

Dynamic Service Composition Using Semantic Information

Keita Fujii

School of Information and Computer Science

University of California

Irvine, CA 92697, USA

+1-949-824-4105

kfujii@ics.uci.edu

Tatsuya Suda

School of Information and Computer Science

University of California

Irvine, CA, 92697 USA

+1-949-824-4105

suda@ics.uci.edu

ABSTRACT

Dynamic composition of complex services from primitive components brings flexibility and adaptability to future applications. By properly selecting and combining components on demand, applications would adapt to individual user preference and would consider available context information.

Existing service composition systems often require users to request services in strict syntax formats, such as data types, service templates or logic formulas. This requirement may become an obstacle for end-users to use such systems. Instead, service composition should be semantics-based so that a service is requested and composed not by its syntax but by its semantics.

In order to enable semantics-based dynamic service composition, both the modeling of components as well as the service composition mechanism must support semantics. To satisfy the requirement of semantic support in the component modeling, we have designed a new model named Component Service Model with Semantics (CoSMoS). CoSMoS integrates the semantic information of a component and the functional information of a component into a single semantic graph representation. A unified interface named Component Runtime Environment (CoRE) is developed to convert different component implementations onto the CoSMoS representation. Using the semantic support of CoSMoS, we have developed a semantics-based service composition mechanism named Semantic Graph based Service Composition (SeGSeC). SeGSeC generates the execution path of the requested service, and checks the semantics of the path against the request. We have implemented a service composition system using the above techniques, and demonstrated that our system supports semantics-based dynamic service composition.

Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Automatic Programming – *program synthesis*; I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods – *Semantic networks*; D.2.10 [Software Engineering]: Design – *Representation*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSO'04, November 15–19, 2004, New York, New York, USA.

Copyright 2004 ACM 1-58113-871-7/04/0011...\$5.00.

General Terms

Algorithms, Design

Keywords

Dynamic service composition, component model, semantic graph, CoSMoS, CoRE, SeGSeC

1. INTRODUCTION

Distributed component technologies such as CORBA and Web Service bring a new vision of a future network environment where a large number of components representing software services, devices, or resources are distributed and transparently accessible. In such an environment, a new application may be composed from a set of components. This concept of composing complex services (or applications) from primitive components is called service composition. Service composition enables quick development of new application functionality through the reuse of the components for multiple compositions [14].

Service composition techniques can be categorized into two types: static service composition, and dynamic service composition [5]. Static (or proactive) service composition is an approach in which application designers implement a new application manually by designing a workflow or a state chart describing the interaction pattern among components. BPEL4WS [2] or WSCI [17], for example, are primarily designed for supporting this approach. The static service composition supports applications involving complex interaction patterns, such as branch or iteration, but requires those applications to be manually designed before being deployed. Therefore, the static service composition is suitable for B2B type applications where interactions among components are often complex but static and easy to provision.

Dynamic (or reactive) service composition, on the other hand, composes an application autonomously when a user queries for an application. eFlow [4] and SWORD [13] are the examples of dynamic service composition systems. Because the dynamic service composition does not depend on a human to compose an application, it may have difficulty in composing applications with complex interaction patterns. Nevertheless, the dynamic service composition has the potential to realize flexible and adaptable applications by properly selecting and combining components based on the user request and context. The dynamic service composition may also elicit a number of useful applications that are not envisioned at the design time. Therefore, the dynamic service composition is suitable for end-user applications in

ubiquitous, mobile environments where available components are dynamic and expected users may vary. Since we believe that the dynamic service composition is the key technology for enabling future ubiquitous applications, this paper focuses its topics on the dynamic service composition.

To illustrate the benefit and issues of the dynamic service composition, let us give an example scenario here. Suppose Tom wants to take his family to a restaurant he recently found on the web. Since he is unsure of the direction from his home to the restaurant, he decides to print out and carry the directions with him. Assume that 1) the restaurant’s Web server stores the restaurant’s information such as its address or menus in a structured document (e.g., in XML), 2) Tom’s PC stores the information about Tom and his house, such as its address, phone number etc, in a database, 3) there is a Web service which, given two address information, generates an image showing the direction from one address to another, and 4) Tom has a printer which is connected to his home network. With the current technologies, Tom can print out the direction from his home to the restaurant. However, he has to perform each step, i.e. get the address of the restaurant and his home, invoke the direction generator Web service, and print out the resulted image, manually by himself. If the dynamic service composition is available, Tom can simply type (or speak) a command “print out a direction from home to the restaurant” onto his PC, and he can receive the printed direction as he requested.

There already exist several dynamic service composition systems researched and developed (e.g. eFlow, STONE). However, those existing systems often require users to request services in strict syntax formats, such as data types (e.g. string, image/jpeg, etc), service templates (e.g. pass address data to a direction service) or logic formulas (e.g. address (X) & address(Y) \rightarrow direction(X,Y)). This requirement may become an obstacle for end-users to use such systems. Instead of using such syntax information, a user should be able to request a service using its semantics, and the dynamic service composition should be able to compose the requested service based on the semantics of the user request.

In order to enable semantics-based dynamic service composition, both modeling of components as well as the dynamic service composition mechanism must support semantics. A component model must be able to represent not only the functional information of a component (i.e. data type, input/output etc) but also the semantic information of a component (i.e. what component represents etc). At the same time, such a component model should keep as much compatibility as possible with existing component technologies. Also, a dynamic service composition mechanism should allow a user to request a service by its semantics, and should be able to compose a service based on the semantics of the user request by taking advantage of the semantic representation of the component model.

To satisfy the above requirements, we have developed (1) Component Service Model with Semantic (CoSMoS), (2) Component Runtime Environment (CoRE), and (3) Semantic Graph based Service Composition (SeGSeC). CoSMoS integrates the semantic information of a component and the functional information of a component into a single semantic graph representation. CoRE acts as middleware to convert different component implementations onto CoSMoS representation. SeGSeC allows a user to request a service using a natural

language sentence, and generates the execution path of the requested service based on the user request. SeGSeC also performs semantic matching to confirm that the semantics of the generated path matches the semantics of the user request. Figure 1 shows the architecture of our semantics-based dynamic service composition system.

