SEMANTICS-BASED DYNAMIC WEB SERVICE COMPOSITION

Keita Fujii

School of Information and Computer Science,
University of California, Irvine, Irvine CA, 92697-3425, U.S.A.
kfujii@ics.uci.edu

Tatsuya Suda

School of Information and Computer Science,
University of California, Irvine, Irvine CA, 92697-3425, U.S.A.
suda@ics.uci.edu

ABSTRACT
This paper presents a semantics-based dynamic service composition architecture that composes an application through combining distributed components based on the semantics of the components. This architecture consists of a component model called Component Service Model with Semantics (CoSMoS), a middleware called Component Runtime Environment (CoRE), and a service composition mechanism called Semantic Graph based Service Composition (SeGSeC). CoSMoS represents the semantics of components. CoRE provides interfaces to discover and access components modeled by CoSMoS. SeGSeC composes an application by discovering components through CoRE, and synthesizing a workflow of the application based on the semantics of the components modeled by CoSMoS.

This paper describes the latest design of the semantics-based dynamic service composition architecture, and also illustrates the implementation of the architecture based on the Web Service standards, i.e., WSDL, RDF, SOAP, and UDDI. The Web Service based implementation of the architecture allows existing Web Services to migrate onto the architecture without reimplementation. It also simplifies the development and deployment of a new Web Service on the architecture by automatically generating the necessary description files (i.e., WSDL and RDF files) of the Web Service from its runtime binary (i.e., a Java class file).

Keywords: dynamic service composition, semantics, service oriented computing, web service, component model

1. Introduction

The paradigm in distributed systems is shifting towards Service Oriented Computing (SOC)[1] where components (also called *services* in SOC) play as basic elements in developing applications. In SOC, applications are *composed* of multiple components, i.e., platform- and network-independent software elements distributed across the network. In the current SOC environments, applications are statically composed by human developers. However, it is more desirable if applications are dynamically composed on demand (i.e., when requested by users) by software agents because it eliminates the need for pre-installation and pre-configuration of applications, and also allows applications to be customized based on user profiles (e.g., usage history) and contexts (e.g., device availability). This concept of dynamically composing applications on demand is called Dynamic Service Composition[2].

To dynamically compose applications from a collection of distributed components, a component in SOC must (1) overtly define its operation in a standard, machine-readable format, (2) publish its definition into the network in order to make itself discoverable by other components, and (3) support a protocol to communicate with a client or other components. A component in SOC thus requires three capabilities: description, discovery, and communication. Web Service is a typical SOC example as it consists of Web Services Description Language (WSDL) for description, Universal Description, Discovery, and Integration (UDDI) for discovery, and Simple Object Access Protocol (SOAP) for communication. WSDL is an XML-based Interface Definition Language. SOAP is an XML-based protocol that enables Web Service operation invocation over the standard transport protocols such as HTTP and SMTP. UDDI is a SOAP-based directory service for publishing and discovering Web Services.

Based on the Web Service standards (i.e., WSDL, SOAP and UDDI), several Dynamic Service Composition systems have recently been proposed and implemented (e.g., Refs. 3-21). However, those systems often require a user to request an application in a manner that may not be intuitive to the user. For instance, a user is required to choose a template of the application[3], or specify the pre/post conditions of the application using logic formula[11].

In order to allow users to request applications in an intuitive manner (i.e., using a natural language), the authors of this paper proposed[22,23] a semantics-based dynamic service composition architecture. The proposed architecture consists of a component model called Component Service Model with Semantics (CoSMoS), a middleware called Component Runtime Environment (CoRE), and a service composition mechanism called Semantic Graph based Service Composition (SeGSeC). CoSMoS represents the semantics of the distributed components. CoRE provides interfaces to discover and access components modeled by CoSMoS. When a user requests an application, SeGSeC composes the requested application by discovering and accessing the distributed components through CoRE, and synthesizing a workflow of the requested application based on the semantics of the components and of the user's request represented by CoSMoS. The feasibility of the proposed architecture was confirmed[22] through preliminary implementation of the architecture, and the scalability of the architecture was also empirically confirmed[23].

This paper summarizes the design of the semantics-based dynamic service composition architecture with emphasis on recent modifications of the architecture, including the refined semantic representation based on Conceptual Graph[24]. In addition, this paper also presents the implementation of the semantics-based dynamic service composition architecture based on the Web Service standards, i.e., WSDL, Resource Description Framework (RDF), SOAP, and UDDI, in order to validate the feasibility, portability and flexibility of the architecture. The Web Service based implementation of the architecture is designed to satisfy the following requirements. First, the Web Service based implementation of the architecture maintains compatibility with existing Web Service based systems because it does not require any modification of the existing Web Service standards (i.e., WSDL/RDF/SOAP/UDDI). Second, the Web Service based implementation of the architecture allows existing Web Services to migrate to the

architecture without reimplementation. Lastly, the Web Service based implementation of the architecture simplifies the development and deployment of a new Web Service by automatically generating the necessary description files (i.e., WSDL and RDF files) of the Web Service from its runtime binary (i.e., a Java class file).

The rest of the paper is organized as follows. Section 2 summarizes the semantics-based dynamic service composition architecture that consists of CoSMoS, CoRE and SeGSeC. Section 3 describes the Web Service based implementation of the architecture in detail. Section 4 concludes the paper.

2. SEMANTICS-BASED DYNAMIC SERVICE COMPOSITION ARCHITECTURE

This section summarizes the semantics-based dynamic service composition architecture. The architecture consists of CoSMoS, CoRE and SeGSeC. CoSMoS is a semantics-aware component model that represents the semantics of the components using semantic graph representation. CoRE is a middleware that provides interfaces to discover and access components modeled by CoSMoS. SeGSeC is a semantics-based dynamic service composition mechanism that composes the requested application based on the semantics of the components and of the request from the user.

2.1. *COMPONENT SERVICE MODEL WITH SEMANTICS (CoSMoS)*

This section presents a semantics-aware component model, Component Service Model with Semantics (CoSMoS). This section first describes the overview of CoSMoS, followed by the detailed design of CoSMoS. This section concludes with the comparison between CoSMoS and the existing Semantic Web Service models such as OWL-S[25].

2.1.1. *CoSMoS Overview*

Many existing component models (e.g., WSDL, JavaBeans/EJB, COM) represent a component by defining its operations and properties. In existing component models, an operation is defined as a pair of inputs to and outputs from the component. Each input and output is defined as a pair of a name and a data type (e.g., “int price”, “String name”). Similarly, each property of a component is also defined as a pair of a name and a data type. Although the names of operations, inputs, outputs and properties may imply their semantics, the existing component models do not explicitly represent the semantics information regarding a component. Therefore, human designers have to rely on external documents such as specification of a component in order to obtain the semantics of the component.

In order to explicitly represent the semantics of a component, a semantics-aware component model named Component Service Model with Semantics (CoSMoS) has been developed. CoSMoS represents the semantics of a component by modeling a component from three aspects: the functional aspect, the semantic aspect and the logical aspect. In the functional aspect, CoSMoS defines the operations and properties of the component using data types. In the semantic aspect, CoSMoS defines the semantics of the component, namely, what each operation, input, output, and property of the component

semantically represents. In the logical aspect, CoSMoS defines rules (i.e., sets of conditions and consequences) that SeGSeC uses in extracting the semantics of a synthesized workflow. Each aspect is represented as a semantic graph, a directed graph that consists of nodes and labeled links. CoSMoS integrates the semantic graph representations of the three aspects of a component and forms a single semantic graph to model the component.

Figure 1 shows an example of how CoSMoS models a component that generates a JPEG image of a map showing a direction from one address to another. The functional aspect in Figure 1 defines that the direction generator component implements an operation which accepts two Address data as inputs and outputs one JPEG image. The semantic aspect in Figure 1 defines that the operation of the direction generator component '*generate*' a '*direction*' '*from*' an '*origin*' '*to*' a '*destination*'. The direction generator component does not define any rules, thus Figure 1 does not show the logical aspect of the component.

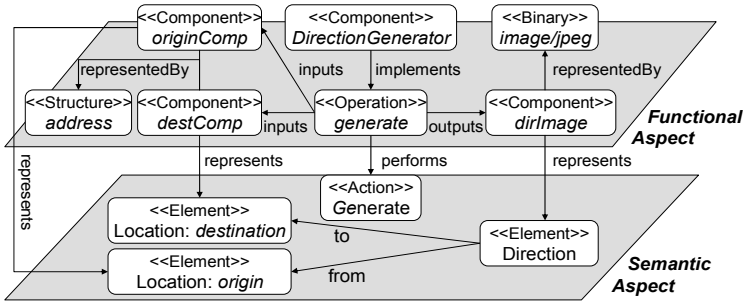


Figure 1. A direction generator component in CoSMoS

2.1.2. *CoSMoS Architecture*

The following subsections explain how CoSMoS models a component in the functional, semantic and logical aspects in detail.

Functional Aspect:

In the functional aspect, CoSMoS defines the operations of a component. In CoSMoS, an operation is defined as a set of inputs, outputs, and exceptions. Each input (and output) of an operation is defined as a component, representing that the operation accepts (or generates) another component as its input (or output). Some components (e.g., a microphone and a printer) may accept or generate a physical object (e.g., sound and a paper) instead of accepting or producing any binary data. CoSMoS supports such components by defining their input or output as a component without any data type. Some components (e.g., a printer) may support several options for its operation (e.g., different paper sizes). CoSMoS supports such an operation by defining an *enumeration* (e.g., a 'paper size' *enumeration*) consisting of several *values* (e.g., 'letter', 'legal' and

'A4' *values*) as one of its inputs. An operation may throw an exception when its execution fails.

