Jeffrey. *TerraService.NET: An Introduction to Web Services* (MS-TR-2002-53). Redmond, WA; Microsoft Research, Advanced Technology Division, Microsoft Corporation, 2002. http://www.ec-gis.org/docs/F5130/WEBSERVICES.PDF

[IBM 02] International Business Machines. *JROM Overview*. http://www.alphaworks.ibm.com/tech/jrom (2002).

[ISAL 01] Intelligent Software Agents Lab in the Robotics Institute at Carnegie Mellon University. *The Intelligent Software Agents Lab*. http://www-2.cs.cmu.edu/~softagents/index.html (2001).

[ISAL 04] Intelligent Software Agents Lab in the Robotics Institute at Carnegie Mellon University. *Tools*. http://www.daml.ri.cmu.edu/tools/details.html (2004).

[Lewis 04] Lewis, Grace A. & Wrage, Lutz. *Approaches to Constructive Interoperability* (CMU/SEI-2004-TR-020, ADA431067).Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tr020.html.

[Lewis 05] Lewis, Grace A. & Wrage, Lutz. *A Process for Context-Based Technology Evaluation* (CMU/SEI-2005-TN-025, ADA441251). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tn025.html.

[Martin 04a] Martin, David, et al. *OWL-S: Semantic Markup for Web Services*. W3C Member Submission: November 22, 2004. http://www.w3.org/Submission/OWL-S/.

[Martin 04b] Martin, David, et al. *Bringing Semantics to Web Services: The OWL-S Approach*. http://www-2.cs.cmu.edu/~softagents/papers /OWL-S-SWSWPC2004-final.pdf (2004).

[Microsoft 05] Microsoft Corporation. *TerraServer–USA*. http://terraserver.microsoft.com/ (2005).

[Srinivasan 05] Srinivasan, Naveen; Paolucci, Massimo; & Sycara, Katia. *CODE: A Development Environment for OWL-S Web Services* (CMU-RI-TR-05-48). Pittsburgh, PA: Robotics Institute, Carnegie Mellon University, 2005. http://www.ri.cmu.edu/pub_files/pub4 /srinivasan_naveen_2005_1/srinivasan_naveen_2005_1.pdf.

[Stanford 06] Stanford Medical Informatics. *what is protégé-OWL*. http://protege.stanford.edu/overview/protege-owl.html (2006).

[W3C 99a] World Wide Web Consortium. *Resource Description Framework*. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (1999).

[W3C 99b] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*. http://www.w3.org/TR/xslt (1999).

[W3C 03] World Wide Web Consortium. *Semantic Web Services Interest Group*. http://www.w3.org/2002/ws/swsig/ (2003).

[W3C 04a] World Wide Web Consortium. *OWL Web Ontology Language Guide: W3C Recommendation February 10, 2004*. http://www.w3.org/TR/owl-guide/ (2004).

[W3C 04b] World Wide Web Consortium. *OWL Web Ontology Language Reference: W3C Recommendation February 10, 2004*. http://www.w3.org/TR/owl-ref/ (2004).

[W3C 04c] World Wide Web Consortium. *OWL Web Ontology Language for Services (OWL-S)*. http://www.w3.org/Submission/2004/07/ (2004).

[W3C 05a] World Wide Web Consortium. *Semantic Web Services Framework (SWSF)*. http://www.w3.org/Submission/2005/07/ (2005).

[W3C 05b] World Wide Web Consortium. *WSDL-S Submission Request to W3C*. http://www.w3.org/Submission/2005/10/ (2005).

[W3C 05c] World Wide Web Consortium. *Web Service Modeling Ontology (WSMO) Submission*. http://www.w3.org/Submission/2005/06/ (2005).

[Wallnau 01] Wallnau, Kurt; Hissam, Scott; & Seacord, Robert. *Building Systems from Commercial Components*. New York, NY: Addison-Wesley, 2001.

[WS-I 04] Web Services Interoperability Organization. *Basic Profile Version 1.1. 2004–08–24*. http://www.ws-i.org/Profiles /BasicProfile-1.1.html (2004).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	April 2006	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Model Problems in Technologies for Interoperability: OWL Web Ontology Language for Services (OWL-S)	FA8721-05-C-0003

6. AUTHOR(S)

Chris Metcalf and Grace A. Lewis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2006-TN-018

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

In a services-oriented environment, services are constantly being added and removed. Application developers often do not have control over the services they utilize. What would happen if a service required by an application were removed from the environment or had its interface changed? What if a new and better service were introduced that an application might be able to utilize? Existing services-oriented frameworks do not protect application developers against these contingencies.

The OWL Web Ontology Language for Services (OWL-S) is a language to describe the properties and capabilities of Web Services in such a way that the descriptions can be interpreted by a computer system in an automated manner. This technical note presents the results of applying the model problem approach to examine the feasibility of using OWL-S to allow applications to automatically discover, compose, and invoke services in a dynamic services-oriented environment.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Web Services, ontology, OWL-S, interoperability, SOA, services, service-oriented architecture, OWL, semantics, Semantic Web Services	60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL