Semantics-Based Dynamic Service Composition

Keita Fujii, *Student Member, IEEE,* and Tatsuya Suda, *Fellow, IEEE*

***Abstract*—Complex services may be dynamically composed through combining distributed components on demand (i.e., when requested by a user) in order to provide new services without preinstallation. Several systems have been proposed to dynamically compose services. However, they require users to request services in a manner that is not intuitive to the users. In order to allow a user to request a service in an intuitive form (e.g., using a natural language), this paper proposes a semantics-based service composition architecture. The proposed architecture obtains the semantics of the service requested in an intuitive form, and dynamically composes the requested service based on the semantics of the service. To compose a service based on its semantics, the proposed architecture supports semantic representation of components [through a component model named Component Service Model with Semantics (CoSMoS)], discovers components required to compose a service [through a middleware named Component Runtime Environment (CoRE)], and composes the requested service based on its semantics and the semantics of the discovered components [through a service composition mechanism named Semantic Graph-Based Service Composition (SeGSeC)]. This paper presents the design, implementation and empirical evaluation of the proposed architecture.**

***Index Terms*—Component model, dynamic service composition, semantics, semantics-based service composition, service oriented architecture.**

I. Introduction

A. Dynamic Service Composition

The recent development of distributed component technologies such as CORBA and Web Service made it possible to componentize various software programs, devices, and resources and to distribute them over a network. In such an environment where a large number of various components are distributed, it is possible to dynamically compose a service *on demand*, i.e., composing a service upon receiving a request from a user, through discovering, combining and executing necessary components. Composing services on demand (i.e., dynamic service composition) has various advantages. For instance, by dynamically composing services on demand, services do not need to be configured or deployed in advance. In addition, by composing services based on requests from users, it is possible to customize the services to individual user profiles.

Manuscript received October 1, 2004; revised June 24, 2005. This work was supported in part by the National Science Foundation (NSF) under Grant ANI-0083074 and Grant ANI-9903427, in part by the Defense Advanced Research Projects Agency (DARPA) under Grant MDA972-99-1-0007, in part by the Air Force Office of Scientific Research (AFOSR) under Grant MURI F49620-00-1-0330, and in part by grants from the California MICRO and CoRe programs, Hitachi, Hitachi America, Denso IT Laboratory, NTT Docomo, Nippon Telegraph and Telephone (NTT), Novell, Fujitsu, NS Solutions Corporation, and NICT.

The authors are with School of Information and Computer Science, University of California, Irvine, Irvine, CA 92697-3425 USA (e-mail: kfujii@ics.uci.edu; suda@ics.uci.edu).

Digital Object Identifier 10.1109/JSAC.2005.857202

To illustrate the advantage of dynamic service composition, consider the following example scenario. Suppose Tom wants to take his family to a new restaurant he recently found on the web, so he wants to print out a map showing a direction to the restaurant from his house. Assume that: 1) the restaurant's Web server stores the restaurant's information such as its address in a structured document (e.g., in XML); 2) Tom's PC stores his personal information such as his home address in a database; 3) there is a Web Service which, given two addresses, generates a map showing a direction from one address to the other; and 4) Tom has a printer connected to his home network. In order to print out the map showing the direction from his home to the restaurant with the technology currently available, Tom has to manually perform the following steps: a) discover the restaurant's homepage and the Web Service that generates a map; b) obtain the addresses of the restaurant and his home; c) invoke the Web Service using the addresses obtained; and d) print out the map generated by the Web Service. However, if Tom is not an experienced PC user, it may be difficult for him to perform these steps. For instance, he may not know how to input the restaurant's address to the Web Service that generates a map. Dynamic service composition, on the other hand, will automatically compose the direction printing service, upon Tom's request, by discovering the four necessary components, identifying the steps a)–d) and executing them on Tom's behalf. Since the direction printing service is composed on demand, it is not required to be configured or deployed in advance. Also, if Tom's daughter, Alice, requests for a direction to the restaurant, and if she carries a PDA with her, the service may be customized for her such that it shows the map on her PDA's display instead of printing it out.

B. Existing Dynamic Service Composition Systems

Several dynamic service composition systems have been proposed and implemented. However, most of the existing dynamic service composition systems require a user to request a service in a manner that may not be trivial and intuitive to the user.

Some of the existing systems (e.g., eFlow [1], ICARIS [2], STONE [3], SELF-SERV [4], and those described in [5]–[8]) require a user to request a service by choosing or creating a service template that describes the structure of the service in a flowchart-like diagram. They compose the requested service through discovering the components necessary to convert (or instantiate) the template into an executable workflow. However, choosing or creating service templates may not be trivial nor intuitive as it requires knowledge on the format of a template.

Some existing systems (e.g., BU-Grid [9] and NinjaWorkflow [10]) require a user to specify the inputs/outputs of the service he/she requests. They compose the requested service through connecting the inputs and outputs of the components such that the execution of the components accepts the user-specified inputs and generates the user-specified outputs. This approach, however, does not apply to services that do not generate any data as output. For instance, a user may have difficulty in specifying an output for a book purchase service since it may not generate any data as output.