The functional aspect of CoSMoS also defines the properties of a component. In CoSMoS, a property of a component is defined as a component, representing that the property can be retrieved as another component.

When an input, output or property of a component represents some data, the functional aspect of CoSMoS specifies its data type using common primitive data types (such as integer, string, float, and Boolean) and/or common data structures (such as array, structured data, enumeration, and binary data). CoSMoS also supports arbitrary data types (e.g., XML Schema data types, or Java collection libraries), as long as it can determine compatibility between two data types (e.g., through parsing XML Schema file, or through Java reflection functionality).

CoSMoS represents the functional aspect of a component as a semantic graph. In CoSMoS, a component is represented as a *Component* node. An operation of a component is represented as an *Operation* node with an '*implements*' link connected to a *Component* node. Similarly, a property of a component is represented as a *Component* node (representing the property) with a '*hasPropertyOf*' link connected to another *Component* node (representing the owner of the property). The inputs and outputs of an operation are represented as *Component* nodes (representing the input or the output) with '*inputs*' and '*outputs*' links connected to an *Operation* node. The exception of an operation is represented as an *Exception* node with a *'throws'* link connected to an *Operation* node. Each *Component* node may have a '*representedBy*' link pointing toward a *DataType* node (or its subclass node such as *Structure* or *Binary*), representing the data type of the component.

Figure 1 shows an example of how CoSMoS represents the functional aspect of the direction generator component as a semantic graph. The functional aspect of Figure 1 represents that '*Component DirectionGenerator*' *implements* '*Operation generate*', which accepts two inputs, '*Component originComp*' and '*Component destComp*', and generates one output, '*Component dirImage*'. Similarly, Figure 2 illustrates how CoSMoS represents the functional aspect of a restaurant component, which has one property ('*Component RestAddress*') representing the address of the restaurant.

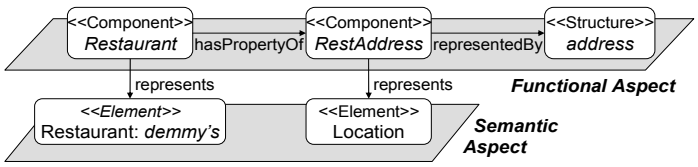


Figure 2. A restaurant component in CoSMoS

Semantic Aspect:

In the semantic aspect, CoSMoS defines the semantics of a component, namely, what each operation, input, output, and property of the component semantically represents.

CoSMoS represents the semantic aspect of a component as a semantic graph, whose notation is based on Conceptual Graph[24]. The semantic aspect of CoSMoS defines each node of a semantic graph as an instance of a *concept*. The *concept* of a node specifies what the node semantically represents. For example, a node defined as an instance of the *concept* '*Human*' represents a human. This is similar to an object-oriented language where an object is defined as an instance of a class, and the class of the object specifies what the object represents. In addition to the *concept*, the semantic aspect of CoSMoS may specify the name of a node. For example, two '*Human*' nodes may be named '*Alice*' and '*Bob*' in order to represent two humans named Alice and Bob. In CoSMoS, the nodes representing nominal *concepts* (e.g., *'Human'*, *'Restaurant'*) are called *Element* nodes, and the nodes representing verbal *concepts* (e.g., *'Generate'*, *'Print'*) are called *Action* nodes.

Once several nodes (i.e., instances of *concepts*) are defined, the semantic aspect of CoSMoS connects them with labeled links and forms a semantic graph. The label of each link is also defined as a *concept*, and specifies the semantics of the relationship between two nodes. For example, the '*Human: Alice*' node and the '*Human: Bob*' node may be linked with a '*isASiblingOf*' link, representing that Alice and Bob are siblings (Figure 3). Similarly, the '*from*' and '*to*' links in Figure 1 represent that the '*Direction*' is '*from*' the '*Location: origin*' and '*to*' the '*Location: destination*'.

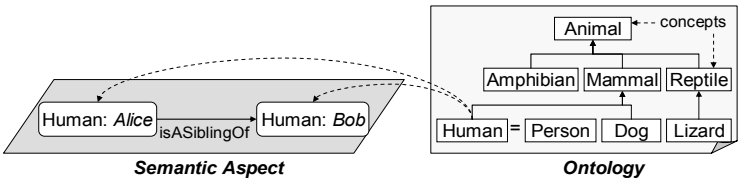


Figure 3. An example of a CoSMoS semantic aspect and Ontology

In order for a software agent to properly interpret the semantic aspect of CoSMoS, the formal definition of the *concepts* used in defining the semantic aspect needs to be provided. The definition of *concepts* (e.g., '*Human'*, '*Mammal'*, '*Animal'*) is called an *ontology*, and it defines the relationships between the *concepts* (e.g., '*Human is a kind of Mammal, which is a kind of Animal*'). CoSMoS assumes that an ontology is defined by using an existing ontology definition framework such as RDF Schema and OWL. This allows component developers to define and use their own ontology to develop components.

An *ontology* is used in determining whether a *concept* is compatible with another or not. CoSMoS considers that a *concept* C1 is compatible with another *concept* C2 either if C1 and C2 are equivalent (e.g., '*Human*' and '*Person*'), or if C2 is a generalized *concept* of C1 (e.g., '*Human*' and '*Animal*'). When, for example, an ontology is defined in either RDF Schema and OWL, CoSMoS considers two *concepts* C1 and C2 are compatible if it is possible to traverse from C1 to C2 through *owl:equivalentTo* and *rdf:subClassOf* properties.

CoSMoS integrates the semantic and functional aspects of a component by linking the nodes in the semantic aspect and the nodes in the functional aspect. An *Element* node

in the semantic aspect and a *Component* node in the functional aspect may be connected with a '*represents*' link in order to represent the semantics of the inputs, outputs or properties of a component. Figure 1, for example, shows that the '*Location: origin*' and '*Location: destination*' nodes in the semantic aspect represent the semantics of the '*Component originComp*' and '*Component destComp*' nodes in the functional aspect. Similarly, an *Action* node in the semantic aspect and an *Operation* node in the functional aspect may be connected with a '*performs*' link in order to represent the semantics of the operation. For instance, Figure 1 shows that the '*Generate*' node in the semantic aspect represents the semantics of the '*Operation generate*' node in the functional aspect.

In CoSMoS, an *Element* node may be declared as a *Wildcard Element*. To illustrate how a *Wildcard Element* is used in CoSMoS, consider a printer component that receives an image as an input and generates a print out of the input image as an output. The input image and the printed output of a printer component should represent the same *Element* (i.e., nominal *concept*) which cannot be determined when designing the component. For example, if the input image illustrates a direction to a restaurant, the printed output also illustrates the direction to the restaurant. If the input image is a textual description of a novel, the printed output also describes the novel. In such cases, CoSMoS uses a *Wildcard Element* node (e.g., the '*Wildcard*' node in Figure 4) to indicate that the input image and the printed output represent the same but arbitrary *Element*.

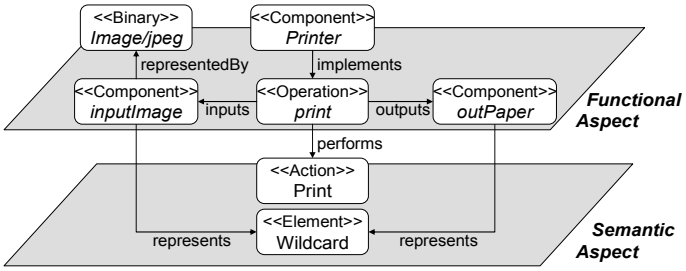


Figure 4. A printer component in CoSMoS

Logical Aspect:

CoSMoS allows components to define rules[a] in the logic aspect. A rule is defined as a set of conditions and consequences, and represents that, when its conditions are met (e.g., "if a person uses a microphone"), its consequences become valid (e.g., "then, the recorded sound is of the person"). Rules are used by SeGSeC when extracting the semantics of a synthesized workflow. Details of how rules are used in SeGSeC will be explained later in Section 2.3.

CoSMoS represents the logical aspect of a component as a semantic graph. In the logical aspect, a rule is represented as a *Rule* node with one or more '*condition*' links and one or more '*consequence*' links. Each '*condition*' and '*consequence*' link is pointing to

[a] Please note that the rules defined in the logical aspect are not intended to represent the internal state of a component or the preconditions/effects of an operation.

another labeled link connecting two *Element* nodes in the semantic aspect. A *Rule* node is also connected to a *Component* node in the functional aspect with a '*knows*' link. Figure 5 shows a microphone component with a rule representing "if a person uses a microphone, the recorded sound is of the person."

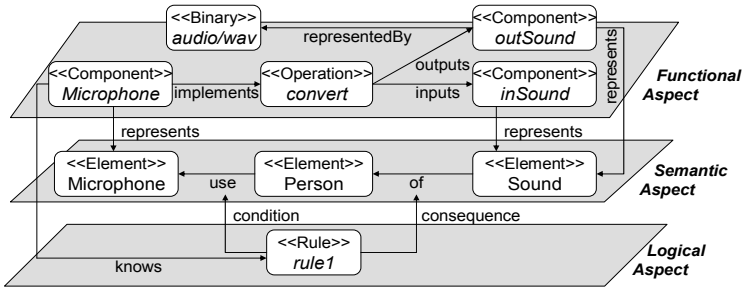


Figure 5. A microphone component in CoSMoS

2.1.3. *CoSMoS Class Diagram*

Figure 6 is a UML class diagram illustrating the formal specification of CoSMoS. As described in Section 2.1.1, CoSMoS models a component as a single semantic graph that integrates the functional, semantic and logical aspects of the component. Nodes in the semantic graph (e.g., *Component* nodes) are defined as instances of the classes (e.g., the '*Component*' class) in Figure 6. The labels of the associations in Figure 6 (e.g., '*implements*') are the *concepts* that are used to label the links between nodes in the semantic graph (e.g., the '*implements*' link between *'Component Microphone'* and *'Operation convert'* in Figure 5).

