In Proc. IEEE Conf. on CEC/EEE, 2008

SSWiM: A Semantic Service, Wrapper and Invocation Manager*

Anna Sibirtseva Zhongnan Shen Jianwen Su
Department of Computer Science
University of California at Santa Barbara

Fuliang Weng Baoshi Yan Yao Meng
Research and Technology Center
Robert Bosch Inc.

Abstract

Integrating service description, discovery, and invocation functionalities presents several fundamental problems in the management of web services and is a basic problem for composing web services over a network. In this paper, we present the design of a system called "Semantic Service, Wrapper, and Invocation Manager" (SSWiM) which provides these key functionalities. In particular, SSWiM manages storage of and queries over service descriptions in a service registry, wraps existing REST services with WSDL service interfaces, and supports for service invocation at run time. We describe WSDL services based on two levels of ontologies: a domain ontology for data sets and a service ontology for services. The input/output messages (data) and the functionality of a (WSDL) service are mapped to the domain ontology and the service ontology, respectively. We developed the data mapping methodology and the mapping algorithm. Service descriptions are registered in our service registry which supports a set of service discovery queries. The wrapper builder in SSWiM can generate WSDL services automatically from REST services so that services can be invoked uniformly in WSDL format.

1 Introduction

The development of web service standards and related emerging technologies have significantly increased the popularity of the Service Oriented Architecture (SOA). A fundamental promise of SOA is the ease and flexibility of composing web services that are bitesized software systems easily managed. An increasing number of web services is being made available online, which presents an opportunity to develop software systems of greater complexity and functionality. One challenge is to develop effective technology to fully utilize the available services. There are, however, many obstacles; a main difficulty is the lack of integrated support for automated service discovery and composition, although specialized support for only discovery or only composition has been studied in the literature [12, 6, 7, 15, 5, 14, 24, 13, 4, 9, 3]. In this paper, we formulate a technical platform which is an initial step towards providing such integrated support, with a goal of supporting practical applications. We demonstrate a specific telematics application for the platform.

In the telematics domain, a spoken dialog system provides a natural way for users to operate devices and access services. It allows the driver to perform their primary task, i.e., driving, with a minimum distraction. One such example is a conversational dialog system, called CHAT, that has been developed to enable the driver to interact with limited but representative applications, such as operating a MP3 player (entertainment), search for a favorable restaurant (POIs), and finding a desirable route with multiple constraints (navigation) [19, 20]. It is natural that dialog accessible web services become an important topic in the field of telematics: using speech to access web services would make web services accessible everywhere in everyday life.

We now describe a scenario for accessing web services through a dialog system. A person is traveling from San Jose to San Francisco in her car, and she wants to visit a museum in San Francisco. The following is her conversation with the dialog system, where T is the traveler, and S is the system.

T: I want to visit some museum in San Francisco.
S: OK. The followings are museums in San Francisco. Museum of Modern Art (MOMA); Asian Art Museum; San Francisco Railway Museum.
T: How much is the admission for MOMA?
S: The admission fee is 12.5 dollars.
T: Where can I park my car?
S: You can park at Fifth and mission Parking Garage.
T: Tell me the route to the garage.
S: Keep on freeway 101, then ...

In this conversation, the system needs to access several web services as well as compose them together, including finding museums in an area, querying the price of a museum, searching for parking structures near a place, and calculating a route. These services are discovered, composed

*Supported in part by a research grant from Robert Bosch LLC and NSF grants ISI-0415195 and CNS-0613998.

and invoked on the fly, depending on the user's requests.

This paper makes three contributions. The first is the development of a system called "Semantic Service, Wrapper and Invocation Manager" (SSWiM). SSWiM integrates the management of ontologies, semantic service descriptions, service discoveries, and service invocations including wrappers (automated creation and execution) for services using different standards. Through integrating these functionalities, SSWiM can serve as a fundamental platform which on one hand interacts with the web through service invocation, and on the other hand can be used as the ontology and (semantic) service registry, and as a single invocation mechanism that interacts with networked services of all types. SSWiM is developed as a part of a telematics application prototype which also includes "upper level" systems including, in particular, a service composer and a dialog system interacting with the driver. In the following, we illustrate a typical example in the telematics application and demonstrate how SSWiM helps.