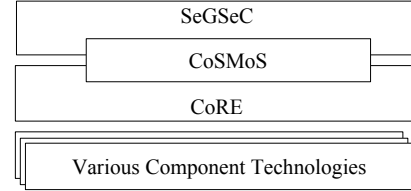


Figure 1. Architecture of the proposed system

The remaining sections of this paper will give details of the proposed work as follows. Section 2 describes CoSMoS. Section 3 presents CoRE. Section 4 illustrates SeGSeC. Section 5 concludes this paper.

2. COMPONENT SERVICE MODEL WITH SEMANTICS (CoSMoS)

This section introduces a new component model named Component Service Model with Semantics (CoSMoS). CoSMoS is an abstract model that represents a component as a single semantic graph. CoSMoS is designed to integrate the functional information of a component and the semantic and logical aspects of the component. This section first gives an overview of CoSMoS, which is followed by the detailed design of CoSMoS.

2.1 CoSMoS Overview

2.1.1 CoSMoS Domains

Many existing component models (e.g. WSDL, JavaBeans/EJB, COM, CCM etc) often lack the capability of representing the semantics of components. Therefore, those models require human application designers to understand, judge and match semantics of components, which prevents autonomous composition of applications. Instead, a component model should be able to represent not only the functional information but also the semantics information of a component. In order to satisfy this requirement, CoSMoS defines four domains, namely, data type domain, semantics domain, logic domain, and component domain (Figure 2).

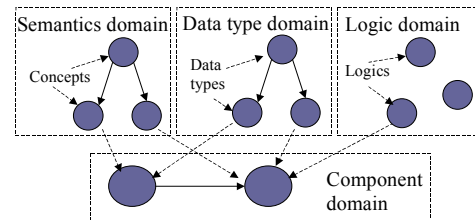


Figure 2. CoSMoS domains

Each domain captures a different aspect of a component. The data type domain defines *data types*, i.e., data formats of components, and relationships among *data types*. The semantics domain defines *concepts*, i.e., entities representing abstract ideas or actions, and ontologies, i.e., relationships among *concepts*. The

logic domain defines *logics*, i.e., knowledge or rules about components' service. The component domain defines *components* by combining *data types*, *concepts* and *logics* defined in the other domains. The component domain also defines *operations* and *properties of components*.

2.1.2 Semantic graph representation

In order to represent and combine information in different domains in a coherent way, CoSMoS applies semantic graph representation. In CoSMoS, a component is represented as a graph of nodes and links, where nodes represent *data types*, *concepts*, *logics*, or *components*, and links represent relationships between the nodes. Each node or link is identified by its URI. A link is associated with a *concept*, which represents the semantics of the link. The *concepts* especially defined as the semantics of link are called *link types*.

2.2 CoSMoS Domains

This section describes details of the four domains defined in CoSMoS using the CoSMoS class diagram shown in Figure 3.

2.2.1 Data Type Domain

Data type domain defines *data types*, i.e., data formats of components. CoSMoS predefines several classes to support commonly used *data types*, namely, primitives (i.e. integer, string, float, Boolean), arrays, structured data, binary data, and files. Also, CoSMoS allows component designers to define arbitrary *data types* by using *DataType* class and two *link types*, *isCompatibleWith* and *subTypeOf*. For instance, the *data types* `java.util.Map` and `java.util.HashMap` in Java's collection library can be defined as *DataType* instances named "java.util.Map" and "java.util.HashMap" connected by a *subTypeOf* link. In this manner, CoSMoS is capable of defining any implementation-specific *data types* (such as Java collection library or XML Schema Datatypes) that cannot be represented by the predefined classes, and also the compatibilities among those *data types*.

2.2.2 Semantics Domain

Semantics domain defines *concepts*, which are entities representing abstract ideas or actions. For example, 'Person', 'Animal', 'generate', and 'print' can be defined as *concepts*. A *concept* is defined as an instance of *Concept* class, and is used to annotate the semantics of a component or an operation. Two subclasses, *Noun* and *Predicate*, are defined for annotating the semantics of components and operations, respectively.

A *concept* can be a 'wildcard', meaning that the *concept* can be substituted by any other *concept*. To illustrate the usage of a *wildcard concept*, consider a Text-to-Speech converter service. The semantics of the input text and output sound data of the Text-to-Speech converter service cannot be predetermined. If the input text represents a direction, the output sound data also represents the direction. If the input text represents a novel, the output sound also represents the novel. A *wildcard concept* can be used in such a case to represent that the input text and the output sound data represent the same but not predetermined *concept* (Figure 4).

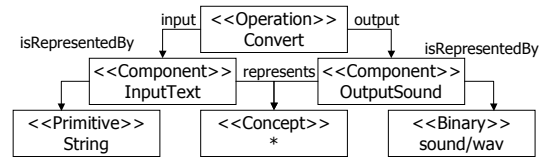


Figure 4. Example of a wildcard concept

CoSMoS also supports ontologies [1], i.e., relationships among *concepts*, by defining two *link types*: *subConceptOf* and *isEquivalentTo*. For instance, an ontology "Human is a kind of Animal" can be defined as a *subConceptOf* link connecting 'Human' *Concept* node and 'Animal' *Concept* node. The support of ontology in CoSMoS allows component designers to arbitrarily define *concepts* and associate them with other existing ones.

2.2.3 Logic Domain

Logic domain defines *logics*, i.e., knowledge or rules about components' service. CoSMoS defines *Logic* class, a subclass of *Link* class, and three *link types*, *implies*, *and*, and *or*, for defining *logics*. *Logic* is represented as a link, i.e., an instance of *Logic* class with *implies* link, connecting two arbitrary links, each of which represents a condition as a tuple of <subject, predicate, object>. For instance, the logic "a person watches a movie" *implies* "the TV plays the movie" can be represented as in Figure 5. *and* and *or* links are used for defining Boolean expressions.