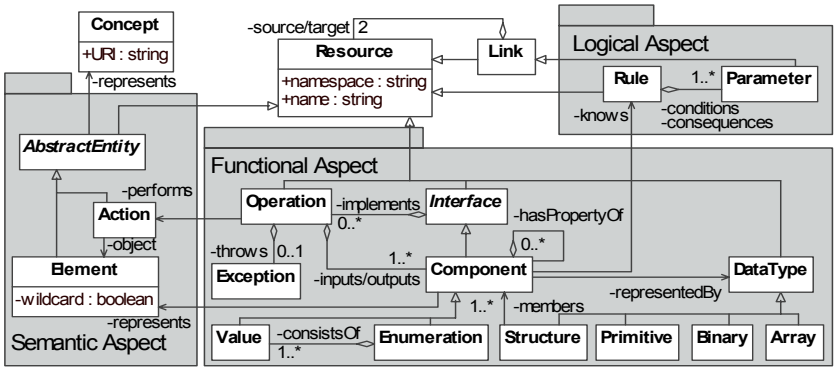


Figure 6. The CoSMoS Class Diagram

2.1.4. *CoSMoS and other Semantic Web Service models*

CoSMoS is unique compared to existing Semantic Web Service models (such as OWL-S[25], WSDL-S[26], WSDF[27], SESMA[28], and WSMO[29]) in the following aspects.

CoSMoS annotates the semantics of an operation of a component using a *concept*. For example, CoSMoS represents the semantics of a book purchase operation provided by an online book store using a '*purchase*' *concept*. Although some of the existing Semantic Web Service models[26] follow the same approach as CoSMoS, many existing models[25,27,28,29] annotate the semantics of an operation as the effect (a.k.a. post condition) of the operation. For example, they represent the semantics of the book purchase operation as an effect specifying that the ownership of the book is transferred to the user. Although the effect may provide more formal and precise semantics of an operation, the *concept*-based semantic annotation of an operation is more suitable for the proposed semantics-based dynamic service composition than the effect-based annotation because, with the *concept*-based annotation, it is easier to map the semantics of the user's request (expressed in a natural language) onto the semantics of a component.

CoSMoS represents not only the semantics of the inputs and outputs of an operation but also the semantic relationships between the inputs and outputs. The semantic relationships between inputs and outputs are represented by labeled links connecting the *Element* nodes representing the semantics of the inputs and outputs. For instance, in the example shown in Figure 1, the labeled link '*from*' represents that the input '*origin*' is '*from*' the output '*direction*', and similarly, the labeled link '*to*' represents that the input '*destination*' is '*to*' the output '*direction*'. Most of the existing Semantic Web Service models, on the other hand, only represent the semantics of the inputs and outputs by specifying the *concepts* of the inputs and outputs, assuming that the ontology which defines the *concepts* of the inputs and outputs provides the semantic relationship between those *concepts*. In other words, existing models do not support the semantic relationships that are not defined in an ontology. Because CoSMoS allows a component designer to arbitrarily specify the semantic relationships, CoSMoS is more flexible in representing the semantic relationships than existing models.

2.2. *COMPONENT RUNTIME ENVIRONMENT (CoRE)*

This section describes Component Runtime Environment (CoRE), a middleware which is designed to support CoSMoS on various component technologies.

CoRE consists of two interfaces (Discovery interface and Access interface) and three groups of modules (DiscoveryEngines, InvokerEngines, and PropertyAccessEngines) (See Figure 7). The Discovery interface provides an interface to discover a component distributed in a network. Upon receiving a query from the Discovery interface, the DiscoveryEngine searches the requested component(s) and provides a CoSMoS representation of the discovered component(s) (e.g., by analyzing the component's metadata) to the Discovery Interface. The Access interface provides an interface to invoke an operation of a component and to retrieve a property of a component. Upon receiving a query from the Access interface, the InvokerEngine invokes an operation of a component, and the PropertyAccessEngine retrieves a property of a component.

DiscoveryEngine, InvokerEngine and PropertyAccessEngine may be implemented with various component technologies, such as Web Service, Jini, or uPnP. When several Engines are installed, CoRE automatically selects a proper Engine for each component by identifying the component technology on which the component is implemented.

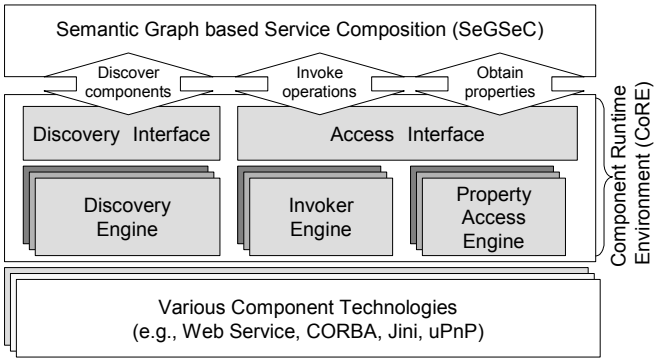


Figure 7. CoRE Architecture

2.3. *SEMANTIC GRAPH BASED SERVICE COMPOSITION (SeGSeC)*

This section presents a service composition mechanism named Semantic Graph based Service Composition (SeGSeC). This section describes the overview of SeGSeC, followed by its detailed mechanism description. This section also describes the features of SeGSeC compared to those of existing Web Service Composition systems, and ends with a brief summary of the performance evaluation of SeGSeC.

2.3.1. *SeGSeC Overview*

SeGSeC composes an application from multiple components based on the semantics of the request from the user and the semantics of the components. SeGSeC assumes that all components are modeled by CoSMoS, and they can be discovered and accessed through CoRE. SeGSeC consists of four modules: RequestAnalyzer, ServiceComposer, SemanticsAnalyzer, and ServicePerformer (See Figure 8).

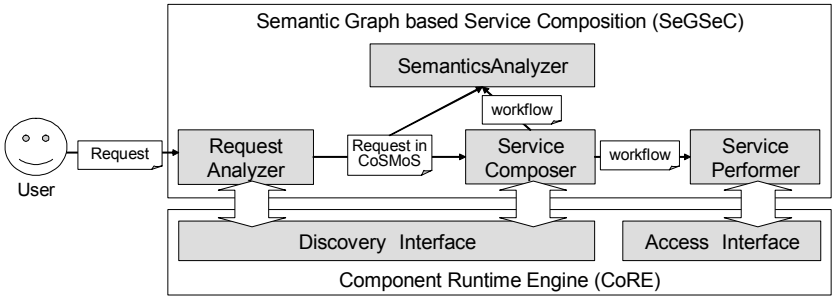


Figure 8. Modules in SeGSeC

When a user requests an application in a natural language, RequestAnalyzer parses the request in a natural language into a CoSMoS semantic graph, and passes the request represented as the semantic graph to ServiceComposer. ServiceComposer, upon receiving the request from RequestAnalyzer, discovers components in the network based on the request, synthesizes a workflow using the discovered components, and passes the workflow and the request to SemanticsAnalyzer. SemanticsAnalyzer extracts the semantics of the workflow from the semantics of the components in the workflow, and examines if the semantics of the workflow satisfies the request. If SemanticsAnalyzer concludes that the semantics of the workflow satisfies the request, the workflow is passed to ServicePerformer, which executes the workflow.

The following subsections describe the detailed algorithm of SeGSeC using an example scenario in which a user requests an application to print out a map showing the direction from user's house to a restaurant, and SeGSeC composes the requested application using four components, Home (Figure 9), Restaurant (Figure 2), Direction Generator (Figure 1), and Printer (Figure 4).

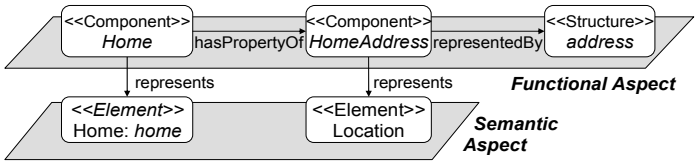


Figure 9. A home component in CoSMoS

2.3.2. *SeGSeC Algorithm*

RequestAnalyzer Module:

When a user requests an application in a natural language (e.g., "print direction from home to demmy's"), RequestAnalyzer parses the request into a CoSMoS semantic graph (e.g., Figure 10). Since the natural language analysis is a well established research area, SeGSeC assumes that RequestAnalyzer uses existing techniques (e.g., BEELINE[30]) for parsing a request (in a natural language) into a semantic graph. RequestAnalyzer may access to the Discovery Interface provided by CoRE and use the semantics of the components when parsing a request. For instance, when parsing the request "print direction from home to demmy's", RequestAnalyzer may discover a restaurant component (shown in Figure 2) by using the keyword "demmy's" and realize that "demmy's" is the name of a restaurant. After parsing a request into a semantic graph, RequestAnalyzer passes the semantic graph to ServiceComposer.