SSWiM enables a dialog system to compose and access web services. It provides a set of functions for a dialog system to discover, compose and invoke services. Fig. 1 shows the framework of the dialog system and SSWiM. The dialog system is responsible for talking to the user, and the composition module composes services to fulfil the user's request. The dialog system replies on SSWiM to query ontologies, find web services and invoke services. A clear advantage of SSWiM is its ability to separate logical level representation of service functionalities with ontologies from the implementation details concerning service invocations.

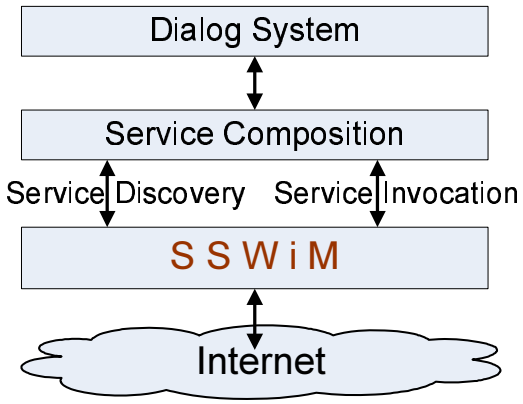

Figure 1. The system framework

We use two levels of ontologies to semantically describe web services. The domain ontology is a conceptualization of a domain into a human understandable, machine-readable format consisting of concepts, attributes and relationships. The service ontology is an abstraction of web service functionalities in a domain and used to annotate real web services. Service semantics can help to enhance the accuracy of service discovery. On the other hand, the dialog system itself needs to understand the semantics of users's requests. Therefore, these two ontologies are shared between the dialog system and SSWiM, and are used to understand what the user says, describe available web services, and exchange data between the dialog system and SSWiM.

The second contribution of the paper is a rule based language for defining semantic mappings for WSDL services. A service is annotated with a conceptual service in the service ontology, and its inputs/outputs are mapped to the domain ontology. We define the syntax and semantics of mapping rules which map XML schemas to the domain ontology. A mapping algorithm is used to convert ontology instances to XML messages and vice verse at run time. A Service Registry in SSWiM stores service descriptions and provides a set of interfaces for service discovery. Services in registry can be queried based on their inputs and outputs as well as their service functionality. Since the dialog system and SSWiM share the same ontologies, queries can be expressed in terms of ontologies. Query results are returned back as a list of service descriptions.

As the third contribution, we developed an automated wrapper builder with graphical interface in SSWiM which can generate WSDL wrappers for REST services automatically. This builder simplifies the task of building WSDL services (the wrappers) from REST services. With service wrappers, the dialog system only needs to deal with WSDL services uniformly. In this paper, we use XPath to define the mapping between WSDL and REST services and present an algorithm which generates wrappers by using code templates. The source code for wrappers can be either in C♯ or Java. Wrappers generated are hosted by the Invocation Engine which is another component of SSWiM.

The Invocation Engine in SSWiM gets invocation requests from the dialog system, transforms ontology instances to XML messages, invokes services and converts output XML messages to ontology instances before they are returned back to the dialog system. Based on the type of services, the invocation engine either invokes wrappers running on the engine or accesses services on the Internet.

The remainder of the paper is organized as follows. Section 2 describes the two types of ontologies. Section 3 defines the syntax and semantics of service descriptions. Section 4 presents the automated service wrapper builder. Section 5 discusses some implementation issues, and Section 6 concludes this paper.

2 Ontologies

Service ontology has been studied variously (OWL-S [11], WSDL-S [18], WSMO [22], SWSO [8] and in [6, 12, 17]). In our framework, we consider two levels of ontologies: the domain ontology and the service ontology. The former describes all the concepts and their relationships in a certain domain. The semantics of these concepts are

well defined and have common understanding. The latter is a collection of abstracted service functionalities which are commonly used in the domain. The service ontology depends on the domain ontology to describe the inputs/outputs of service functionalities.

Concepts and *attributes* are two main entities in a domain ontology. A concept can have multiple attributes and each attribute is a pair of name and type. The type of an attribute can be a certain concept or one of the primitive types. The relationship between concepts are captured by two relations: *subclass* and *superclass*. The subclass inherits all the attributes from its superclass and can have its own attributes. The subclass and superclass relationships construct the concept hierarchy of a domain ontology. Some attributes can serve the role of identifying a concept. These attributes are called *defining characteristics* and are required for the concept. A defining characteristic of a concept can represent the concept and sometimes can be used in place of the concept.