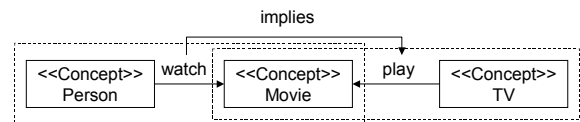


Figure 5. Example of logic

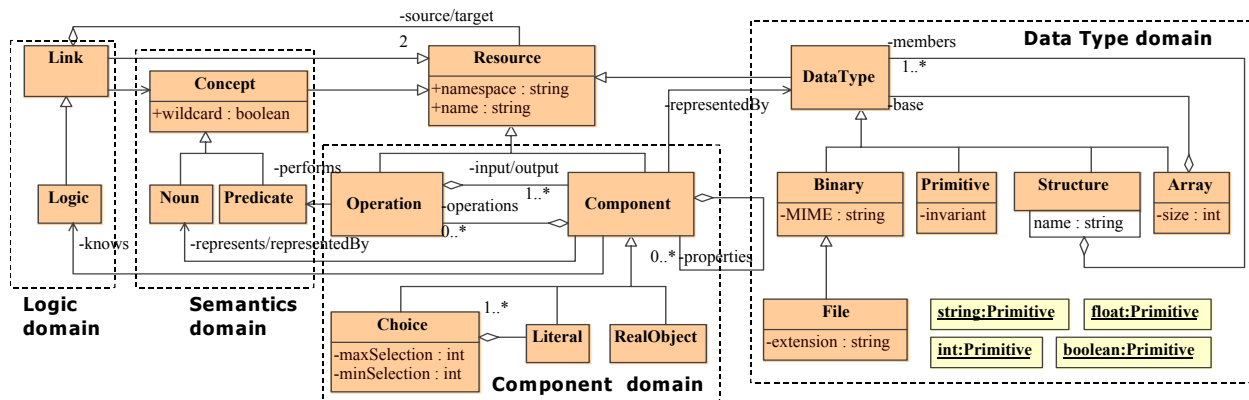


Figure 3. CoSMoS class diagram

Logics defined in the logic domain are used for reasoning in the service composition process. Details of how logics are used in the composition process are described later in Section 4.

2.2.4 Component Domain

Component domain defines *components*. CoSMoS defines a *component* as a combination of its *data type*, *concepts*, *logics*, *operations* and *properties*. A *component* may have its *data type* (e.g. ‘String’) and/or a *concept* (‘English’) representing the format of the *component*. The *isRepresentedBy* link is used to connect a *component* with its *data type* or *concept* representing the format of the *component*. A *component* may also have a *concept* (e.g. ‘Direction’) to represent the semantics of the *component*. The *represents* link is used to connect a *component* with a *concept* representing the semantics of the component. For example, a *component* representing ‘Direction’ in ‘English’ as a ‘String’ value can be modeled as Figure 6. A *component* may also have *logics* defined in the logic domain. The *knows* link is used to link a component with its logics.

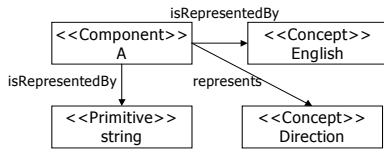


Figure 6. Example of a component

A *component* may have one or more *operations*, similar to functions, methods, or interfaces in programming languages. An *operation* is defined as an instance of *Operation* class, and consists of input *component(s)* and output *component(s)*. An *operation* may have a *predicate*, a subtype of a *concept*, to represent the semantics of the operation. ‘print’ or ‘generate’ are the example of *predicates*.

A *component* may have one or more *properties*. A *property* is a *component* representing the attribute of another *component*. For example, a ‘Home’ *component* may have properties ‘Address’, ‘Phone number’ and so on. *Properties* can be mapped onto tagged elements in an XML document, or cells in an entry in Relational Database.

CoSMoS defines several subclasses of *Component* class, namely, *RealObject*, *Choice*, and *Literal*, for special purposes. *RealObject* class is defined to represent an object in the real world, such as a printer or a paper. *Literal* class is defined to represent a constant string value (e.g. “A4”, “A5” etc) and its *concept*. *Choice* class is defined to model a service that asks a user to choose one or more items from a list of items. For example, suppose there is a printer service that allows a user to select the paper size among A4, B4, B5, and Letter size. Such a printer service can be modeled in CoSMoS by using a *Choice* component ‘paperSize’ with *Literal* components “A4”, “B4”, “A5”, and “Letter” as its candidates.

2.3 Examples

Figure 7 and Figure 8 are the examples of CoSMoS representation. Figure 7 shows a direction generator component that generates a graphical map and a string value describing a direction from one address to another. Figure 8 shows a printer component that accepts and prints out a jpeg image.

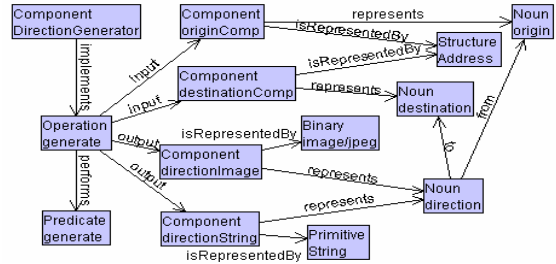


Figure 7. Direction generator component in CoSMoS

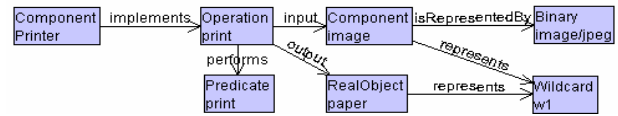


Figure 8. Printer component in CoSMoS

2.4 Comparison to existing work

This section describes the features of CoSMoS, compared with other existing component models.

2.4.1 Support of semantics

Many existing component models (e.g. WSDL, JavaBeans/EJB, COM, CCM etc) do not specify how to represent or model the semantics of components. Due to the lack of semantic information, those models require human application designers to understand, judge and match the semantics of components. On the other hand, CoSMoS supports the representation of semantics by defining *Concept* and *Logic* classes, allowing machines to autonomously retrieve and match semantics of components.

2.4.2 Expression of semantics

Carman et al. [3] proposes an approach to retrieve semantic information of a WSDL description using WordNet. Although their approach provides *concepts* of the elements described in WSDL, it cannot supply the relationships among the retrieved *concepts*. As shown as the example in Figure 7, CoSMoS is capable of modeling the relationships among *concepts*, providing richer semantics than a mere aggregation of *concepts*.