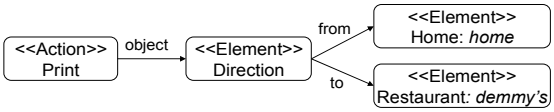


Figure 10. CoSMoS representation of the request "print direction from home to demmy's"

ServiceComposer Module:

Upon receiving a request (represented as a semantic graph) from RequestAnalyzer, ServiceComposer discovers components in the network through the Discovery Interface of CoRE and synthesizes a workflow from the received request.

In order to synthesize a workflow from a request, ServiceComposer first discovers a component whose operation *performs* (i.e., has a '*performs*' link to) the *action* specified in the request. This component that ServiceComposer first discovers is called an initial component. In the example scenario described in Section 2.3.1, ServiceComposer first discovers a Printer component (shown in Figure 4) as the initial component because its operation *performs* the '*Print*' action that appears in the request (shown in Figure 10).

After discovering an initial component, ServiceComposer synthesizes a workflow that only contains the initial component, and then expands the workflow through the process called Input Complement. In the Input Complement, ServiceComposer first discovers several components whose outputs or properties are compatible with the inputs of the components in the workflow. ServiceComposer considers that an output and an input (or a property and an input) are compatible if their data types are compatible and also if they represent the *Element* nodes whose *concepts* are compatible. Then, ServiceComposer decides which of the discovered components to add to the workflow by comparing them with the request from the user.

The Input Complement is a recursive process which, given an operation in the workflow, discovers components that provide inputs of the operation, selects a component among the discovered ones, and expands the workflow by adding the selected component into the workflow. The following is a pseudo-code of the Input Complement.

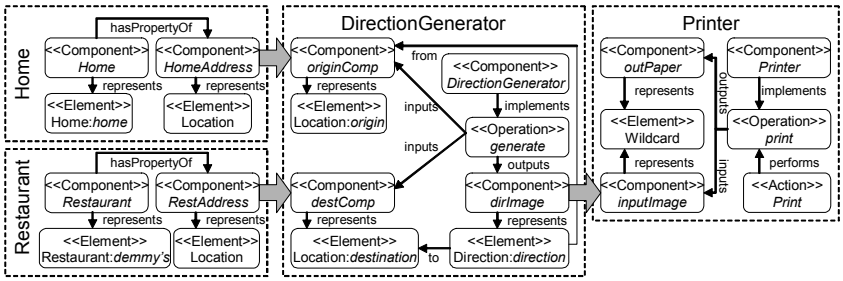
```
01:inputComplement(operation, userRequest, workflow){
02: create a list of empty lists L[1] ... L[N];
03: # N = the number of the inputs of the operation
04: for each input i_x of the operation{
05:  if i_x is an Enumeration{
06:   add its values to L[x];
07:  }else{
08:   discover outputs/properties that are
09:   compatible with i_x and store them in L[x];
10:  }
11: }
12:
13: compute possible combinations LC[1] … LC[M] of
14: the components in L[1] ... L[N];
15: # choose N components from each of L[1] … L[N]
16: # and store them into LC[x] such that the members of
17: # two lists LC[x] and LC[y] are different if x!=y
18:
19: sort LC[1]...LC[M] based on the similarities;
```

```
20: # similarity: the number of elements that
21: # appear both in userRequest and in LC[x]
22:
23: for each LC[x]{
24:  expand the workflow such that
25:  LC[x] provides the inputs of the operation op;
26:  if LC[x] contains outputs{
27:   for each output{
28:    identify operation op2 that generates the output;
29:    inputComplement(op2, userRequest, workflow);
30:   }
31:  }else{
32:   if semanticMatching(userRequest, workflow)
33:    return workflow;
34:  }
35: }
36:}
```

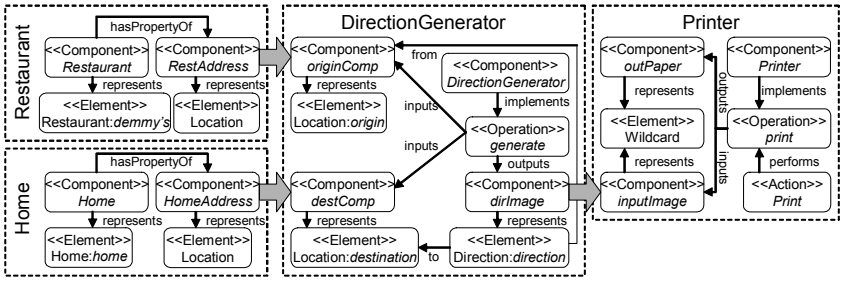
In the Input Complement, ServiceComposer first identifies the inputs of the given operation and discovers all the components whose outputs or properties are compatible with the inputs of the given operation (Line 2-11). After discovering components that provide inputs of the given operation, ServiceComposer determines which components among the discovered ones to add to the workflow (Line 13-21). In order to do so, ServiceComposer first computes all possible combinations of the outputs and properties of the discovered components such that each of the combinations provides all the inputs of the given operation (Line 13-14). Then, ServiceComposer calculates the similarity of each of the combinations against the given request, and selects the one with the highest similarity (Line 19). The similarity of a combination against a request is defined as how many *Element* nodes in the CoSMoS representations of the components in the combination also appear in the request. After selecting a combination, ServiceComposer expands the workflow by adding the components in the combination to the workflow such that the outputs and properties in the combination become inputs of the given operation (Line 24-25). ServiceComposer iterates the Input Complement as long as the workflow contains an operation whose inputs need to be *complemented* (Line 26-29). When ServiceComposer finishes *complementing* all the operations in the workflow, ServiceComposer passes the workflow and the request to SemanticsAnalyzer (Line 32-33).

In the example scenario described in Section 2.3.1, after discovering the Printer (shown in Figure 4) as the initial component, ServiceComposer performs the Input Complement and *complements* the '*print*' operation of the Printer with the '*generate*' operation of the Direction Generator (shown in Figure 1). ServiceComposer then iterates the Input Complement to *complement* the '*generate*' operation of the Direction Generator. In this iteration of the Input Complement, ServiceComposer discovers the Home (shown in Figure 9) and Restaurant (shown in Figure 2) because their properties (i.e.,

'*HomeAddress*' and '*RestAddress*') are both compatible with the two inputs (i.e., '*originComp*' and '*destComp*') of the '*generate*' operation. In this case, ServiceComposer computes four combinations, {'*HomeAddress*', '*HomeAddress*'}, {'*RestAddress*', '*RestAddress*'}, {'*HomeAddress*', '*RestAddress*'}, and {'*RestAddress*', '*HomeAddress*'}, and selects either {'*HomeAddress*', '*RestAddress*'} or {'*RestAddress*', '*HomeAddress*'} because they have the highest similarity of 2 against the given request shown in Figure 10. Depending on which combination is selected, ServiceComposer synthesizes either of the two workflows shown in Figure 11 (a) and (b). Note that the workflow in Figure 11 (a) provides the requested application of printing out the direction from home to the restaurant, and that the workflow in Figure 11 (b) does not provide the requested application as it prints out the direction from the restaurant (i.e., not from home) to home (i.e., not to the restaurant). In order to identify whether the synthesized workflow satisfies the request from the user, ServiceComposer asks SemanticsAnalyzer to examine the workflow against the request.



(a) Print direction from home to restaurant



(b) Print direction from restaurant to home

Figure 11. Workflows created through Input Complement

SemanticsAnalyzer Module:

Upon receiving a workflow and a user's request from ServiceComposer, SemanticsAnalyzer extracts the semantics of the workflow using the semantics of the components in the workflow, and examines if the semantics of the workflow satisfies the request or not. This process is called Semantic Matching.

In the Semantic Matching, SemanticsAnalyzer first converts the workflow into a semantic graph. A workflow (e.g., Figure 11 (a)) consists of a set of components (e.g.,

'*Home*', '*Restaurant*', '*DirectionGenerator*', and '*Printer*' components in Figure 11 (a)), each of which is modeled as a semantic graph (i.e., dotted boxes in Figure 11 (a)). A workflow also specifies data flows among components, i.e., which output or property of a component becomes an input to another component (shown as thick arrows in Figure 11 (a)). SemanticsAnalyzer converts a workflow into a semantic graph by interconnecting the semantic graphs of the components in the workflow with '*usedBy*' links such that each '*userBy*' link corresponds to each data flow in the workflow.

After converting a workflow into a semantic graph, SemanticsAnalyzer extracts the semantics of the workflow by applying a set of predefined rules called *semantics retrieval rules*[b] (Table 1) onto the semantic graph. The *semantics retrieval rules* add new links onto a semantic graph converted from a workflow such that the newly added links represent the semantics of the workflow. If the components in the workflow define any rules in the logical aspect, SemanticsAnalyzer also applies those rules onto the semantic graph in addition to the *semantics retrieval rules*. This allows component designers to specify how SemanticsAnalyzer extracts the semantics of a workflow, resulting in flexibility and extensibility in Semantic Matching.