Fig. 2 shows a portion of a domain ontology. "POI"(Point of Interest), "Gas Station" and "Museum" are concepts, furthermore, "Gas Station" and "Museum" are subclasses of "POI". The attribute "name" of "POI" is a string, while the type of "location" is another concept "Location". The attributes "name", "contactNumber" and "location" are defining characteristics of the concept "POI" because each of them can uniquely identify the corresponding POI. An attribute of a concept may have multiple values, for example, a museum may have several exhibits going on in it.

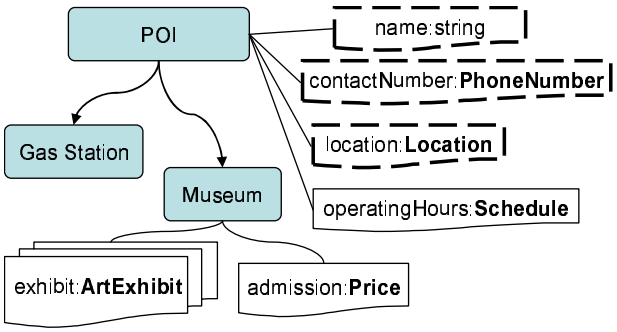

Figure 2. An example of domain ontology

Formally, a *domain ontology* O_d is a pair (C, Sup), where C is a set of concepts, and Sup is a binary relation on C. $< c_1, c_2 > \in Sup$ if c_1 is a superclass of c_2. A *concept* $c \in C$ is a four tuple $< A, \tau, D, M >$, where A is a set of attribute names, τ is a mapping from A to $C \cup \{integer, string, real\}$, D and M are boolean functions on A. τ maps each attribute to its type, D decides if an attribute is defining characteristic, and M describes if an attribute can have multiple values.

An *instance* of a concept is an instantiation of the concept with attributes populated with values. An instance of the "Museum" concept represents and describes a concrete museum in the real world. In our ontology-based SSWiM framework, instances are used to exchange data between the upper level applications, e.g., the dialog system, and the lower level service invocation. Ontology instances from applications are transformed to XML messages before web services are invoked, and messages from web services are transformed back to instances before they are returned to the applications. The mapping process will be detailed later.

Based on domain ontology, we can define service ontology. Service ontology is a set of abstracted service functionalities in a domain. Each service function in the domain is called a conceptual service whose inputs and outputs are concepts in domain ontology. A concrete web service may realize one or more conceptual services in the service ontology. Fig. 3 shows a part of service ontology. "FindPOI" and "GetRoute" are two conceptual services in a service ontology. Their inputs and outputs are concepts in the domain ontology. "Georeferences", "Route" and "POI", for examples, are concepts in a certain domain ontology. Note that the input concept "Location" for the "GetRoute" conceptual service and the output concept "POI" for the "FindPOI" conceptual service are connected with big arrows which means the input or the output could be a list of values instead of a single value. For instance, the "GetRoute" conceptual service may take a source, a destination, and several intermediate stops as inputs which are all instances of "Location", and generates a route as output.

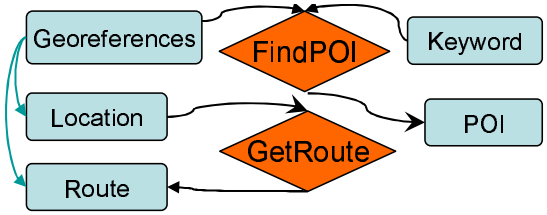

Figure 3. An example of service ontology

A *service ontology* O_s is a triple (S, I, O), where S is a set of conceptual service names, and I, O are mappings from S to $2^{C \times \{true, false\}}$. I, O maps each conceptual service respectively to its input/output domain concepts and indicates if the input/output concept can hold multiple values. The input/output concept of a service can have multiple values if the service consumes a list of instances of that concept.

In our SSWiM framework, we intentionally divide ontologies into two levels. This separation provides flexibility and scalability for possible upper level applications, such as service composition. Service composition can be done based on conceptual services, while the result of service composition is realized at run time by finding concrete web services in service registry. As services in the registry are changed dynamically, the upper level composition can be relatively stable because the realization of the composition is left to the service discovery at run time. The same thing

happens when service requirements are changed by the application. The service ontology becomes a "common language" between the upper level applications and the lower level service discovery and invocation.