Some existing systems (e.g., SWORD [11], SHOP2 [12], [13], and those described in [14]–[17]) require a user to request a service using a logic language. For example, the system described in [15] requires a user to choose a meta-program described in Golog logic programming language. Similarly, SWORD [11] requires a user to request a service by specifying pre/post conditions of the service using first-order logics. These systems compose the service requested in a logic language through a form of planning. They have shown that service composition is efficiently implemented by applying planning techniques, and that logic languages (e.g., first-order logic formulas) are useful in composing services that do not generate any data as output (e.g., a book purchase service). However, understanding logic programming languages may not be trivial nor intuitive for nonexperts.

C. Semantics-Based Dynamic Service Composition Architecture

In order to allow users to request services in an intuitive manner, this paper proposes a semantics-based dynamic service composition architecture [18]. The proposed architecture assumes that a user requests a service in an intuitive form (e.g., using a natural language) and that the semantics of the requested service (expressed by a user in an intuitive form) is converted into a machine-understandable format (e.g., a semantic graph) through existing technologies.[1] Based on the semantics of the requested service described in a machine-understandable format, the proposed architecture composes a service.

The proposed architecture composes a service through discovering the components necessary to compose the requested service based on the semantics of the requested service, creating a workflow of the requested service using the discovered components, and executing the workflow. In order to support discovering components based on the semantics of the requested service, the proposed architecture contains a semantics-aware component model named Component Service Model with Semantic (CoSMoS) and a middleware named Component Runtime Environment (CoRE). CoSMoS models the semantics of the components using semantic graph representation. CoRE provides the functionality to discover and execute components that are modeled in CoSMoS. In order to support creating and executing a workflow of the requested service, the proposed architecture also contains a semantics-based service composition mechanism named Semantic Graph-Based Service Composition (SeGSeC). SeGSeC first instructs CoRE to discover components based on the semantics of the requested service. Next, SeGSeC creates a workflow using the discovered components such that the workflow satisfies the semantics of the requested service. Then, SeGSeC instructs CoRE to execute the workflow.

[1]For instance, a natural language sentence may be converted into a semantic graph through existing natural language analysis techniques (e.g., [28]).

The remaining sections of this paper are organized as follows. Section II describes a detailed design of CoSMoS. Section III describes the architecture of CoRE. Section IV describes the architecture and empirical evaluation of SeGSeC. Section V concludes the paper.

II. Component Service Model With Semantics (CoSMoS)

This section describes a semantics-aware component model, CoSMoS.

A. CoSMoS Overview

Many existing component models (e.g., WSDL, JavaBeans/EJB, COM, etc.) define a component by specifying the operations that the component performs and the properties of the component. An operation is modeled as a pair of inputs to and outputs from the component, and each input, output, and property is modeled as a pair of a name and a data type. For example, the existing component models define a component that generates a joint photographic experts group (JPEG) image of a map showing a direction from one address to another by using a “generateDirection” operation with two inputs, “from: Address” (i.e., an input named “from” with the data type “Address”) and “to: Address,” and one output, “direction: JPEG.” Although the names of operations, inputs, outputs and properties may imply their semantics (e.g., the name of the input “from” may imply that it is an origin), the existing component models do not explicitly represent the semantics information.

Similar to the existing component models, CoSMoS in the proposed architecture defines a component by specifying its operations and properties. In addition, in order to model the semantics of a component, CoSMoS introduces *concepts*, entities representing abstract ideas (e.g., “direction” and “restaurant”) and actions (e.g., “print” and “generate”), and annotates the semantics of the operations, inputs, outputs and properties of the component using *concepts*. For instance, with CoSMoS, the semantics of the direction generator component (i.e., a component to generate a map showing a direction from one address to another) may be modeled by specifying the *concepts* “origin,” “destination” and “direction” as the semantics of the two address inputs and the JPEG output, respectively. CoSMoS also uses a *concept* to specify the relationship between two *concepts*. For instance, a “from” *concept* may be used to specify the relationship between the “origin” and “direction” *concepts*, that is, to specify that a “direction” is “from” an “origin.” Although this approach of modeling the semantics of a component using *concepts* is similar to the approach proposed by the Semantic Web Service standards such as OWL-S [19] and WSMO [20], CoSMoS is unique in that it annotates the *concept* of an operation of a component in addition to the concept of an input, output, and property of a component, and that it distinguishes the *concept* and the data type of an input, output and property.

In order to define a component using operations, properties and *concepts* in a machine-understandable format, CoSMoS describes a component as a semantic graph that consists of nodes and labeled links. Nodes in the semantic graph represent operations, inputs, outputs and properties of a component, as well

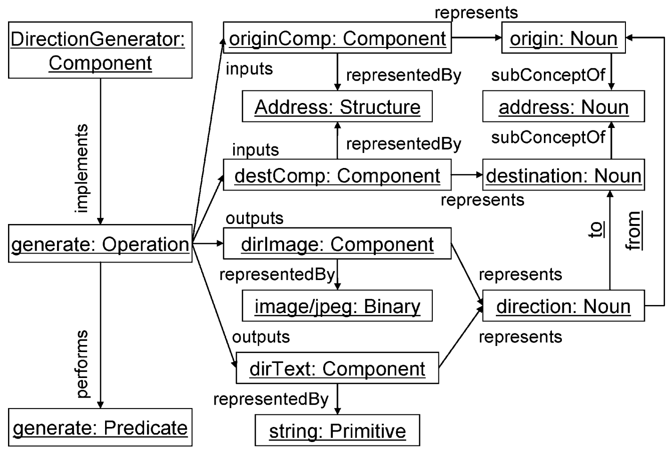


Fig. 1. Direction generator component in CoSMoS.

as their data types and *concepts*. Labeled links in the semantic graph represent the relationships among the nodes. For example, Fig. 1 illustrates the semantic graph representation of the direction generator component in CoSMoS.