Table 1. Semantics retrieval rules

Name	Rule	Meaning
Rule 1	performs(O,A) & outputs(O,C) & represents(C,E) → object(A,E)	If O performs A, O outputs C, and C represents E, then, E is the object of A.
Rule 2	hasPropertyOf(C_1,C_2) & represents(C_1,E_1) & represents(C_2,E_2) → applyTo(E_2,E_1)	If C_1 has a property C_2, C_1 represents E_1, and C_2 represents E_2, then, apply E_2 to E_1.
Rule 3	usedBy(C_1, C_2) & represents(C_1, E_1) & represents(C_2, E_2) → applyTo(E_2, E_1)	If C_1 is used by C_2 (i.e., C_1 becomes an input C_2), C_1 represents E_1, and C_2 represents E_2, then, apply E_2 to E_1.
Rule 4	L(E_1, W) & L(E_2, E_3) & applyTo(E_1, E_2) → applyTo(W, E_3)	If E_1 has a link L to W, E_2 has the same link L to E_3, and E_1 is applied to E_2, then, apply W to E_3.
Rule 5	applyTo(E_1, E_2) & L(N, E_1) → L(N, E_2)	If E_1 is applied to E_2, and N has a link L to E_1, then, add the same link L from N to E_2, too.

Note: C_i = Component, E_i=Element, A_i=Action, O_i=Operation, W=Wildcard element, L=arbitrary link, N=arbitrary node

After extracting the semantics of a workflow, SemanticsAnalyzer examines if the semantics of the workflow satisfies the user's request by comparing the semantic graph representing the semantics of the workflow (i.e., the semantic graph emerged after applying *semantics retrieval rules*) and the semantic graph representing the request. If all the links of the semantic graph representing the request also appear in the semantic graph representing the semantics of the workflow, SemanticsAnalyzer concludes that the semantics of the workflow satisfies the request, and notifies ServiceComposer accordingly. ServiceComposer, then, presents the workflow to the user, and asks the user whether to execute the workflow or not. If the user replies positively, ServiceComposer passes the workflow to ServicePerformer, which in turn executes the workflow. If

[b] Please note that *semantics retrieval rules* consist of the *concepts* predefined in CoSMoS.

SemanticsAnalyzer concludes that the semantics of the workflow does not satisfy the request, or if the user replies negatively, ServiceComposer tries to create another workflow from the request.

In the example scenario described in Section 2.3.1, if SemanticsAnalyzer receives the workflow shown in Figure 11 (a) from ServiceComposer, it first converts the workflow into a semantic graph by connecting {'*HomeAddress*', '*originComp*'}, {'*RestAddress*', '*destComp*'}, and {'*dirImage*', '*inputImage*'} with '*usedBy*' links. SemanticsAnalyzer, then, applies the *semantics retrieval rules* onto the semantic graph converted from the workflow. This results in adding several new links, including "*object*('*Print*', '*Direction*')", "*from*('*Direction*', '*Home: home*')", and "*to*('*Direction*', '*Restaurant: demmy's*')", to the semantic graph. Figure 12 and Figure 13 illustrate how the *semantics retrieval rules* add two of the newly added links, "*object*('*Print*', '*Direction*')" and "*from*('*Direction*', '*Home: home*')" to the semantic graph. Since the newly added links also appear in the semantic graph representing the user's request (shown in Figure 10), SemanticsAnalyzer concludes that the semantics of the workflow satisfies the request.

Apply Rule 1:	performs('Operation print', 'Print') & outputs('Operation print, 'Component outPaper') & represents('Component outPaper', 'Wildcard') → object('Print', 'Wildcard')
Apply Rule 3:	usedBy('Component dirImage', 'Component inputImage') & represents('Component inputImage', 'Wildcard') & represents('Component dirImage', 'Direction') → applyTo('Wildcard', 'Direction')
Apply Rule 5:	applyTo('Wildcard', 'Direction') & object('Print', 'Wildcard') → object('Print', 'Direction')

Figure 12. An example showing how semantics retrieval rules are applied (1)

Apply Rule 2:	hasPropertyOf('Component Home', 'Component HomeAddress') & represents('Component Home', 'Home: home') & represents('Component HomeAddr', 'Location') → applyTo('Location', 'Home: home')
Apply Rule 3:	usedBy('Component HomeAddress', 'Component originComp') & represents('Component originComp', 'Location origin') & represents('Component HomeAddr', 'Location') → applyTo('Location origin', 'Location')
Apply Rule 5:	applyTo('Location origin', 'Location') & from('Direction', 'Location origin') → from('Direction', 'Location')
Apply Rule 5:	applyTo('Location', 'Home: home') & from('Direction', 'Location') → from('Direction', 'Home: home')

Figure 13. An example showing how semantics retrieval rules are applied (2)

On the other hand, however, if SemanticsAnalyzer receives the workflow in Figure 11 (b), it concludes that the semantics of the workflow does not satisfy the user request, because the links added by the *semantics retrieval rules* contain "*from*('*Direction*',

'*Restaurant: demmy's*')" and "*to*('*Direction*', '*Home: home*')" instead of "*from*('*Direction*', '*Home: home*')" and "*to*('*Direction*', '*Restaurant: demmy's*')". Thus, with the support of SemanticsAnalyzer, ServiceComposer concludes that only the workflow in Figure 11 (a) satisfies the request from the user.

ServicePerformer Module:

Upon receiving a workflow from ServiceComposer, ServicePerformer executes the workflow by invoking operations of the components and retrieving properties of the components as specified in the workflow. ServiceComposer accesses the Access Interface of CoRE to execute the workflow.

2.3.3. *SeGSeC and other Web Service Composition systems*

Several systems have been proposed and developed for composing an application through combining several Web Services. SeGSeC is unique compared to those existing systems in the following aspects.

The systems proposed in Refs. 3-10 compose an application using a template. A template is written either in a logical programming language such as Golog[3], as an OWL-S composite process[4], as a BPEL workflow[5], or in an original template description language[6]. Using a template, those systems compose an application through discovering the components necessary to convert (or instantiate) the template into an executable workflow. This approach requires a template for each and every application to be developed in advance, and thus severely limits the adaptability of the systems as it cannot compose new applications until new templates become available. This approach also requires a user to either create a template for an application that he/she requests or choose a template among those developed by application developers. This is not trivial for non-expert users as it requires the knowledge on the format of and the language used in a template. Unlike those template-based systems, SeGSeC synthesizes a workflow directly from the semantics of components when a user requests an application. This allows composing new applications without developing any templates, resulting in higher adaptability than the template-based systems.

The systems proposed in Refs.11-21 require a user to specify in his/her request the precondition and effect (a.k.a. post conditions) of an application that s/he requests. The precondition of an application may specify the input data that a user supplies with the application (e.g., Refs.13, 15), or may specify the initial condition of the application using First Order Logic (FOL) (e.g., Refs. 11, 17). Similarly, the effect of an application may specify the output data that a user expects from the application, or may specify the goal condition of the application using FOL. Given the precondition and effect of an application, those systems synthesize a workflow of the requested application through interconnecting the interfaces of the components in a network such that the synthesized workflow satisfies the specified precondition and effect. Unlike the template-based approach described previously, this approach does not require any template to be developed in advance. Thus, this approach allows composing new applications without developing any templates. However, specifying the precondition and effect of an application requires the knowledge regarding data types or FOL, and thus is not trivial for

non-expert users. SeGSeC, on the other hand, allows a user to request an application using a natural language, and this is more intuitive than choosing/creating a template or specifying the precondition/effect of an application. SeGSeC, thus, achieves higher usability than the systems proposed in Refs.11-21.

2.3.4. *Performance of SeGSeC*

The performance and scalability of SeGSeC were examined[23] through a series of empirical measurements. Due to space limitation, this paper briefly describes the results of the empirical measurements. See Ref. 23 for more detail discussion on the empirical measurements and additional measurements.

In order to evaluate the performance and scalability of SeGSeC, the average time for SeGSeC to compose applications was measured as a function of the number of components deployed in a network. In the measurements, 5 example applications (including the application explained in Section 2.3.2) were composed using up to 13 components deployed in a network.

The results of the empirical measurements show that SeGSeC composes applications in a reasonable time (i.e., less than a second) when the number of deployed components is small. The results also show that as the number of deployed components increases, the overhead of discovering components also increases significantly, and the overhead of the other processes of SeGSeC remains relatively constant. This implies that SeGSeC scales to the number of components deployed in a network provided that it can discover components efficiently.

3. WEB SERVICE BASED IMPLEMENTATION OF SEMANTICS-BASED DYNAMIC SERVICE COMPOSITION ARCHITECTURE

Section 2 summarized the design of the semantics-based dynamic service composition architecture. This section presents an implementation of the semantics-based dynamic service composition architecture based on the Web Service standards in order to validate the feasibility, portability and flexibility of the architecture. Web Service is the most practical implementation of the Service Oriented Computing and presents a good *platform* to implement the architecture.

The semantics-based dynamic service composition architecture has been implemented in Java using the Web Service standards, i.e., WSDL, UDDI and SOAP. The Resource Description Framework (RDF) is also used in implementing the architecture in order to describe the semantic information regarding the Web Services. The Web Service based implementation of the architecture is designed to satisfy the following requirements. First, the implementation maintains compatibility with existing Web Service based systems because it does not require any modification of the existing Web Service standards (i.e., WSDL/RDF/SOAP/UDDI). Second, the Web Service based implementation of the architecture allows existing Web Services to migrate to the architecture without reimplementation. Lastly, the Web Service based implementation of the architecture simplifies the development and deployment of a new Web Service by automatically generating the necessary description files (i.e., WSDL and RDF files) of

the Web Service from its runtime binary (i.e., a Java class file). The Web Service based implementation of the architecture is available for download at Ref. 32.

As described in Section 2, the semantics-based dynamic service composition architecture consists of CoSMoS, CoRE, and SeGSeC. The following subsections describe how CoSMoS and CoRE are implemented based on the Web Service standards. Once CoSMoS and CoRE are implemented based on the Web Service standards, SeGSeC is able to compose applications from the Web Services.