3 Descriptions for WSDL Services

In this session, we discuss our approach for service description. Web services are currently described by WSDL[21] files in which service interfaces are specified in terms of input/output messages. We extend WSDL service descriptions based the domain ontology and the service ontology. Service messages and service operations are mapped to these ontologies, and a mapping algorithm makes conversions between messages and ontology instances at run time. Instead of using UDDI[16, 10], service descriptions are stored in a service registry in SSWiM which provides a set of query interfaces using ontologies.

3.1 Service Description

Web services in SSWiM are described with domain and service onotologies in addition to WSDL files. Service descriptions are stored in service registry and can be retrieved by service queries using ontology. Ontology enabled service search can help to eliminate the ambiguities caused by service discovery solely based on syntax and enhance the accuracy of service queries.

A *service description* D is a six tuple (F, I, O, M_i, M_o, W), where $F \in S$ is the conceptual service that this service realizes, I and O are the concepts that the service takes as inputs and outputs resp., M_i and M_o are mappings from input/output message schemas to the domain ontology resp., and W is the WSDL file associated with this service.

A web service realizes one conceptual service in a domain. The "YahooLocal"[23] web service, for example, realizes the "FindPOI" conceptual service. Although it is possible that one service realizes several conceptual services, for example, the conceptual services realized by "ESRI ArcWeb Services"[1] include "FindPOI", "GetRoute" and "FindMap", we treat them as individual web services. F denotes the conceptual service realized by a web service. The input concepts I for the "YahooLocal" service are "Keyword" and "Address", and the output concept O is "POI".

Note that input/output concepts of the concrete service and the input/output concepts of the conceptual service that the concrete service realizes are not necessary to be the same. Usually, the input/output concepts of the concrete service could be subclasses of the corresponding concepts of the conceptual service because conceptual services are supposed to be more general than concrete web services. For example, the input concept "StruturedAddress" for the "YahooLocal" service is a subclass of "Location" which is the input concept for conceptual service "FindPOI".

As mentioned before, ontology instances are the data exchanged between upper level applications and SSWiM. An application may invoke several web services and the outputs of the former service may become the inputs of the next one. In this sense, ontology instances provide a uniform format for data exchange. On the other hand, the input/output messages of a WSDL web service are defined in XML schema. Fig. 4 shows part of the schema of the output message of "YahooLocal" service. This schema needs to be mapped to the output concept "POI" which is shown in Fig. 2. Therefore, mappings have to be established between the XML schemas of input/output messages and the input/output concepts in a domain ontology. M_i and M_o define these mappings.

```
<schema>
    <element name="ResultSet">
        <complexType>
            <sequence>
                <element name="Result" type="ResultType"/>
            </sequence>
        </complexType>
    <element>
    <complexType name="ResultType">
        <sequence>
            <element name="Title" type="string" />
            <element name="City" type="string" />
            <element name="Rating" type="RatingType" />
        </sequence>
    </complexType>
    <complexType name="RatingType">
        <sequence>
            <element name="AverageRating" type="float" />
            <element name="TotalRatings" type="integer" />
        </sequence>
    </complexType>
</schema>
```

Figure 4. XML schema for yahoo local service

The mapping between XML schemas and concepts in the domain ontology is defined in terms of rules. The mapping rules are simple enough that they do not cause much overhead at run time when ontology instances are converted to XML messages and vice verse. The rule syntax can be defined as follows.

$$E\ (\ .\ E)^* \rightarrow C\ (\ .\ A)^*$$

E is an element name in the XML schema, C is a concept name, and A is an attribute name.

Consider the mapping rules for the output message of "YahooLocal" service. The schema is shown in Fig. 4, and the output concept "POI" is shown in Fig. 2. The set of mapping rules are summarized below.

ResultSet.Result→POI
ResultSet.Result.Title→POI. name:string
ResultSet.Result.City→POI. location:Georeferences|
Location|Address. city:string
ResultSet.Result.Rating.AverageRating→POI.rating:double

In the third rule above, the left side are elements "ResultSet", "Result"and "City" in the XML schema, and "City" is

a subelement of "Result" which itself is a subelement of "ResultSet". The "." operator on the left side is used to represent subelement relationship. On the right side, "POI" is a concept, and "location", "city" are attributes. The "." operator on the right side is used to represent concept-attribute relationship. The type of "location" attribute is "Georeferences|Location|Address", and the type of "city" attribute is "string". "City" is an attribute of "Address". For each attribute name, it is followed by its type and separated by the ":" operator. The "|" operator in the rule represents the subclass relationship in the domain ontology. For example, "Location" is a subclass of "Georeferences" and "Address" is a subclass of "Location". The name of a concept is always prefixed by its super classes which are separated by "|".