B. CoSMoS Design

In order to describe a component as a semantic graph, CoSMoS defines the 18 classes illustrated in the UML class diagram in Fig. 2. These classes are used to specify what each node and labeled link in a semantic graph represents. For example, CoSMoS uses the *Operation* class in Fig. 2 to specify that the "generate: Operation" node (i.e., an instance of the *Operation* class) in Fig. 1 represents an operation of a component. There are four domains for the classes defined in CoSMoS based on their roles, namely, component domain, data type domain, semantics domain, and logic domain. The following subsections explain these classes in each domain in detail.

1) Component Domain: CoSMoS assumes that a component may implement one or more operations and that each operation accepts another component(s) as its input(s) and generates another component(s) as its output(s). CoSMoS specifies an operation of a component as an *operation* node (an instance of the *Operation* class) with an "implements" link connected to a *component* node (an instance of the *Component* class). The inputs and outputs of an operation are specified as *component* nodes (representing the input or the output) with "inputs" and "outputs" links connected to an *operation* node. For example, in Fig. 1, "DirectionGenerator: Component" implements "generate: Operation," which accepts two inputs, "originComp: Component" and "destComp: Component," and generates two outputs, "dirImage: Component" and "dirText: Component." An operation may throw an exception, which is specified as an *exception* node (an instance of the *Exception* class) with a "throws" link connected to an operation node.

In order to support components (e.g., a microphone and a printer) that accept or generate real objects (e.g., sound and a paper) instead of digital data, CoSMoS defines the *RealObject* class (a subclass of the *Component* class) and models the real objects as *realObject* nodes (instances of the *RealObject* class). Also, in order to support components that ask users for several options (e.g., a printer that asks a user to choose different paper sizes), CoSMoS defines two subclasses of the *Component* class, namely, *Literal* and *Choice*, and uses a *choice* node (an instance of the *Choice* class) grouping several *literal* nodes (instances of the *Literal* class) to specify possible options. For example, in CoSMoS, a printer component that asks a user to select the paper size between Letter size and Legal size may have a *choice* node consisting of the "Letter" and "Legal" *literal* nodes as an input of its operation.

CoSMoS also assumes that a component may have one or more properties and that each property is retrieved as another component. CoSMoS specifies a property of a component as a *component* node (representing the property) with a "hasPropertyOf" link connected to another *component* node (representing the owner of the property). For example, Fig. 3 illustrates a restaurant component with its address (e.g., "RestAddress: Component") as a property.

2) Data Type Domain: CoSMoS specifies the data type of an input/output of an operation and the data type of a property as a *dataType* node (an instance of the *DataType* class or its subclass) with a "representedBy" link connected to a *component* node (representing the input/output or the property). For instance, in Fig. 3, the property "RestAddress: Component" is represented by the structured data "Address: Structure."

CoSMoS predefines several *primitive* nodes (instances of the *Primitive* class, which is a subclass of the *DataType* class), namely, "int: Primitive," "string: Primitive," "float: Primitive," and "Boolean: Primitive," in order to support common primitive data types, such as integer, string, float, and Boolean. CoSMoS also predefines several subclasses (i.e., *Array*, *Structure*, *Binary*, and *File*) of the *DataType* class in order to support common data structures, such as array, structured data, binary data, and file types. CoSMoS also supports implementation-specific data types (such as Java collection library and XML schema datatypes) by defining *dataType* nodes connected by "compatibleWith" and "subTypeOf" links. For instance, the implementation-specific data types "java.util.Map" and "java.util.HashMap" in Java may be defined as two *dataType* nodes, "java.util.Map: DataType" and "java.util.HashMap: DataType," connected by a "subTypeOf" link. Two data types are compatible if they are linked by "equivalentTo" or "subTypeOf" links. Two data types are also compatible if they are both structured data types (i.e. instances of the *Structure* class) and their member data types are also compatible (Fig. 4).

3) Semantics Domain: CoSMoS defines the *Concept* class and its subclasses, *Noun* and *Predicate*, to support *concepts* (i.e., entities representing abstract ideas and actions).