3.1. *CoSMoS for Web Services*

This section presents how to describe a component modeled by CoSMoS using WSDL[c] and RDF. As described in Section 2.1, CoSMoS models a component from three aspects: the functional aspect, the semantic aspect and the logical aspect. WSDL is used to describe the functional aspect of CoSMoS. RDF is used to describe the semantic and logical aspects of CoSMoS. A WSDL file describing the functional aspect and an RDF file describing the other two aspects are bound by another WSDL file (called a binding WSDL file) that *imports* those two files (See Figure 14). A binding WSDL file may bind a WSDL file that describes the functional aspect of an existing Web Service and a RDF file that is newly created to describe the semantic and logical aspects of the Web Service. This allows the existing (already deployed) Web Service to migrate to the semantics-based dynamic service composition architecture without reimplementation.

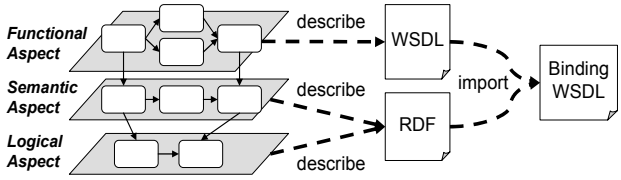


Figure 14. Describing CoSMoS in WSDL and RDF

3.1.1. *CoSMoS in WSDL*

WSDL defines an interface (called *portType*) of a Web Service as a set of *operations*. Each *operation* is defined as a pair of input and output *messages*, and each *message* is defined as a list of <name, data type> pairs called *parts*. WSDL may define data types using a schema language (e.g., XML Schema). WSDL may also specify how and where to access the interface using *binding* and *service* elements.

The functional aspect of CoSMoS can be described using WSDL in the following manner. An operation of a component in CoSMoS is defined as an *operation* in WSDL, whereas input and output components of an operation in CoSMoS are defined as *parts* that comprise the input and output *messages* of the *operation* in WSDL. Data types used in CoSMoS are defined in WSDL using XML Schema. The following is the WSDL

[c] This paper assumes WSDL version 1.1. However, the mapping between WSDL and CoSMoS described in this paper is also applicable onto WSDL version 2.0 with little modification.

description of the functional aspect of the Direction Generator component shown in Figure 1.

```
<wsdl:definitions …>
 <wsdl:types>
  <schema …>
   <complexType name="Address">
    <sequence>
     <element name="street" type="xsd:string"/>
     <element name="city" type="xsd:string"/>
     <element name="state" type="xsd:string"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>
 <wsdl:message name="output">
  <wsdl:part name="dirImage" type="xsd:base64Binary"/>
 </wsdl:message>
 <wsdl:message name="input">
  <wsdl:part name="originComp" type="impl:Address"/>
  <wsdl:part name="destComp" type="impl:Address"/>
 </wsdl:message>
 <wsdl:portType name="DirectionGenerator">
  <wsdl:operation name="generateOp" …>
   <wsdl:input message="impl:input" name="input"/>
   <wsdl:output message="impl:output" name="output"/>
  </wsdl:operation>
 </wsdl:portType>
 …
</wsdl:definitions>
```

In order to parse a WSDL file describing the functional aspect of a component (i.e., Web Service) into a CoSMoS semantic graph, a WSDL parser is implemented. Since no modification is made on WSDL to describe the functional aspect of CoSMoS, the WSDL parser can parse any regular WSDL files. However, WSDL cannot describe the semantic and logical aspects of CoSMoS. In order to complement WSDL, RDF is used to describe the semantic and logical aspects of CoSMoS.

3.1.2. *CoSMoS in RDF*

RDF describes a resource as a set of statements. A statement is a tuple of subject, predicate, and object. A subject identifies the resource that the statement describes. A predicate identifies the property or characteristics of the subject of the statement. An object identifies the value of the property of the statement. A set of statements in RDF is

often represented as a graph in which nodes represent the subjects and objects of the statements and arcs represent the predicates of the statements.

The semantic aspect of CoSMoS can be described using RDF in the following manner. An *Element* node (or an *Action* node) in the semantic aspect is defined as an RDF resource with a "rdf:type" predicate referring to the RDF resource "cosmos:Element" (or "cosmos:Action") and another "cosmos:concept" predicate specifying the URI of the concept the node represents. When a *Component* node (defined in the functional aspect) represents an *Element* node (defined in the semantic aspect), an RDF statement is defined such that its subject, predicate and object correspond to the *Component* node, the 'cosmos:represents' *concept*, and the *Element* node, respectively. Similarly, when an *Operation* node (defined in the functional aspect) *performs* an *Action* node (defined in the semantic aspect), an RDF statement is defined such that its subject, predicate and object correspond to the *Operation* node, the 'cosmos:performs' *concept*, and the *Action* node, respectively. A labeled link between two *Element* nodes is defined as an RDF statement whose subjects, predicates and objects correspond to the source *Element* node, the label of the link, and the target *Element* node, respectively. The following is the RDF description of the semantic aspect of the Direction Generator component show in Figure 1. This example uses the WordNet[31] as an ontology for defining the *concepts* of the *Element* node*s*.

```
<rdf:RDF ...>
  <rdf:Description rdf:about="#generateOp">
    <cosmos:performs rdf:resource="&wn[d];Generate"/>
  </rdf:Description>
  <rdf:Description rdf:about="#origin">
    <rdf:type rdf:resource="&cosmos;Element"/>
    <cosmos:concept rdf:resource="&wn;Location"/>
  </rdf:Description>
  <rdf:Description rdf:about="#originComp">
    <cosmos:represents rdf:resource="#origin"/>
  </rdf:Description>
  <rdf:Description rdf:about="#destination">
    <rdf:type rdf:resource="&cosmos;Element"/>
    <cosmos:concept rdf:resource="&wn;Location"/>
  </rdf:Description>
  <rdf:Description rdf:about="#destComp">
    <cosmos:represents rdf:resource="#destination"/>
  </rdf:Description>
  <rdf:Description rdf:about="#direction">
    <rdf:type rdf:resource="&cosmos;Element"/>
    <cosmos:concept rdf:resource="&wn;Direction"/>
    <en:from rdf:resource="#origin"/>
```

[d] "wn" is the namespace prefix representing" http://xmlns.com/wordnet/1.6/".

```
    <en:to rdf:resource="#destination"/>
  </rdf:Description>
  <rdf:Description rdf:about="#dirImage">
    <cosmos:represents rdf:resource="#direction"/>
    <cosmos:MIME>image/jpeg</cosmos:MIME>
  </rdf:Description>
</rdf:RDF>
```

Similarly, the logical aspect of CoSMoS can be described using RDF in the following manner. As described in Section 2.1.2, the logical aspect of CoSMoS defines rules, and each rule is a set of conditions and consequences. A *Rule* node in the logical aspect of CoSMoS is defined as an RDF statement whose subject, predicate and object correspond to the condition(s) of the rule, a "cosmos:implies" predicate, and the consequence(s) of the rule, respectively. Each condition and consequence, represented as a labeled link in CoSMoS, is defined as an RDF reification of a statement where the subject and object of the statement correspond to the source and target *Element* nodes of the link, and the predicate of the statement corresponds to the label of the link. If a rule contains multiple conditions or consequences, they are grouped into a single RDF resource using the RDF container "rdf:Bag". When a *Component* node (defined in the functional aspect) has a '*knows' link to* a *Rule* node (defined in the logical aspect), an RDF statement is defined such that its subject, predicate and object correspond to the *Component* node, the 'cosmos:knows' *concept*, and the *Rule* node, respectively. The following is the RDF description of the logical aspect of the Microphone component show in Figure 5.

```
<rdf:RDF ...>
  <rdf:Statement rdf:ID="cond">
    <rdf:subject rdf:resource="#microphone"/>
    <rdf:predicate rdf:resource="&wn;Use"/>
    <rdf:object rdf:resource="#person"/>
  </rdf:Statement>
  <rdf:Statement rdf:ID="cons">
    <rdf:subject rdf:resource="#sound"/>
    <rdf:predicate rdf:resource="&wn;Of"/>
    <rdf:object rdf:resource="#person"/>
  </rdf:Statement>
  <rdf:Statement rdf:ID="rule1">
    <rdf:subject rdf:resource="#cond"/>
    <rdf:predicate rdf:resource="&cosmos;implies"/>
    <rdf:object rdf:resource="#cons"/>
  </rdf:Statement>
  <rdf:Description rdf:about="#Microphone">
    <cosmos:knows rdf:resource="#rule1"/>
  </rdf:Description>
</rdf:RDF>
```

In order to parse an RDF file describing the semantic and logical aspects of a component (i.e., Web Service) into a CoSMoS semantic graph, an RDF parser is implemented. The RDF parser collaborates with the WSDL parser described in Section 3.1.2 so that when the WSDL parser parses a binding WSDL file it can relay the imported RDF file to the RDF parser.