Mapping for attributes in XML schema can be handled in similar rules. Without loss of generality, we only discuss mapping rules for elements in XML schema in this paper.

It is not necessary that every element in an XML schema is mapped to the domain ontology. The "TotalRatings" element, for example, does not have a mapping rule. These elements without mapping rules are ignored at service invocation.

Sometimes, it is possible that two or more elements in an XML schema are mapped to the same attribute of a concept or two or more attributes are mapped to the same element. For example, one element in the input message of "YahooLocal" service is called "Street", and this element is a concatenation of the "StreetNumber" and the "StreetName" attributes of the "Address" concept. In this case, the mapping rule $E(.E)^* \rightarrow C(.A)^*$ is not sufficient. To handle this situation, the mapping rules for input message schemas are extended to

$$E\ (\ .\ E)^* \rightarrow C\ (\ .\ A)^*(+C\ (\ .\ A)^*)^*.$$

All the attributes in the right side of the rule are mapped to the same element on the left side. The "+" operator stands for concatenation.

For example, the mapping rule for the "street" element of "YahooLocal" service can be expressed as

street→Georeferences|Location|Address.StreetNum:string
+Georeferences|Location|Address.StreetName:string.

Similarly, the mapping rules for output message schemas are extended to

$$E\ (\ .\ E)^*(+E\ (\ .\ E)^*)^* \rightarrow C\ (\ .\ A)^*.$$

All the elements in the left side of the rule are concatenated and mapped to the attribute on the right side.

Ontology instances are converted to input XML messages and output XML messages are converted back to ontology instances before and after the service invocation. The mapping algorithm is straightforward. For an ontology instance, the algorithm first checks if the type of the instance is one of the input concept in I or a subclass of it. If this is the case, all the attributes of this instance are traversed recursively and for each attribute, the algorithm searches the mapping rules to see if there is one rule for this attribute. After a rule is picked, the corresponding element is constructed in the XML message. If the attribute needs to be concatenated with other attributes, the values of these attributes are extracted from the instance and are concatenated to set the value of the XML element. The algorithm converts output XML messages to ontology instances works in a similarly way except that it starts from the output XML message and constructs the corresponding instance based on the elements in the message.

Based on the above discussion, the input message for "YahooLocal" service is built up as follows. The instance of concept "Georeferences|Location|Address" has attributes "StreetNum", "StreetName" and "City" with corresponding values "151", "3rd Street" and "San Francisco". The set of mapping rules are {YahooPOI.city→Address.City, YahooPOI.street→Address.StreetNum+Address.StreetName}. The super classes for "Address" and the type of attributes are omitted here in the rules. When the "City" attribute is visited, the first rule is applied, and the input message is constructed as "<YahooPOI><city>San Francisco</city></YahooPOI>". When the "StreetNum" attribute is visited, the second rule is applied. Because "StreetNum" needs to be concatenated with "StreetName" in the rule, "151" and "3rd Street" are combined to set the value of "Street" element. The final message is "<YahooPOI><city>San Francisco</city><street>151 3rd Street</street></YahooPOI>".

3.2 Service Registry

The service registry in SSWiM is a repository for service descriptions and provides a set of query functions. Upper level applications search for web services through the registry with queries formulated using the domain ontology and the service ontology. For each query, a list of service descriptions are returned back to the applications so that they can select one service from the list and use again SSWiM to invoke the service.

In SSWiM, the service registry provides the following interfaces for service discovery.

- hasInputConcept(a set of Concepts)
- hasOutputConcept(a set of Concepts)
- realizeConceptualService(ConceptualService)

Given a set of concepts in the domain ontology, the *hasInputConcept* query function returns all the service descriptions which contains the set of concepts as inputs, and the *hasOutputConcept* query function returns all the service descriptions which contains the set of concepts as outputs. The *realizeConceptService* query function looks for

all the service descriptions which realize the conceptual service. For example, *realizeConceptualService("FindPOI")* will return the descriptions of "YahooLocal" and "ESRI POI" web services because these two services realize the "FindPOI" conceptual services.