CoSMoS models the semantics of an input/output of an operation and the semantics of a property as a *noun* node (an instance of the *Noun* class) with a "represents" link connected to a *component* node (representing the input/output or the property). For instance, in Fig. 3, the property "RestAddress: Component" represents the concept "address: Noun." Unlike OWL-S and WSMO that define a *concept* by specifying its data type, CoSMoS defines a *concept* and a data type separately in order

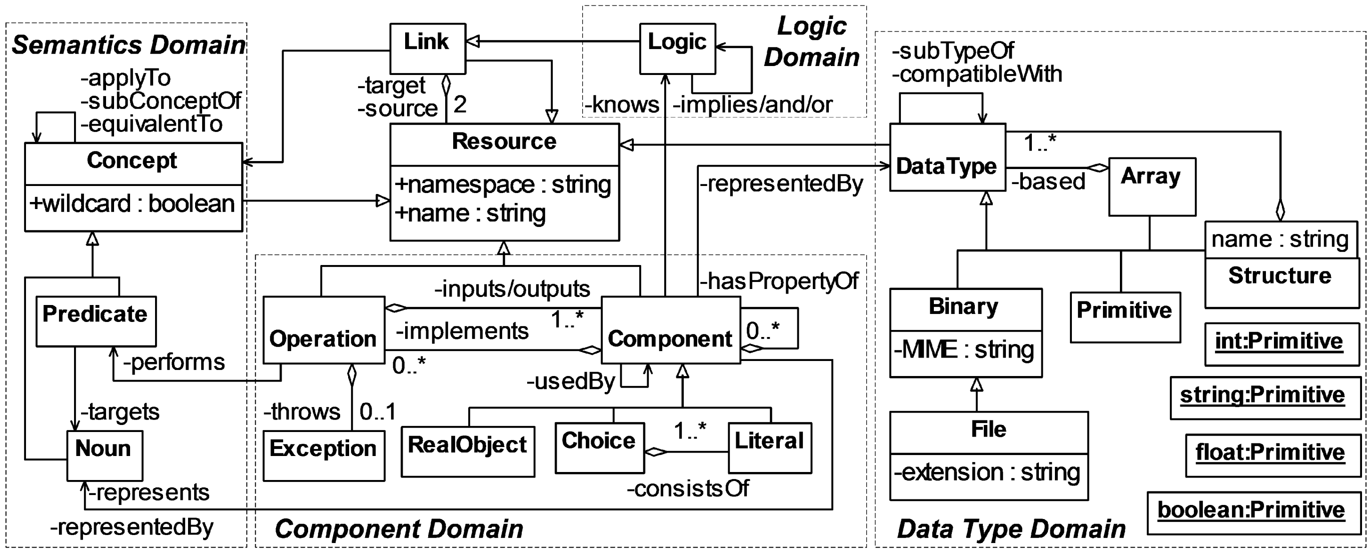


Fig. 2. CoSMoS class diagram.

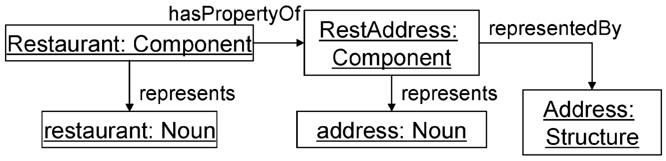


Fig. 3. Restaurant component in CoSMoS.

For data types D1 and D2, compatibleWith(D1, D2) iff
equivalentTo(D1, D2), or subTypeOf(D1, D2), or
$\exists$D3:compatibleWith(D1, D3) & compatibleWith(D3, D2), or
D1, D2== Structure & $\forall x \in$D1, $\exists y \in$D2: compatibleWith(x, y)

Fig. 4. Data type compatibility.

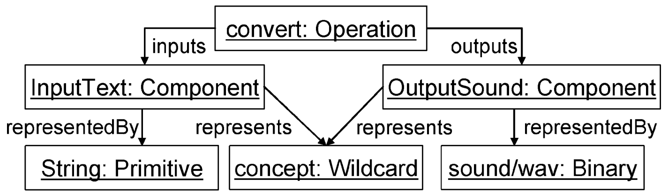


Fig. 5. Example of a wildcard concept.

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

Fig. 6. Concept compatibility.

to support the inputs, outputs and properties that are represented by different data types but represent the same *concept*.

CoSMoS models the semantics of an operation as a *predicate* node (an instance of the *Predicate* class) with a "performs" link connected to an *operation* node. For example, in Fig. 1, "generate: Operation" performs "generate: Predicate." Despite its simplicity, the predicate-based semantics representation of CoSMoS is suitable for the proposed architecture because predicates may be easily obtained from an intuitive form (e.g., a natural language). In OWL-S, WSMO and other recently proposed Semantic Web Service models such as WSDF [21], METEOR-S [22], and SESMA [23], the semantics of an operation (e.g., a book purchase operation provided by an online book store) is modeled by its effect/postcondition (e.g., the ownership of the book is transferred to the user) defined as a logic formula [e.g., "own(user, book)"]. Although logic formulas may support the formal definition of the semantics of an operation, it may not be trivial to obtain a logic formula [e.g., "own(user, book)"] from an intuitive form (e.g., a sentence "purchase a book"). Using predicates provide a valuable alternative to using logic formulas in order to compose services requested in an intuitive form.

A *concept* node (an instance of the *Concept* class or its subclass) may set a Boolean flag named "wildcard" to indicate that it is a *wildcard concept*. To illustrate how a *wildcard concept* is used, consider a text-to-speech converter component. If the input text describes a direction to a restaurant, the output sound data should also read out the direction. If the input text describes a novel, the output sound should also read out the novel. A *wildcard concept* is used to indicate that the input text and the output sound data represent the same but arbitrary *concept*. (Fig. 5).