3.1.3. *Automatic Generation of WSDL and RDF*

In order to ease the development of a new Web Service modeled by CoSMoS, the Web Service based implementation of the semantics-based dynamic service composition architecture supports the automatic generation of the WSDL and RDF files from an annotated Java class file. Annotation, a feature of Java 1.5, enables to annotate and embed the semantic and logical aspects of CoSMoS into a Java class file. The following is an example Java source code of the Direction Generator component in Figure 1.

```
public class DirectionGenerator {
  @Prefix  … String wn = " http://xmlns.com/wordnet/1.6/";
  @Element … String origin = "&wn;Location";
  @Element … String destination = "&wn;Location";
  @Element … String direction = "&wn;Direction";

  @Link(from="direction",to="origin")
  static final String from="&wn;from";

  @Link(from="direction",to="destination")
  static final String to="&wn;to";

  @Action("&wn;Generate")
  @Return("direction") @Binary("image/jpeg")
  byte[] generate(
     @Param("origin") Address originComp,
     @Param("destination") Address destComp) {…}
}
```

Apache Axis, a library for Web Service, is used to automatically generate a WSDL file from a Java class file. Two servlets, RDF Generator and Binding WSDL Generator, are implemented in order to generate an RDF file and a binding WSDL file automatically from an annotated Java class file. This automatic generation of WSDL and RDF greatly simplifies the development of a new Web Service for the semantics-based dynamic service composition architecture.

3.2. *CoRE for Web Services*

This section describes how to publish Web Services modeled by CoSMoS onto a UDDI repository, and how to allow CoRE to discover and invoke the Web Services using UDDI and SOAP. As described in Section 2.2, CoRE consists of DiscoveryEngines, InvokerEngines, and PropertyAccessEngines. In order to allow CoRE to discover Web Services modeled by CoSMoS, a DiscoveryEngine based on UDDI is implemented. In order to allow CoRE to invoke Web Services modeled by CoSMoS, an InvokerEngine based on SOAP is implemented. Please note that PropertyAccessEngine is not implemented for Web Services because Web Services cannot expose any property. Figure 15 shows the architecture of CoRE that is implemented using UDDI and SOAP.

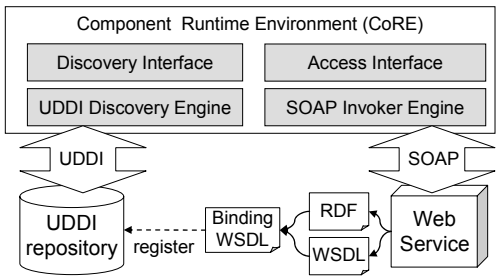


Figure 15. Implementing CoRE using UDDI and SOAP

3.2.1. *Web Service Publication onto UDDI*

UDDI specifies a set of APIs and protocols that enables a client to publish and discover Web Services. In order to organize Web Services, UDDI defines the following elements: *businessEntry*, *businessService*, *bindingTemplate* and *tModel (Technical Model)*. A *businessEntry* represents a physical company and contains multiple *businessServices*, which represent the (web) services provided by the company. Each *businessService* may contain multiple *bindingTemplates*, the instructions on how to invoke the service. Each *bindingTemplate* specifies the access point of the service and also may specify several *tModels*. A *tModel* may specify the technical specification of the *bindingTemplate* using WSDL.

Since UDDI only supports business-related information for organizing Web Services, it cannot organize or lookup Web Services based on their semantic information. Although it is possible to extend UDDI to directly support the semantic information of the Web Services (e.g., Ref. 33), this paper proposes an approach that requires no modification of existing UDDI repositories in order to maintain compatibility.

As described in Section 3.1, a Web Service modeled by CoSMoS is described by a binding WSDL file that *imports* another WSDL file describing the functional aspect of the component and an RDF file describing the semantic and logical aspects of the component. One can publish a Web Service modeled by CoSMoS onto a UDDI repository by (1) registering the binding WSDL file of the Web Service as a *tModel*, and (2) specifying the *tModel* (i.e., the registered binding WSDL file) in the *bindingTemplate*

of the Web Service. Since no modification is made on UDDI, this approach is applicable to any existing UDDI repositories.

3.2.2. *Web Service Discovery from UDDI*

In order to allow CoRE to discover the Web Services modeled by CoSMoS, a DiscoveryEngine based on UDDI is developed. The DiscoveryEngine first retrieves all *tModels* specifying the binding WSDL files from UDDI and parses them into CoSMoS semantic graphs by using the WSDL and RDF parsers described in Section 3.1. Then, upon receiving a query from the Discovery Interface, the DiscoveryEngine evaluates the parsed CoSMoS semantic graphs of the Web Services against the query, and identifies which Web Services match the query. After identifying the Web Services that match the query, the DiscoveryEngine retrieves the access points (i.e., URLs) of the Web Services by retrieving their *bindingTemplates* from UDDI, and returns the pairs of the CoSMoS semantic graphs and the access points of the Web Services to the Discovery Interface.

3.2.3. *Web Service Invocation via SOAP*

SOAP is an XML-based message exchange protocol, which is a de facto standard for invoking Web Services on remote hosts. In order to allow CoRE to invoke the Web Services modeled by CoSMoS, an InvokerEngine based on SOAP is developed. Upon receiving a request to invoke a Web Service from the Access Interface, the InvokerEngine obtains the following information regarding the Web Service: its access point (i.e., URL), its operation name, the data types of its input(s) and output(s), and the actual input data (i.e., arguments). The access point of a Web Service is provided by the DiscoveryEngine when it discovers the Web Service from UDDI. The operation name of the Web Service is identified when SeGSeC synthesizes a workflow containing the Web Service. The data types of the input(s) and output(s) are obtainable from the functional aspect of the Web Service. The actual input data is provided when SeGSeC executes the workflow. With completed information, the InvokerEngine invokes the Web Service by sending the request encoded as an XML message to the specified access point, and returning the return value from the Web Service back to the Access Interface.

3.3. *Empirical Evaluation*

In order to test and evaluate the Web Service based implementation of the semantics-based dynamic service composition architecture empirically, various components are developed and deployed onto the implementation, and several applications are composed using those components. Since the performance of the architecture has already been examined[23], this paper focuses on the feasibility, compatibility and portability of the Web Service based implementation of the architecture.

Table 2 summarizes the components developed for and deployed onto the Web Service based implementation of the architecture. Some components are developed as annotated Java class files and deployed as Web Services. Some other components are developed by annotating the WSDL files of the existing Web Services with new RDF

files. There are also several components that are not deployed as Web Services, such as device components and data components.

Table 2. Components developed for the Web Service based implementation of the architecture

	Components developed
Original Web Services	Direction generator, Business directory[a], Text-to-sound converter, Sound-to-text converter, Business card creator[b]
Existing Web Services	Email sending service[34], Fax sending service[35], SMS sending service[36], Instant messaging sending service[37], Zip code lookup[38], Distance calculator[39,c]
Device components	Keyboard, display, microphone, speaker
Data components	Home, Office, Restaurant, Tom, Alice

[a] Given a name of a person, it returns the address, phone number, homepage and email address of the person.
[b] Given a name, address, phone number, email address and homepage of a person, it creates an image of a business card of the person.
[c] Given two zip codes, it calculates the distance between the two zip codes.

The client software of the architecture (Figure 16) allows a user to deploy different sets of components and request an application using a natural language. Based on the request from the user, the client software performs SeGSeC and composes various applications using the components deployed. For instance, it can compose the direction printing service described in Section 2.3.

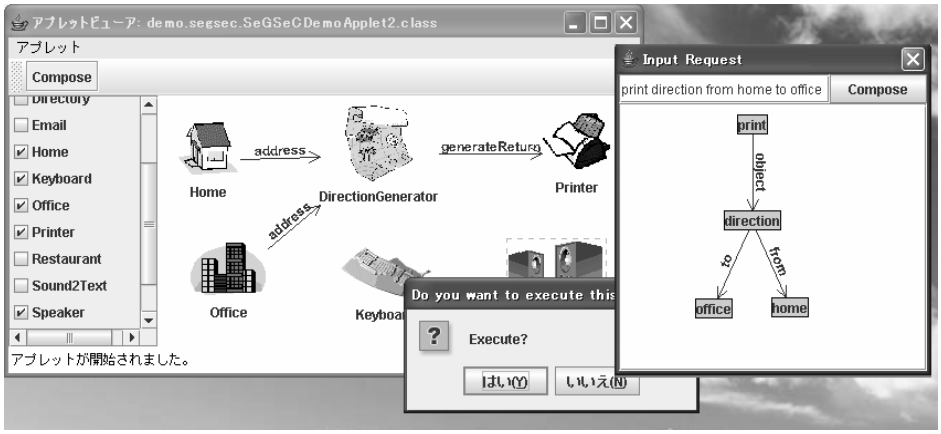
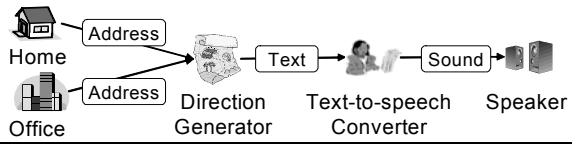
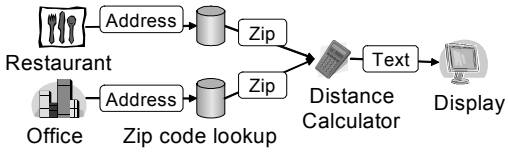
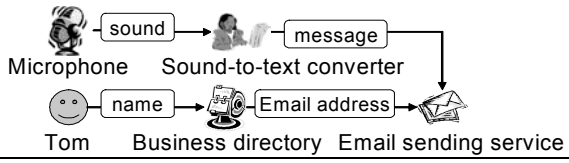
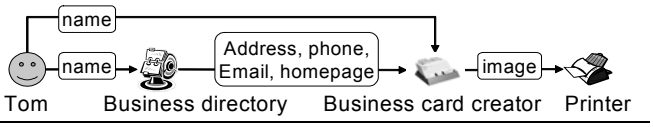


Figure 16. The user interface of the Web Service based implementation of the architecture

Table 3 shows some other example applications that can be composed by the architecture.