OWL-S [12], an OWL-based Web service ontology, is attracting attention from Semantic Web Service communities (e.g. [16][18]). OWL-S is claimed to support semantics of a service using ontologies defined by OWL. However, OWL-S does not clearly distinguish *data types* (i.e. XML Schema Datatypes) and *concepts* (instances of owl:Class), as both can be used as parameter types of input/output. Because of this, OWL-S has difficulty in modeling services in which an input and an output have different *data types* but share the same *concept*, like the example shown in Figure 4. In order to model such services, OWL-S has to define auxiliary classes and ontologies, which is not a straightforward option. CoSMoS, on the other hand, explicitly separates the definition of *data types* and *concepts*, and thus directly supports such services by allowing *components* with different *data types* to represent the same *concept*.

2.4.3 Support of semantics of operation

Many of the existing component models, including OWL-S, do not mark up the semantics of operations. Some models define an operation as a set of inputs and outputs, while others define an operation as a set of inputs, outputs, precondition, and

postcondition. In CoSMoS, an operation may be associated with a *predicate* (a subtype of *concept*) to represent its semantics, which allows more precise definition of the operation's semantics than a set of inputs, outputs and pre/postconditions.

2.5 Implementation

In order to verify the feasibility of CoSMoS, we designed several description languages for CoSMoS, and also implemented CoSMoS in Java.

Since CoSMoS is an abstract model, it is possible to describe CoSMoS in various ways, i.e., by using an original description language, by extending existing standard languages, or by putting additional metadata into programming code. For the first approach, we developed an XML dialect called CoSMoS/XML. Figure 9 is an example of CoSMoS/XML, describing the direction generator component presented in Figure 7. CoSMoS/XML can be used by itself as a metadata of a component, or can be embedded into another XML-based metadata, such as WSDL or DeploymentDescriptor in EJB. For the second approach, we demonstrated that CoSMoS can be described by using WSDL and RDFS with small extensions. Figure 10 shows the description of the direction generator component in Figure 7 in WSDL and RDFS. For the last approach, we designed CoSMoS/Javadoc, a set of original Javadoc tags that can be used to describe CoSMoS in the comments of a Java source code. We are also planning to support CoSMoS using metadata in Java 1.5 and attributes in .Net Framework. The detail specifications of those languages are available at [7].

Our Java implementation of CoSMoS consists of a package of core classes, a set of parsers, a set of generators, and a set of GUI components. The core classes implement the CoSMoS classes defined in Figure 3, and allow a CoSMoS model to be created, stored and manipulated as Java objects. The parsers parse a file described in either CoSMoS/XML, WSDL, RDFS or Java source

```
<cosmos targetComponent="DirectionGenerator" ... >
<structure name="Address"> ... </structure>
...
<noun name="cosmos://english#direction">
<link represents="cosmos://english#from"
target="cosmos://english#origin"/>
<link represents="cosmos://english#to"
target="cosmos://english#destination"/>
</noun>
<component name="DirectionGenerator">
<operation name="generate"
performs="cosmos://english#generate">
<input>
<Address name="originComp"
represents="cosmos://english#origin"/>
<Address name="destinationComp"
represents="cosmos://english#destination"/>
</input>
<output>
...
</output>
</operation>
</component>
</cosmos>
```

Figure 9. Example of CoSMoS/XML

```
<wsdl:definitions ...>
<wsdl:types>
...
<complexType name="Address">
...
</wsdl:types>
<rdf:RDF>
...
<rdfs:Class rdf:ID="&en;direction">
<rdfs:subClassOf rdf:resource="&cosmos;Noun"/>
</rdfs:Class>
<rdf:Statement>
<rdf:subject rdf:resource="&en;direction"/>
<rdf:predicate rdf:resource="&en;from"/>
<rdf:object rdf:resource="&en;origin"/>
</rdf:Statement>
...
</rdf:RDF>
<wsdl:message name="generateRequest">
<wsdl:part name="originComp" type="impl:Address"
cosmos:represents="&en;origin"/>
<wsdl:part name="destinationComp" type="impl:Address"
cosmos:represents="&en;destination"/>
</wsdl:message>
...
<wsdl:portType name="DirectionGenerator">
<wsdl:operation name="generate" ...>
<wsdl:input name="generateRequest"
message="impl:generateRequest"/>
<wsdl:output name="generateResponse"
message="impl:generateResponse"/>
</wsdl:operation>
</wsdl:portType>
...
</wsdl:definitions>
```

Figure 10. Example of CoSMoS in RDFS and WSDL

code into a set of Java objects. The generators generate either Java source code templates or a CoSMoS/XML file from a given set of Java objects. The GUI components generate the graphical representation of a given CoSMoS model (e.g. Figure 7, Figure 8). Our implementation of CoSMoS is available for download at [7].

3. COMPONENT RUNTIME ENVIRONMENT (CoRE)

Since CoSMoS is a new component model, existing component technologies do not directly support CoSMoS. In order to allow CoSMoS to be used in the existing component technologies, we have defined a unified interface called Component Runtime Environment (CoRE).

CoRE provides a unified interface to discover and access components implemented in various component technologies. Using CoSMoS parsers, CoRE converts the metadata of the discovered components into CoSMoS. This allows the upper layer (i.e. a service composition and execution layer) to analyze and manipulate the components using CoSMoS. CoRE has a pluggable architecture so that a new component technology can be easily implemented onto CoRE. Figure 11 shows the architecture of CoRE.

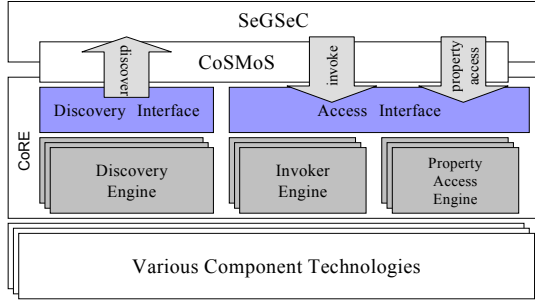


Figure 11. CoRE architecture

CoRE provides two sets of interfaces: Discovery interface to discover components, and Access interface to access components. The Discovery interface provides the functionality to discover a component by its keyword, by its URI, by its property, or by the input or output of its operation. The Access interface provides the functionality to invoke an operation of a component, and the functionality to retrieve a property of a component.