Two *concept* nodes may be connected with a "subConceptOf" or "equivalentTo" link to model that one *concept* is a subtype of (or equivalent to) another. For instance, a "print" *predicate* node may have a "subConceptOf" link to an "output" *predicate* node. Two *concepts* are compatible if they are linked by "equivalentTo" or "subConceptOf" links. Two *concepts* are also compatible if either of them is a *wildcard concept* (Fig. 6).

CoSMoS also uses a *concept* to label a link in a semantic graph. CoSMoS predefines several *concepts* to label links in a semantic graph (such as "implements" and "inputs" shown as the labels of the associations in Fig. 2). In addition, CoSMoS allows arbitrary (nonpredefined) *concepts* to label links. In order to distinguish the arbitrary link labels from the predefined ones, the arbitrary link labels are depicted with an underbar (e.g., "from" and "to" in Fig. 1) in the figures in this paper.

4) Logic Domain: CoSMoS allows components to provide some logics to specify that some fact (i.e., a condition) *implies*

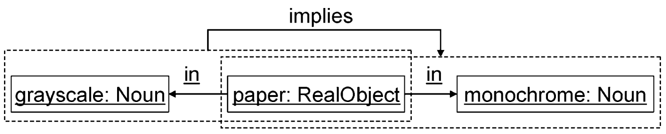


Fig. 7. Example of logic.

another fact (i.e., a consequence). CoSMoS supports logics in order to allow users to request the same service in different ways. For instance, a printer component may provide a logic specifying that “a paper in grayscale” *implies* “a paper in monochrome” so that a user can request monochrome printouts in two ways: “in monochrome” and “in grayscale.” Details of how logics are used in composing services are explained later in Section IV.

CoSMoS defines a logic as an “implies” link, an instance of the *Logic* class (a subclass of the *Link* class), connecting two arbitrary links,[2] one specifying the condition of the logic and another specifying the consequence of the logic. Fig. 7 shows an example logic specifying that “a paper in grayscale” *implies* “a paper in monochrome.” CoSMoS also defines two *concepts*, “and” and “or,” to support logical product and logical sum.

C. CoSMoS Implementation

In order to express a CoSMoS semantic graph in XML, an XML-based description language named CoSMoS/XML has been designed [24]. Also, in order to deploy CoSMoS onto existing Web Service technologies easily, an extension of WSDL named CoSMoS/WSDL and an RDF schema-based description language named CoSMoS/RDFS have been designed [24].

CoSMoS has also been implemented in Java [24]. The current CoSMoS implementation consists of a package of Java classes implementing the CoSMoS classes (in Fig. 2), a set of parsers to parse a file described in CoSMoS/XML, CoSMoS/WSDL, and CoSMoS/RDFS into a set of Java objects modeling a CoSMoS semantic graph, and a set of generators to generate either a Java source code template or a CoSMoS/XML file from the Java objects created by the parsers.

III. Component Runtime Environment (CoRE)

This section describes CoRE, a middleware designed to support CoSMoS on various component technologies. Fig. 8 shows the architecture of CoRE.

CoRE consists of two interfaces, the discovery interface and the access interface, and three groups of modules, namely, DiscoveryEngine, InvokerEngine, and PropertyAccessEngine. The discovery interface provides an interface to discover a component distributed in a network. 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 discovery interface, the DiscoveryEngine discovers a component(s) and provides a CoSMoS semantic graph representation of the component(s) (e.g., by analyzing the component’s metadata described in CoSMoS/XML) to the

[2]In CoSMoS, a link is to connect two nodes (i.e., instances of the resource class), and the Link class is defined as a subclass of the resource class. Therefore, CoSMoS allows a link to connect other two links.

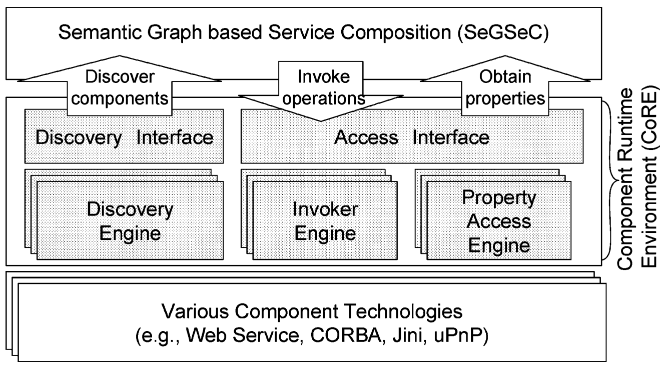

Semantic Graph based Service Composition (SeGSeC)
Discover components
Invoke operations
Obtain properties
Discovery Interface
Access Interface
Discovery Engine
Invoker Engine
Property Access Engine
Component Runtime Environment (CoRE)
Various Component Technologies (e.g., Web Service, CORBA, Jini, uPnP)

Fig. 8. CoRE architecture.

discovery interface. 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 PropertyAccess Engine may be implemented with various component technologies, such as Web Service, CORBA, Jini, or uPnP. For instance, DiscoveryEngine and InvokerEngine may be implemented using Web Service technologies, such as UDDI, WSDL (or CoSMoS/WSDL), and SOAP. Those Engines may also be implemented on a distributed mobile agent architecture, such as the bio-networking architecture [25]–[27]. CoRE automatically selects a proper engine for each component by identifying the technology on which the component is implemented.