4 Wrapper Builder

Since web services in SSWiM use WSDL format, whereas many concrete services nowadays use REST format, we introduce a concept of automatic service wrapper builder. The wrapper builder generates web service wrappers. A web service wrapper provides an interface between SSWiM WSDL environment and REST concrete services. The concept of wrapper builder adds scalability to the SSWiM framework, which has to deal with large and growing set of service wrappers as new concrete services evolve.

4.1 Data mapping

Wrapper itself is a web service. Fig. 5 illustrates the operation of a web service wrapper for our framework.

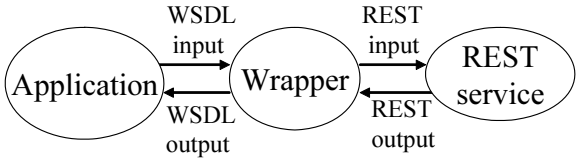

Figure 5. Web Service Wrapper Functionality

The wrapper takes input data in WSDL format from the invoking application and forms a REST request to a concrete REST service. Then it gets the reply from this service in REST format and transforms it into WSDL format returned to the application.

Each wrapper has a set of mapping rules. These mapping rules specify how the data transformation happens between REST and WSDL. Each wrapper will have input mapping rules and output mapping rules. To explain input mapping rules let's consider the structure of REST service request. A standard REST query is constructed of 3 main building blocks: URL of the web service, delimiters, and input parameters. For example, Fig. 6 illustrates the components of a typical REST query. REST service reply is an XML document.

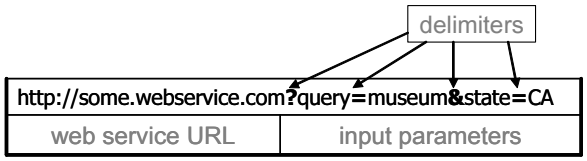

Figure 6. An example of REST query components

To build a REST request, a wrapper program needs data from user as a triple (U, D, I), where U is the URL of the REST web service, D is the set of delimiters, and I is the set of input parameters. The whole REST-request is a single string, so no additional data type transformations are needed, and the REST request is formed by string concatenation operations on the user input.

To build a WSDL reply from REST reply, the wrapper needs a triple (X, T, O), where X are XPath expressions that specify the location of data desired in REST reply, T are the data types of WSDL service outputs, and O is the desired names for WSDL service outputs.

The set X of XPath expressions is necessary for parsing the REST output which is an XML document. Using XPath expressions is a design decision motivated by the lack of common structure in REST services' XML output. For instance, Fig. 7 are the XML documents returned by "YahooLocal" and "Upcoming"[2] REST services. The point of interest ("Museum of Modern Art", "San Francisco") returned by "YahooLocal" REST service is contained inside the XML tags <Title> and <City> under <Result> parent tag, whereas "Upcoming" REST service has this data formatted as attributes of the XML tag <event>. Thus, to extract the string "Museum of Modern Art", the XPath expressions for "YahooLocal" and "Upcoming" are "Result/Title" and "event/attribute::venuename" respectively. Therefore, if set X is provided by the user, wrapper builder has all the necessary information to build the XML parsing capability of the wrapper.

```
                YahooLocal
<Result>
    <Title>Museum Of Modern Art</Title>
    <City>San Francisco</City>
</Result>
            Upcoming
<rsp stat="ok" version="1.0">
    <event  id="389857"
            venue_name="Museum of Modern Art"
            venue_city="San Francisco">
    </event>
</rsp>
```

Figure 7. XML documents of REST services

The set T of output elements' data types and O of desired output element names are needed to build the WSDL output in correspondence with domain ontology concepts. If the data types of some outputs are complex, the set T should contain the whole description of the complex type data structure.

4.2 Templates

Our wrapper builder provides the capability of selecting the programming language of wrappers. We have currently implemented the support for C♯ and Java, which are the most widely used languages in the field of web services. The code generated by wrapper builder for a wrapper, can

be abstractly categorized into 2 types: invariant code and dynamic code. Invariant code includes all the lines that stay the same among all wrappers written in the same language. For example, for all wrappers, written in C♯, the code that declares library includes ("Using System", "Using System.Web", etc) is invariant code. Dynamic code includes those parts of code that are generated depending on the information provided by users in the form of 2 triples. The following list describes all the invariant elements used in the code generation:

- Library includes.
- HTTP Web Request Initialization.
- XML Parsing Initialization.

The dynamic code includes:

- Composing REST request from (U, D, I).
- Classes generated from (X, T, O) for WSDL outputs.
- Parsing REST replies to construct WSDL outputs.