When initializing CoRE, one or more Engines can be plugged in under the Discovery or Access interfaces. A DiscoveryEngine, an Engine for the Discovery interface, either bridges CoRE onto a repository service, such as UDDI, or implements a discovery mechanism, such as Gnutella protocol [8]. A DiscoveryEngine incorporates a CoSMoS parser to convert the metadata (e.g. WSDL) of the discovered component into CoSMoS. An InvokerEngine, an Engine for the Access interface, implements local or remote invocation of operations (i.e. methods) using a specific technology, such as CORBA, SOAP, or RMI. A PropertyAccessEngine, another Engine for the Access interface, provides a mechanism to retrieve a property of a component through a specific technology, such as File search, WWW access, or database query.

We have been implementing CoRE in Java. Currently, we have completed the implementation of CoRE's Engines for a local environment, i.e., the Engines to discover and access the components stored on a local host. The Engines using Web Service technologies are also under development.

4. SEMANTIC GRAPH BASED SERVICE COMPOSITION (SeGSeC)

With the semantics support of CoSMoS, we have developed a semantics-based service composition mechanism named Semantic Graph based Service Composition (SeGSeC). SeGSeC allows a user to request a service using a natural language sentence, and generates the execution path of the requested service. An execution path describes the structure of a service, i.e., it specifies which operations or properties in which components should be accessed in what order. SeGSeC also performs semantic matching to confirm that the semantics of the generated path matches the semantics of the user request. This section first presents the architecture of SeGSeC, which is followed by the detailed algorithm description of SeGSeC.

4.1 SeGSeC Architecture

SeGSeC consists of the following processes: user request analysis, execution path generation, semantic matching, and service execution. Those processes are implemented as a collection of

agents, namely, RequestAnalyzer, ServiceComposer, Reasoner, and ServicePerformer (Figure 12).

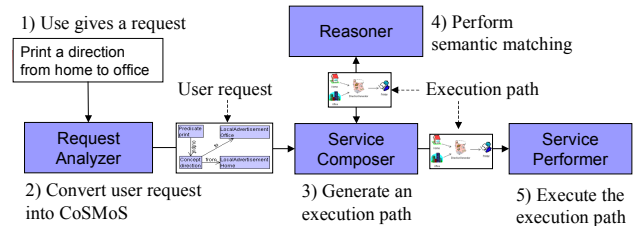


Figure 12. Agents in SeGSeC

RequestAnalyzer parses a user request given as a string text into a CoSMoS semantic graph representation. After parsing a user request, RequestAnalyzer gives the request to ServiceComposer. Upon receiving a request from RequestAnalyzer, ServiceComposer first generates an execution path by connecting operations of components. Then, ServiceComposer collaborates with Reasoner, an agent implementing a reasoning functionality, to perform semantic matching. After completing the semantic matching, Service-Composer asks a user whether to execute the path or not. If the user agrees to execute the path, ServiceComposer gives the path to Service-Performer. On receiving an execution path, ServicePerformer executes the path by accessing the components (i.e. invoking operations and retrieving properties) through CoRE Access interface.

4.2 Service Composition Algorithm

This section describes the algorithm of SeGSeC in detail. As an example, we illustrate how SeGSeC composes the direction printing service (described in Section 1) from the user request “print direction from home to restaurant” using four components: Home (Figure 13), Restaurant (Figure 14), Direction generator (Figure 7), and Printer (Figure 8).

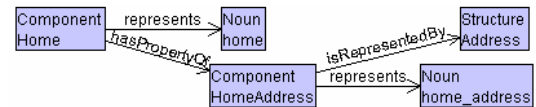


Figure 13. Home component

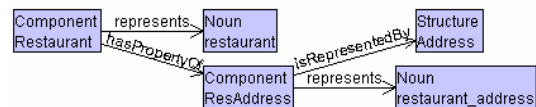


Figure 14. Restaurant component

4.2.1 Step 1: User Request Analysis

Given a string value (e.g. “print direction from home to restaurant”) from a user, RequestAnalyzer first breaks it into individual words through syntactic analysis. Next, for each word, RequestAnalyzer discovers corresponding CoSMoS nodes and links (i.e. *concepts*) through CoRE Discovery interface. Then, RequestAnalyzer creates a single CoSMoS semantic graph using the discovered nodes and links (e.g. Figure 15).

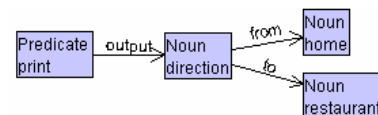


Figure 15. User Request in CoSMoS

Current SeGSeC assumes that a user request contains a single predicate (e.g. “print”), a noun which is the object of the predicate (“direction”), and auxiliary nouns with prepositions (“from home”, “to restaurant”). The following is the BNF syntax definition of the user request assumed in SeGSeC.

UserRequest ::= Predicate Object

*Object ::= Noun (Concept Noun)**

After parsing a user request into a CoSMoS semantic graph, RequestAnalyzer gives the graph to ServiceComposer.

4.2.2 Step 2: Operation Discovery

Upon receiving a user request (as a semantic graph) from RequestAnalyzer, ServiceComposer first searches for an operation that *performs* the predicate in the request through CoRE Discovery interface. In the example scenario, ServiceComposer discovers the ‘print’ operation of the Printer component.

If no operation is found for the given predicate, or ServiceComposer fails to compose a service with the discovered operation, ServiceComposer discovers more operations by finding synonyms of the predicate or by generalizing the predicate using ontologies.

4.2.3 Step 3: Input Complement

After discovering the target operation, ServiceComposer next performs a process called Input Complement. Input Complement is the process which, given an operation, creates an execution path by finding other components that can be supplied as the inputs of the operation. The following kinds of components are considered as the inputs of the operation: (1) components specified in the user request, (2) outputs of other components’ operations, and (3) properties of other components.