CoRE has been implemented in Java [24]. The current CoRE implementation includes a DiscoveryEngine to discover components stored on a local host, an InvokerEngine to invoke components’ methods, and a PropertyAccessEnvine to retrieve components’ properties. The CoRE implementation based on the Web Service technologies is also under development.

IV. Semantic Graph-Based Service Composition (SeGSeC)

This section describes SeGSeC, a service composition mechanism to compose a service from multiple components based on the semantics of the service requested by a user. SeGSeC assumes that all components are modeled with CoSMoS. SeGSeC also depends on CoRE to discover and execute components. This section describes the architecture, implementation, and empirical evaluation of SeGSeC.

A. SeGSeC Architecture

SeGSeC consists of four modules: RequestAnalyzer, Service Composer, SemanticsAnalyzer, and ServicePerformer (Fig. 9).

When a user requests a service in a natural language [Fig. 9, (1)], RequestAnalyzer parses the request in a natural language into a CoSMoS semantic graph representation and, then, passes the semantic graph (i.e., the user request) to ServiceComposer [Fig. 9, (2)]. Upon receiving the user request from Request Analyzer, ServiceComposer discovers components based on the user request and creates a workflow using the discovered components [Fig. 9, (3)]. ServiceComposer, then, passes the workflow and the user request to SemanticsAnalyzer, which in

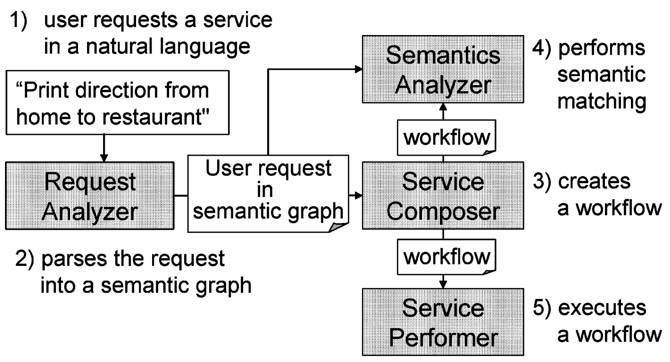

1) user requests a service in a natural language
"Print direction from home to restaurant"
Request Analyzer
User request in semantic graph
2) parses the request into a semantic graph
Semantics Analyzer
4) performs semantic matching
workflow
Service Composer
3) creates a workflow
workflow
Service Performer
5) executes a workflow

Fig. 9. Modules in SeGSeC.

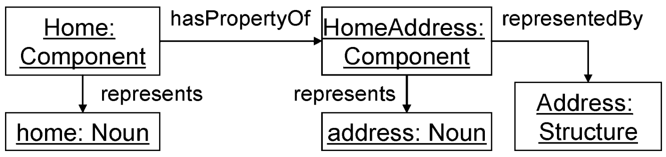


Fig. 10. Home component in CoSMoS.

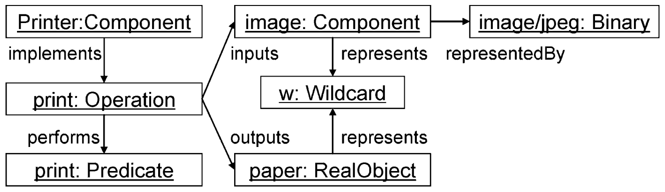


Fig. 11. Printer component in CoSMoS.

turn examines if the semantics of the workflow satisfies the user request [Fig. 9, (4)]. If SemanticsAnalyzer concludes that the semantics of the workflow satisfies the user request, it notifies the result to ServiceComposer. ServiceComposer then passes the workflow to ServicePerformer, which in turn executes the workflow [Fig. 9, (5)].

The following sections describe the details of Request Analyzer, ServiceComposer, SemanticsAnalyzer, and Service Performer using an example scenario of how the direction printing service (described in Section I) is composed from four components, Home (Fig. 10), Restaurant (Fig. 3), Direction Generator (Fig. 1), and Printer (Fig. 11), when a user requests the service by providing a sentence "print direction from home to restaurant."

1) RequestAnalyzer Module: When a user requests a service in a natural language (e.g., "print direction from home to restaurant"), RequestAnalyzer parses the request in a natural language into a CoSMoS semantic graph representation (e.g., Fig. 12). SeGSeC requires that the request contains one predicate (e.g., "print"). Since the natural language analysis is a well established area of research, SeGSeC assumes that RequestAnalyzer uses existing techniques (e.g., [28]) for parsing a natural language sentence into a semantic graph. After parsing a natural language sentence into a semantic graph, RequestAnalyzer passes the semantic graph (i.e., a user request) to ServiceComposer.

2) ServiceComposer Module: Upon receiving a user request (i.e., a semantic graph converted from a natural language sentence), ServiceComposer discovers components based on

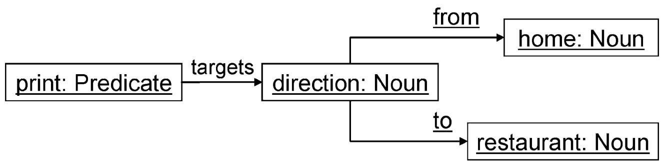


Fig. 12. User request converted into CoSMoS.

the user request and creates a workflow using the discovered components.