Table 3. Example applications that can be composed by the architecture

Request	Composed application
Play direction from home to office	
Show distance between office and restaurant	
Send email to Tom	
Print businesscard of Tom	

This empirical evaluation confirmed that the semantics-based dynamic service composition architecture can be implemented using Web Service standards without any modification, and thus demonstrated that the Web Service based implementation of the architecture maintains compatibility with other Web Service based systems. It also verified that the Web Service based implementation of the architecture allows the existing Web Services to migrate to the architecture without reimplementation by annotating the WSDL files of the existing Web Services with new RDF files. The empirical evaluation also demonstrated that a new Web Service for the architecture can be developed easily as the Web Service based implementation of the architecture is capable of automatically deploying an annotated Java class file as a Web Service for the architecture by automatically generating the necessary description files (i.e., WSDL and RDF files). Most importantly, the empirical evaluation verified that the concept of the semantics-based dynamic service composition, i.e., composing applications based on the semantics of the components and of the user's request, is applicable to the Web Service domain.

4. CONCLUSION AND FUTURE WORK

This paper presents the semantics-based dynamic service composition architecture, which dynamically composes the application requested by a user based on the semantics of the components and of the user request. The architecture consists of a semantics-aware component model called Component Service Model with Semantic (CoSMoS), a middleware called Component Runtime Environment (CoRE), and a semantics-based dynamic service composition mechanism called Semantic Graph based Service Composition (SeGSeC). This paper describes the latest design of the architecture and also illustrates the Web Service based implementation of the architecture. Through the empirical evaluation, this paper confirmed that the Web Service based implementation of the architecture satisfies the following requirements. First, the Web Service based implementation of the architecture maintains compatibility with existing Web Service based systems because it does not require any modification of the existing Web Service standards (i.e., WSDL/RDF/SOAP/UDDI). Second, the Web Service based implementation of the architecture allows existing Web Services to migrate to the architecture without reimplementation. Lastly, the Web Service based implementation of the architecture simplifies the development and deployment of a new Web Service by automatically generating the necessary description files (i.e., WSDL and RDF files) of the Web Service from its runtime binary (i.e., a Java class file). The empirical evaluation using the Web Service based implementation of the architecture also verified that the concept of the semantics-based dynamic service composition is applicable to the Web Service domain.

The semantics-based dynamic service composition architecture may be extended to compose applications not only based on the semantics of the components and of the user request but also based on the user's context information such as location, time, or history in order to provide more adaptable applications. This awaits further research.

Acknowledgements

This research is supported by the NSF through grants ANI-0083074, ANI-9903427 and ANI-0508506, by DARPA through grant MDA972-99-1-0007, by AFOSR through grant MURI F49620-00-1-0330, and by grants from the California MICRO and CoRe programs, Hitachi, Hitachi America, Hitachi CRL, Hitachi SDL, DENSO IT Laboratory, DENSO International America LA Laboratories, NICT (National Institute of Communication Technology, Japan), NTT Docomo and Novell.

References

1. M.P. Papazoglou and D. Georgakopoulos, Service Oriented Computing, in *Comm. ACM*, vol. 46, no. 10, 2003, pp. 25–28.
2. Chakraborty, D. and Joshi, A Dynamic Service Composition: State-of-the-Art and Research Directions, *Technical Report TR-CS-01-19*, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, USA, 2001.
3. S. McIlraith and T. Son, Adapting Golog for Composition of Semantic Web Services, in *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482-493, April, 2002.
4. D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha, A Reactive Service Composition Architecture for Pervasive Computing Environments, in *Proceedings of the 7th Personal Wireless Communications Conference (PWC 2002)*, Singapore, October 2002.
5. K. Sivashanmugam, J. Miller, A. Sheth, and K. Verma, Framework for Semantic Web Process Composition, *International Journal of Electronic Commerce*, Winter 2004-5, Vol. 9(2) pp. 71-106.
6. J. Cardoso and A. Sheth, Semantic e-Workflow Composition, to appear in *Journal of Intelligent Information Systems*, 2003.
7. Q. Z. Sheng, B. Benatallah, M. Dumas, and E. Mak, SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment, in *Proceedings of the twenty-eighth Very Large DataBase Conference (VLDB'2002)*, Hong Kong, China, August 2002.
8. P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma, Dynamic Workflow Composition using Markov Decision Processes, in *Proceedings of the Second International Conference on Web Services (ICWS)*, pp. 576-582, San Diego, CA, July 6-9, 2004.
9. P. Pires, M. Mattoso, M. Benevides, Building Reliable Web Services Compositions, *Web, Web-Services, and Database Systems 2002*. Springer LNCS 2593, ISBN 3-540-00745-8, pp. 59-72, 2003.
10. P. Traverso and M. Pistore, Automated Composition of Semantic Web Services into Executable Processes, in *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, 7-11 Nov. 2004, Hiroshima, Japan.
11. S. R. Ponnekanti and A. Fox, SWORD: A Developer Toolkit for Web Service Composition, in *Proceedings WWW Conference (11)*, Honolulu, Hawaii, May 7-11, 2002.
12. E. Sirin, J. Hendler, and B. Parsia, Semi-automatic composition of web services using semantic descriptions, in *Proceedings of the Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, Angers, France, April 2003.
13. W. Cheung, J. Liu, K. Tsang, R. Wong, Towards Autonomous Service Composition in A Grid Environment, in *Proceedings of the 2004 IEEE International Conference on Web Services*, San Diego, California, July, 2004.
14. M. Carman, L. Serafini and P. Traverso, Web Service Composition as Planning, in *Proceedings of the ICAPS 2003 Workshop on Planning for Web Services*, June 2003.
15. M., Takahiro Kawamura, T. R. Payne, and K. Sycara, Semantic Matching of Web Services Capabilities, in *Proceedings of the 1st International Semantic Web Conference (ISWC2002)*, Sardinia, Italy.
16. J. Yang and M. Papazoglou, Web Components: A Substrate for Web Service Reuse and Composition, in *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, Toronto, Canada, 2002.
17. J. Peer, Towards Automatic Web Service Composition using AI Planning Techniques, 2003.
18. J. Hendler, D. Wu, E. Sirin, D. Nau, and B. Parsia, Automatic web services composition using Shop2, In *Proceedings of The Second International Semantic Web Conference (ISWC 2003)*, Sundial Resort, Sanibel Island, Florida, USA, 2003.

19. B. Limthanmaphon and Y. Zhang, Web Service Composition with Case-Based Reasoning, In *Proceedings of the Fourteenth Australasian Database Conference (ADC2003)*, Adelaide, Australia. CRPIT, 17. Schewe, K.-D. and Zhou, X., Eds., ACS. 201-208.
20. M. Sheshagiri, M. desJardins, and T. Finin, A Planner for Composing Services Described in DAML-S, *in Proceedings of the ICAPS 2003 Workshop on Planning for Web Services*, July 2003.
21. J. Rao, P. Kungas and M. Matskin. Logic-based Web Service Composition: from Service Description to Process Model, In *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS 2004)*, San Diego, California, USA, July 6-9, 2004.
22. K. Fujii and T. Suda, Dynamic Service Composition Using Semantic Information, In *Proceedings of the Second International Conference on Service Oriented Computing (ICSOC '04)*, November 2004.
23. K. Fujii and T. Suda, Semantics-based Dynamic Service Composition, to appear in the IEEE Journal on Selected Areas in Communications (JSAC), special issue on Autonomic Communication Systems.
24. J. F. Sowa, Conceptual Graphs Summary, in *Conceptual Structures: Current Research and Practice*, P. Eklund, T. Nagle, J. Nagle, and L. Gerholz, eds., Ellis Horwood, 1992, pp. 3-52.
25. OWL Service Coalition, OWL-S 1.0 Release, http://www.daml.org/services/owl-s/1.0/.
26. R. Akkiraju et al., Web Service Semantics - WSDL-S (2005), http://www.w3.org/Submission/WSDL-S/.
27. A. Eberhart, Ad-hoc Invocation of Semantic Web Services, in Proceedings of the IEEE International Conference on Web Services (ICWS 2004), July 2004.
28. J. Peer. Semantic Service Markup with SESMA, in Proceeding of Web Service Semantics Workshop (WSS'05) at the Fourteenth International World Wide Web Conference (WWW'05), 2005.
29. Web Service Modeling Ontology, http://www.wsmo.org/.
30. G.A. Mann, BEELINE - A Situated, Bounded Conceptual Knowledge System, *International Journal of Systems Research and Information Science*, 1995, 7, pp37-53.
31. D. Brickley, Wordnet for the Web, http://xmlns.com/2001/08/wordnet/
32. K. Fujii, Dynamic Service Composition (2005), http://netresearch.ics.uci.edu/kfujii/dsc/
33. N. Srinivasan, M. Paolucci, and K. Sycara, Adding OWL-S to UDDI, implementation and throughput, in *the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004*), 2004, San Diego, California, USA.
34. Email sending service, http://www.abysal.com/soap/AbysalEmail.wsdl
35. Fax sending service, http://www.webservicex.com/fax.asmx?wsdl
36. SMS sending service, http://www.webservicex.com/sendsmsworld.asmx?WSDL
37. Instance messaging sending service, http://www.scdi.org/~avernet/webservice/yim.wsdl
38. Zip code lookup service, http://www.webservicex.com/uszip.asmx?WSDL
39. Distance calculator, http://webservices.imacination.com/distance/Distance.jws?wsdl