In the Input Complement process, for each input i_x in the inputs ($i_1 \dots i_n$) of the specified operation, ServiceComposer discovers the components whose *data types* and *concepts* are compatible with those of the input i_x , and stores them in a list L_x . Data type compatibility and concept compatibility are defined as in Figure 16 and Figure 17.

For data types DT1 and DT2, DT1.compatibleWith(DT2) iff
 DT1 == DT2, or
 DT1.equivalentTo(DT2), or
 DT1.subTypeOf(DT2), or
 \exists DT3: DT1.compatibleWith(DT3) &
 DT3.compatibleWith(DT2), or
 DT1, DT2 == Structure &
 $\forall x \in DT1, \exists y \in DT2: x.compatibleWith(y)$

Figure 16. Data type compatibility

For concepts S1 and S2, S1.compatibleWith(S2) iff
 S1 == S2, or
 S1.equivalentTo(S2) or
 S1.subConceptOf(S2), or
 \exists S3: S1.compatibleWith(S3) & S3.compatibleWith(S2), or
 S1 == Wildcard, or
 S2 == Wildcard

Figure 17. Concept compatibility

After creating the list of ($L_1 \dots L_n$) for the inputs ($i_1 \dots i_n$), ServiceComposer next computes all possible combinations ($LC_1 \dots LC_n$) of the components such that (1) each combination LC_x contains exactly one component from each of the lists ($L_1 \dots L_n$), and (2) the members of LC_x are different from the members of LC_y if $x \neq y$. After computing all the possible combinations, ServiceComposer sorts the combinations based on their similarity values (i.e. the number of common node in the combination and the user request) and compatibility values (the number of times the rules in Figure 16 and Figure 17 were applied when checking the compatibilities), and chooses the top one. Then, ServiceComposer creates an execution path by declaring that the components in the combination are used for the inputs of the specified operation. ServiceComposer iterates the Input Complement process and extends the execution path until all the operations in the execution path become executable, i.e., all the inputs of the operations are either specified in the user request or provided as the properties of the components. Figure 18 shows the pseudo-code of the input complement process.

In the example scenario, ServiceComposer creates either of the two execution paths in Figure 19 through the Input Complement process.

```

inputComplement(Operaiton op, ExecutionPath EP){
  create a list L of empty lists
  # L[1] ... L[N]: each L[x] is an empty list
  # N = op.inputs.size

  for  $i_x = op.inputs[x]$ 
    if  $i_x$  is a Choice
      add all its Literal alternatives to L[x]
    otherwise
      from the user request, outputs, or properties,
      find a component c which is compatible with  $i_x$ 
      #through CoRE Discovery interface
      add c into L[x]

  create a list LC of possible combinations of components
  in L[1]...L[N]
  # LC[1] ... LC[M]: each LC[x] is a list
  # LC[x] contains one component from each of L[1] ... L[n]
  # LC[x]!=LC[y] if  $x \neq y$ 

  sort LC[1]...LC[m] in LC based on
  similarity to the user request, and
  # number of common nodes in LC[x] and the user request
  compatibility value
  # calculated when checking compatibility

  for each LC[x]
    add LC[x] in the front of the execution path EP
    if LC[x] contains an operation's output
      perform input complement for the operation
    otherwise
      perform semantic matching on EP
}

```

Figure 18. Input Complement pseudo code

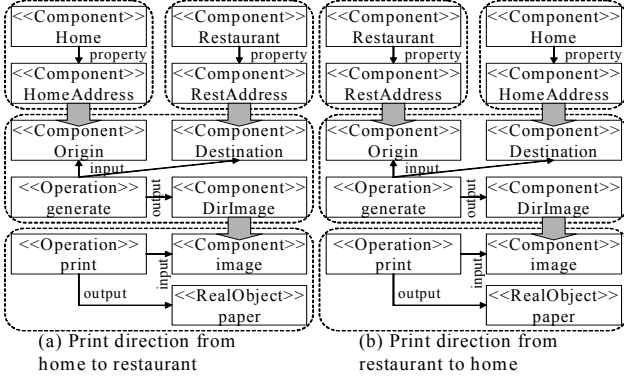


Figure 19. Generated execution paths

4.2.4 Step 4: Semantic Matching

After creating a possible execution path through the Input Complement process, ServiceComposer performs semantic matching to check whether the semantics of the discovered path matches the user request or not.

In order to perform semantic matching, ServiceComposer first converts the execution path into a semantic graph by adding *argument* links between the components in the path. As described earlier, an execution path defines which component (say, *argument component*) is used as an input (say, *input component*) of the operation of another component. To represent such a data flow as a semantic graph, ServiceComposer adds *argument* links between every pair of *argument component* and *input component*. For instance, all the gray thick arrows in Figure 19 will become *argument* links in this process. Then, ServiceComposer gives the execution path (in the form of semantic graph, as in Figure 19) and the user request (which is also a semantic graph, as in Figure 15) to Reasoner to check whether the semantics of the execution path is satisfied by the user request.

Reasoner applies the rules in Table 1 onto the execution path in order to derive the semantics of the path. Those rules are designed such that they derive semantic facts from the given execution path, by adding proper links between *concepts* in the execution path based on the data flows (i.e. *argument* links) and the structure of components (i.e. operations and properties) defined in the execution path. For example, the first rule in Table 1 draws on a fact that an operation generating an output semantically means that the predicate of the operation generates the concept of the output.

In order to derive the semantic facts from the execution path and compare them with the user request, Reasoner interprets the links in the user request as *goal statements* and the links in the execution path as *fact statements*, and performs reasoning (such as forward chaining) to check if the *goals* (i.e. user request) can be derived by applying the rules in Table 1 onto the *facts* (execution path). If the components in the execution path contain some logics, they are also supplied as the rules. If Reasoner could successfully conclude that the *goals* are derived by the *facts* by applying the rules in Table 1, Reasoner notifies ServiceComposer that the given execution path satisfies the user request. In this case, ServiceComposer proceeds to the next step, Service Execution, to execute the path. If Reasoner concludes that the *goals* are not derived, ServiceComposer returns back to the Input Complement process to create another execution path.