In order to create a workflow based on the user request, ServiceComposer first discovers an initial component whose operation *performs* (i.e., has a "performs" link to) the predicate specified in the user request, and creates a workflow that only contains the initial component. In the example scenario described in Section IV-A, ServiceComposer discovers the printer component (Fig. 11) as the initial component because its operation *performs* the "print" predicate. If ServiceComposer fails to discover a component whose operation *performs* the specified predicate, it tries to discover another component using a predicate that is compatible with the specified predicate. For example, if ServiceComposer does not discover any component whose operation *performs* the "print" predicate, but if it discovers a component specifying that the "output" predicate is compatible with the "print" predicate, then ServiceComposer tries to discover another component whose operation *performs* the "output" predicate. If ServiceComposer still cannot discover any component, it notifies the user that it failed to compose the requested service.

After discovering an initial component and creating a workflow containing the initial component, ServiceComposer discovers other components that provide the inputs of the operation of the initial component, and expands the workflow by adding those components into the workflow. This step is called Input Complement. The input complement is implemented as a recursive function (Fig. 13) with three arguments, an operation whose inputs need to be discovered, a user request, and a workflow that the function expands. ServiceComposer initiates the function by providing the operation of the initial component, the user request, and the workflow containing the initial component as the arguments.

In the input complement, ServiceComposer first identifies the inputs of the operation given as an argument of the function and discovers all the components whose outputs (of their operations) or properties are compatible with the inputs of the given operation [Fig. 13(a)]. ServiceComposer considers that an output and an input (or a property and an input) are compatible if their data types and concepts are compatible. In the example scenario described in Section IV-A, when ServiceComposer is performing the input complement for the "generate" operation in the direction generator (Fig. 1), it discovers the home (Fig. 10) and restaurant (Fig. 3) because their properties (i.e., "HomeAddress" and "RestAddress") are compatible with the two inputs (i.e., "originComp" and "destComp") of the "generate" operation. If an input of the given operation is a *choice* (i.e., an instance of *Choice* class), ServiceComposer uses the *literal* values contained in the *choice* instead of the components whose outputs or properties are compatible with the *choice*.

```
inputComplement(operation, userRequest, workflow){
    create a list of empty lists L[1] ... L[N]                              ⎫
    # N = the number of the inputs of the operation                         ⎪
    for each input i_x of the operation{                                    ⎪
        if i_x is a Choice{                                                 ⎪
            add its Literal values to L[x]                                  ⎬ (a)
        }else{                                                              ⎪
            discover outputs/properties that are compatible with i_x       ⎪
            add the discovered outputs/properties to L[x]                   ⎪
        }                                                                   ⎪
    }                                                                       ⎭

    compute possible combinations LC[1] ... LC[M] of                        ⎫
    the components in L[1] ... L[N]                                         ⎪
    # choose N components from each of L[1] ... L[N]                        ⎬ (b)
    # and store them into LC[x] such that the members of                    ⎪
    # two lists LC[x] and LC[y] are different if x!=y                       ⎭

    sort LC[1]...LC[M] based on the similarity values                       ⎫
    # similarity value: the number of concepts that appear in               ⎬ (c)
    # userRequest and the components in LC[x]                               ⎭

    for each LC[x]{
        specify in the workflow that the components in LC[x]                ⎫ (d)
            provide the inputs of the operation op                          ⎭
        if LC[x] contains outputs{                                          ⎫
            for each output{                                                ⎪
                identify the operation op2 that generates the output        ⎬ (e)
                inputComplement(op2, userRequest, workflow)                 ⎪
            }                                                               ⎭
        }else{                                                              ⎫
            if semanticMatching(userRequest, workflow)                      ⎪
            # if the semantics of the workflow satisfies                    ⎬ (f)
            # the user request                                              ⎪
                return workflow                                             ⎪
        }                                                                   ⎭
    }
}
```

Fig. 13. Input complement pseudocode.

After discovering the components that provide the inputs of the given operation (i.e., components whose outputs or properties are compatible with the inputs of the operation), ServiceComposer next determines which components among the discovered ones to add to the workflow [Fig. 13(b) and (c)]. In order to do so, ServiceComposer first considers all possible combinations of the outputs, properties, and *literal* values of the discovered components such that each of the combinations provides all the inputs of the given operation [Fig. 13(b)]. In the example scenario described in Section IV-A, Service Composer discovers that "HomeAddress" and "RestAddress" properties are compatible with the two inputs, "originComp" and "destComp" of the direction generator's "generate" operation. Therefore, ServiceComposer considers four combinations, {"HomeAddress," "HomeAddress"}, {"RestAddress," "RestAddress"}, {"HomeAddress," "RestAddress"}, and {"RestAddress," "HomeAddress"}, because each of these four combinations provides the two inputs of the Direction generator's "generate" operation. After considering all possible combinations, ServiceComposer calculates the similarity value of each combination. The similarity value of a combination is calculated as the number of the concepts that appear in both the user request and the semantic graph representations of the components that provide the outputs and properties contained in the combination. For example, in the example scenario, the similarity value of the combination {"HomeAddress," "RestAddress"} is 2 because two concepts, "home: Noun" and "restaurant: Noun," appear in both the user request (Fig. 12) and the semantic graph representations of the Home (Fig. 10) and Restaurant (Fig. 3). After calculating the similarity values of all the combinations, ServiceComposer selects the one with the highest similarity value [Fig. 13(c)]. In the example scenario, ServiceComposer selects either {"HomeAddress," "RestAddress"} or {"RestAddress," "HomeAddress"} because they have the highest similarity value of 2. Depending on which combination it selected, ServiceComposer will create different workflows.