Thus, we have 2 templates in wrapper builder: all invariant code for C♯ wrappers constitutes the abstract notion of C♯-template, all invariant code for Java wrappers constitutes the abstract notion of Java-template.

4.3 The algorithm

To better understand the algorithm of wrapper builder, let's consider the execution flow of a wrapper. A REST-to-WSDL wrapper takes the following steps to execute:

1. Build REST request provided with values of input parameters given by the user.
2. Send a request and receive response from REST web service. The response is stored for later XML parsing.
3. Perform XML Parsing of the response, extracting the necessary data from the XML file. Objects of WSDL output class are created to hold these data. In case of the example in Fig. 6, objects for "Museum" class are created as WSDL output.
4. Return the WSDL output to calling application.

Now let's consider the algorithm of wrapper builder. The wrapper builder takes as inputs the 2 three-tuples: (U, D, I) describing wrapper's REST input, and (X, T, O) describing data output format. The wrapper builder produces a wrapper for REST service as an output.

Below are the major steps taken by wrapper builder algorithm.

1. Get the user-provided information in the form of 2 triples.
2. Create wrapper project directories. The service wrapper produced by wrapper builder is created as a standard Visual Studio Web Service Project or Java EJB Module Project.
3. Wrapper code generation. This step generates invariant and dynamic code based on templates and user-provided information.

5 Implementation

The previous sections discuss several technical aspects of SSWiM, including ontologies, service description, service registry, service invocation and wrapper builder. Put them together, Fig. 8 shows the architecture of SSWiM. SSWiM was implemented and different applications can be built upon it, such as the dialog system mentioned in section 1. This section gives a more detailed descriptions for the design and implementation of each module in SSWiM.

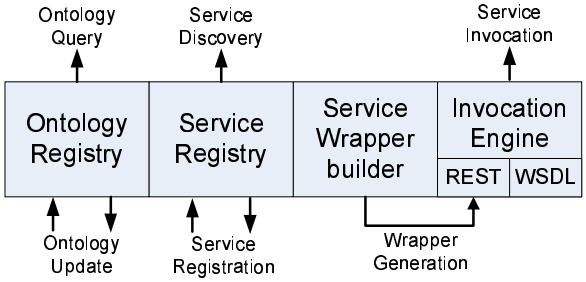

Figure 8. The architecture of SSWiM

The ontology registry is the repository for the domain ontology and the service ontology. It provides two interfaces: query the ontologies and update the ontologies. Concepts, attributes and their relationships can be retrieved by the upper level applications through the query interface. Although ontologies are relatively stable, they still can be modified by the update interface. The ontology registry itself can be a web service, which makes it convenient to be accessed through the internet. In our current implementation, the ontologies are stored in relational databases.

The service registry stores all the service descriptions presented in section 3. The upper level applications discover relevant services from the registry through the three discovery functions. The service registry itself can be a web service, and new services are registered by invoking this service. Currently, the service registry is implemented using a relational database. A relation to store all service descriptions has the schema Services(*serviceID, realizedConceptService, inputConcepts, outputConcepts, InputMapping, OutputMapping, Operation, WSDLURL, Rating*). The fields in sequence are the ID of the service, the concept service realized, the set of input concepts, the set of output concepts, the mapping rules for input message, the mapping rules for output message, the service operation in WSDL, the URL of the WSDL file, and some rating of the service. Each service description is one tuple in the database. A service discovery is translated to an SQL query which is evaluated upon the database. The three service discovery functions use the *realizedConceptService*, *inputConcepts* and *outputConcepts* fields resp. The *Rating* field is to help applications select a service from a service list. The *serviceID* is used to retrieve detailed information of a

service such as *inputMapping* and *outputMapping*.

The automated wrapper builder generates WSDL wrappers which resides on the invocation engine from REST services. The wrapper builder can be accessed either as a web service or through a graphical user interface(GUI). Fig. 9 shows the GUI part of the wrapper builder that reads XPath expressions X and output types T in an output description tuple (X, T, O).