Table 1. SeGSeC semantic matching rules

Rule	Meaning
$O.performs(P)$ & $O.output(C)$ & $C.represents(S)$ $\rightarrow P.output(S)$	If an operation O performs a predicate P, and O outputs a component C representing S, then, consider that P outputs S.
$Y.argument(X)$ & $X.represents(S_x)$ & $Y.represents(S_y)$ $\rightarrow S_x.equivalentTo(S_y)$	If Y is the argument of X (i.e., Y is used as the input X), the semantics of X is equivalent to the semantics of Y.
$Y.propertyOf(X)$ & $X.represents(S_x)$ & $Y.represents(S_y)$ $\rightarrow S_x.equivalentTo(S_y)$	If Y is the property of X, then the semantics of X is equivalent to the semantics of Y.
$S_1.equivalentTo(S_2)$ & $X.LINK(S_1)$ $\rightarrow X.LINK(S_2)$	If a concept S_1 is equivalent to a concept S_2 , apply all the links towards S_1 onto S_2 .

In the example scenario, after applying the rules in Table 1, Reasoner concludes that only the execution path in Figure 19 (a), which prints out the direction from home to the restaurant, satisfies the user request in Figure 15.

4.2.5 Step 5: Service Execution

After ServiceComposer concludes that the semantics of the execution path matches the user request, ServiceComposer passes the execution path to ServicePerformer. ServicePerformer then asks the user whether to execute the path or not. If the user agrees to execute, ServicePerformer executes the given execution path by accessing the components, i.e., by invoking operations and retrieving properties of components in the specified order, through CoRE Access interface. If the user disagrees with the path, ServicePerformer asks ServiceComposer to continue the Input Complement process to search for alternative paths.

4.3 Comparison to existing work

This section describes the features of SeGSeC, compared with other existing service composition systems.

4.3.1 Creation of execution path

eFlow [4], Aurora [9], STONE [11], ICARIS [10], SELF-SERV [15], and Composer [16] are the dynamic composition systems which compose an application from a given service template by selecting components to fill in the template at runtime. Similarly as static composition systems, those template-based systems support applications involving complex interaction patterns such as conditional branch or iteration. However, they are unable to compose a proper service unless there already exists a template that implements the requested service. For example, a user cannot use a direction printing service if no such template is available. On the other hand, SeGSeC does not require any template to compose a service. Instead, SeGSeC generates an execution path from a given user request and the metadata of available components. Compared to SeGSeC, the template-based systems are more suitable for B2B applications where interactions are fairly complex and static, but are less suitable for ubiquitous end-user applications where available components and expected users may vary.

4.3.2 Support of semantics

Ninja [6] creates an execution path from a user request by performing interface matching. Although the approach is similar to SeGSeC, Ninja only compares and matches the data types of interfaces, and does not consider semantics of the interface information at all. SeGSeC takes the semantics of components and of the composed service into account, by using the semantics representation of CoSMoS and by performing semantic matching.

SWORD [13] models a service as a set of inputs, outputs, precondition and postcondition, and applies a planning technique to derive execution path(s) that satisfies a given user request. Although SWORD supports semantics of service as a pair of pre/postconditions, SWORD requires services (pre/post conditions) to be annotated by logic formulas (first order logics). This requirement is error-prone because the interface information (i.e. input/output of a service) and its semantics (i.e. pre/post conditions) have to be defined independently using different notations, which may cause inconsistency between them. CoSMoS, on the other hand, integrates interface information and its semantics using semantic graph representation. Also, SeGSeC automatically interprets CoSMoS semantic graph representation as a set of logic sentences. Therefore, the risk of inconsistency among interface information and its semantics is minimized.

4.3.3 Usability

Ninja [6], SWORD [13], and SHOP2 [18] assume that a user request is given as a pair of inputs and outputs or as a logic formula, without really mentioning how the request can be generated. SeGSeC assumes that a user request is given as a natural language sentence. As there exist many existing researches for language analysis, our assumption on the user request generation is more feasible than those of other existing systems.

Composer [16], a template-based dynamic service composition system, asks a user to choose which components to use during its service composition process. This gives more controllability for a user on the composition process. However, understanding the structure of the requesting service may be too demanding for novice users. SeGSeC composes the requested service autonomously without involving any user interaction, but asks user permission to execute a composed service, and also shows alternatives if the user disagrees with the composed one. This approach achieves the same selectivity as Composer.

4.4 Implementation

In order to empirically test and evaluate SeGSeC, we implemented SeGSeC in Java using J2SE 1.4.2. Figure 20 is the screenshot of the demo system based on our CoSMoS, CoRE and SeGSeC implementation. Using the demo system, a user can deploy components on a local host, compose an application by typing a request as a sentence, and execute the composed service. So far, we have implemented 10 components and demonstrated that several services, including the direction printing service and the others shown in Figure 21, can be composed on the demo system.