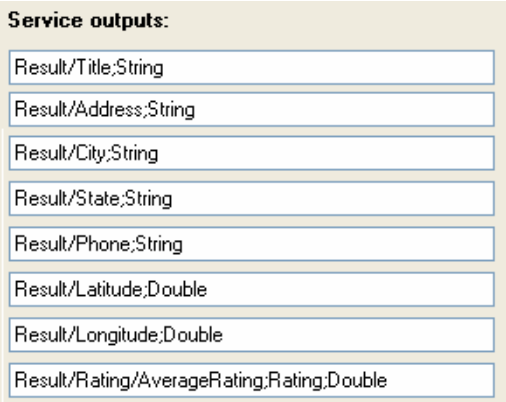

Figure 9. The GUI of wrapper builder

As shown in Fig. 9, XPath expressions and output types are separated by a semicolon. If an output type is complex, the primitive type which comprises it is provided after complex type separated by a semicolon.

The invocation engine gets service invocation requests from upper level applications and decides to invoke either a service wrapper hosted on the engine or a WSDL service through the internet. REST services are wrapped as individual WSDL services running on the invocation engine and are transparent to the engine for service invocation.

6 Conclusions

In this paper, we present SSWiM that provides a platform for upper level applications to manage ontologies, discover services, wrap REST services with WSDL interfaces and invoke services of different types. As a part of our current work, we focus on upper level applications such as the dialog system with service composition which utilizes SSWiM. The dialog system talks to the user, and completes the user's requests by performing online service composition with the help of SSWiM.

References

[1] Esri arcweb developer's guide. http://www.arcwebservices.com/v2006/help/index.htm.

[2] Upcoming api documentation - version 1.0. http://upcoming.yahoo.com/services/api/.

[3] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. 31st Int. Conf. on Very Large Data Bases (VLDB)*, pages 613–624, 2005.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. 1st Int. Conf. on Service Oriented Computing*, volume 2910 of *LNCS*, pages 43–58, 2003.

[5] A. Bernstein and M. Klein. Discovering services: Towards high precision service retrieval. In *Proc. of the CaiSE workshop on Web Services, e-Business, and the Semantic Web: Foundations, Models, Architecture, Engineering and Applications*, 2002.

[6] J. Cardoso and A. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.

[7] I. Elgedawy, Z. Tari, and M. Winikoff. Exact functional context matching for web services. In *Proc. Int. Conf. on Service Oriented Computing (ICSOC)*, pages 143–152, 2004.

[8] S. B. et al. Semantic Web Services Ontology (SWSO) Version 1.0. `http://www.daml.org/services/swsf/1.0/swso/`, May 2005.

[9] C. Gerede, R. Hull, O. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proc. 2nd Int. Conf. on Service-Oriented Computing (ICSOC)*, 2004.

[10] J. Luo, B. Montrose, A. Kim, A. Khashnobish, and M. Kang. Adding OWL-S support to the existing UDDI infrastructure. In *Proc. 4th IEEE Int. Con. on Web Services (ICWS)*, 2006.

[11] OWL-S 1.1 Release. `http://www.daml.org/services/owl-s/1.1/`, November 2004.

[12] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proc. Int. World Wide Web Conf. (WWW)*, pages 553–562, 2004.

[13] Z. Shen and J. Su. On automated composition for web services. In *WWW*, 2007.

[14] G. Spanoudakis, A. Zisman, and A. Kozlenkov. A service discovery framework for service centric systems. In *Proc. Int. Conf. on Service Computing (SCC)*, 2005.

[15] N. Srinivasan, M. Paolucci, and K. Sycara. Adding OWL-S to UDDI, implementation and throughput. In *First Int. Workshop on SemanticWeb Services and Web Process Composition*, 2003.

[16] U.T.Committee. Uddi spec technical committee draft 3.0.2, Oct. 2004. http://uddi.org/pubs/uddi_v3.htm.

[17] K. Verma and A. Sheth. Semantically annotating a web service. *IEEE Internet Computing*, 11(2):83–85, 2007.

[18] W3C. Web services semantics - wsdl-s 1.0, Nov. 2005. http://www.w3.org/Submission/WSDL-S/.

[19] F. Weng, L. Cavedon, B. Raghunathan, and et. al. Developing a conversational dialogue system for cognitively overloaded users. In *Proc. of Interspeech*, 2004.

[20] F. Weng, B. Yan, Z. Feng, and et. al. Chat to your destination. In *Proc. of ACL Sigdial workshop*, 2007.

[21] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[22] Web Service Modeling Ontology. `http://www.wsmo.org/`.

[23] YahooDeveloper. Yahoo! local search apis. http://developer.yahoo.com/search/local/.

[24] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.