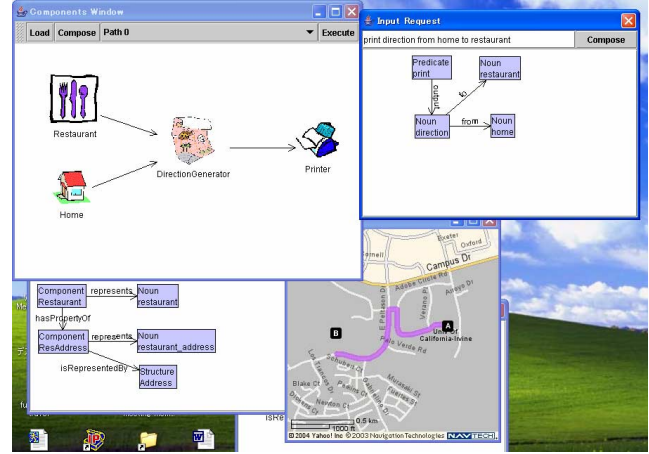


Figure 20. Screenshot of our demo system

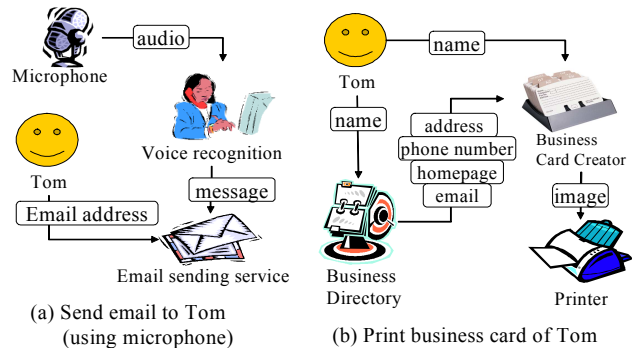


Figure 21. Other services implemented on SeGSeC

4.5 Measurement

In order to evaluate the performance of our SeGSeC implementation, we conducted preliminary performance measurements using three sample services (i.e. the print direction service and those in Figure 21). The measurements were performed using J2SE ver.1.4.2, running on a Pentium 4 2.24GHz machine with 1GB memory. Table 2 is the measurement results, showing that the execution times (in milliseconds) of the Input Complement and Semantic Matching processes for each service.

The results of our measurement illustrate that our current SeGSeC implementation is able to compose the requested services in a reasonable amount of time, i.e., less than a second. It also indicates that most of the composition time is consumed by the semantic matching process performed by Reasoner. This implies that the performance of our current implementation could be improved further by optimizing the reasoning engine.

Table 2. Measurement results

User request	Input Complement [ms]	Semantic Matching [ms]
Print direction from home to restaurant	5	77
Send email to Tom	8	79
Print business card of Tom	10	480

5. CONCLUSION AND FUTURE WORK

This paper presents our semantics-based dynamic service composition system that consists of CoSMoS, CoRE and SeGSeC. CoSMoS integrates the semantic information of a component and the functional information of a component into a single semantic graph representation. CoRE converts different component implementations onto CoSMoS representation. Using the semantic support of CoSMoS, SeGSeC generates the execution path of the requested service, and checks the semantics of the path against the user request. We have implemented the above techniques and demonstrated that our system supports semantics-based dynamic service composition.

As for future work, we are considering extending CoSMoS to support user interface and streaming data. Further analysis of the design of CoSMoS, such as how much complex data structure, semantics or logics CoSMoS can support, will also be needed. Also, we are designing a distributed version of SeGSeC, in which multiple agents collaboratively compose a service in a distributed manner. In addition, we are considering extending SeGSeC by using heuristic information, such as performance metrics or user evaluation, to make the algorithm more efficient and adaptable.

6. ACKNOWLEDGEMENTS

This work is supported by the NSF through grants ANI-0083074 and ANI-9903427, 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, Novell, Nippon Telegraph and Telephone (NTT), NTT Docomo, Fujitsu, NS Solutions Corporation, DENSO IT Laboratory.

7. References

- [1] Bouguettaya, A., *Introduction to the Special Issue on Ontologies and Databases*, Distributed and Parallel Databases Journal. Kluwer Publishers, 1999. 7(1).
- [2] *Business Process Execution Language for Web Services Version 1.1*, [online] <http://www-106.ibm.com/developerworks/library/ws-bpel/>
- [3] Carman, M., Serafini, L., and Traverso P. *Web Service Composition as Planning*, ICAPS'03 Workshop on Planning for Web Services, Trento, Italy, June 2003
- [4] Casati, F., Ilnicki, S., Jin, L.-J., Krishnamoorthy, V., and Shan, M.-C. *Adaptive and dynamic service composition in eFlow*, In Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE), Stockholm, Sweden, 2000.
- [5] 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.
- [6] Chandrasekaran, S., Madden, S., and Ionescu, M., *Ninja Paths: An Architecture for Composing Services over Wide Area Networks*, CS262 class project writeup, UC Berkeley, 2000.
- [7] Fujii, K. *Dynamic Service Composition*, [online] <http://netresearch.ics.uci.edu/kfujii/dsc/>
- [8] *The Gnutella Protocol Specification v0.4*, [online] Fhttp://www9.limewire.com/developer/gnutella_protocol_0.4.pdf
- [9] Marazakis, M., Papadakis, D., and Nikolaou, C. *The Aurora Architecture for Developing Network-Centric Applications by Dynamic Composition of Services*, Technical Report TR 213, FORTH/ICS, 1997.
- [10] Mennie, D., and Pagurek, B. *An Architecture to Support Dynamic Composition of Service Components*, Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000), Sophia Antipolis, France, 2000.
- [11] Minami, M., Morikawa, H., and Aoyama, T. *The Design and Evaluation of an Interface-based Naming System for Supporting Service Synthesis in Ubiquitous Computing Environment*, Trans. of The Institute of Electronics, Information and Communication Engineers, vol.J86-B, no.5, pp.777-789, May 2003.
- [12] *OWL-S 1.0 Release*, [online] <http://www.daml.org/services/owl-s/1.0/>
- [13] Ponnekanti, S. R. and Fox, A. *SWORD: A Developer Toolkit for Web Service Composition*, to appear in The Eleventh World Wide Web Conference (Web Engineering Track), Honolulu, Hawaii, May 7-11, 2002.
- [14] Raman, B. and Katz, R. H. *An architecture for highly available wide-area service composition*, Computer Communications Journal, special issue on "Recent Advances in Communication Networking", May 2003.
- [15] Sheng, Q. Z., Benatallah, B., Dumas, M., and Mak, E. *SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment*, In Proceedings of the 28th Very Large DataBase Conference (VLDB'2002), Hong Kong, China, August 2002.
- [16] Sirin, E., Hendler, J., Parsia, B. *Semi-automatic Composition of Web Services using Semantic Descriptions*, workshop on Web Services: Modeling, Architecture and Infrastructure, in conjunction with ICEIS2003, 2002.
- [17] *Web Service Choreography Interface (WSCI) 1.0*, [online] <http://www.w3.org/TR/wsci/>
- [18] Wu, D., Parsia, B., Sirin, E., Hendler, J., and Nau, D. *Automating DAML-S web services composition using SHOP2*, In Proceedings of 2nd International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, October 2003.