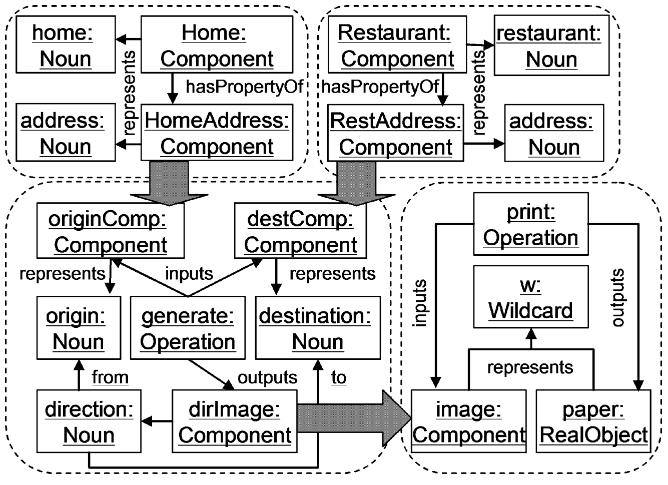
After selecting a combination, ServiceComposer expands the workflow by adding the components that provide the outputs and properties in the combination to the workflow so that the outputs and properties in the combination provide the inputs of the given operation [Fig. 13(d)]. ServiceComposer iterates the input complement, while the workflow contains an operation whose inputs need to be discovered [Fig. 13(e)]. In the example scenario described in Section IV-A, ServiceComposer iterates the input complements twice to discover the inputs of the "print" operation of the Printer (Fig. 11) and the inputs of the "generate" operation of the direction generator (Fig. 1). Then, ServiceComposer creates either of the two workflows shown in Fig. 14(a) and (b). Note that the workflow in Fig. 14(a) provides the requested service to print out the direction from home to the restaurant, but the workflow in Fig. 14(b) does not provide the requested service as it prints out the direction from the restaurant (i.e., not from home) to home (i.e., not to the restaurant). After creating a workflow, ServiceComposer passes the workflow and the user request to SemanticsAnalyzer to examine whether the semantics of the workflow satisfies the user request [Fig. 13(f)].

3) SemanticsAnalyzer Module: Upon receiving a workflow and the user request from ServiceComposer, SemanticsAnalyzer performs a step called Semantic Matching and examines whether the semantics of the workflow satisfies the user request or not. SemanticsAnalyzer examines the workflow by: 1) converting the workflow into a semantic graph; 2) adding new links to the semantic graph such that the resulting semantic graph models the semantics of the workflow; and 3) checking whether all the links in the user request also appear in the resulting semantic graph (i.e., the semantic graph that models the semantics of the workflow). The detail of the semantic matching is explained below in detail.

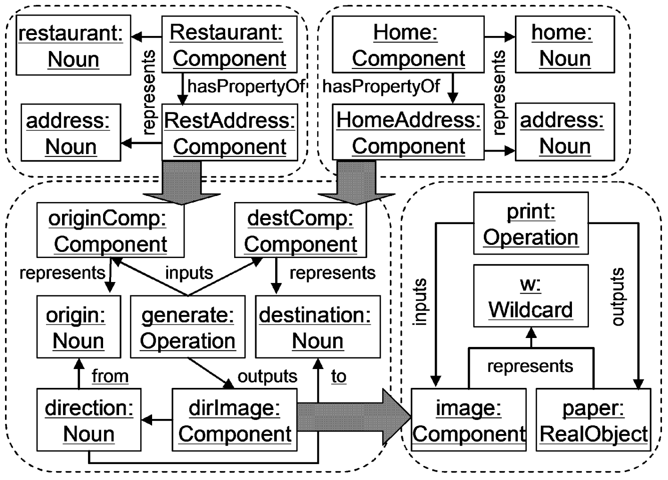
In the semantic matching, SemanticsAnalyzer first converts the workflow into a semantic graph. A workflow [e.g., Fig. 14(a)] consists of a set of components [e.g., "Home," "Restaurant," "DirectionGenerator," and "Printer" in Fig. 14(a)], each of which is modeled as a semantic graph [i.e., dotted boxes in Fig. 14(a)]. A workflow also specifies which output or property of a component provides an input of another component [shown as thick arrows in Fig. 14(a)]. If a workflow specifies that an input of a component is provided by an output (or a property) of another component, SemanticsAnalyzer connects the input and the output (or the property) with a "usedBy" link. This results in a single semantic graph consisting of the components (modeled as semantic graphs) connected by "usedBy" links. For instance, SemanticsAnalyzer converts the

TABLE I
SEMANTICS RETRIEVAL RULES

Name	Rule	Meaning
Rule 1	Operation(O) & Predicate(P) & Component(C) & Noun(S) & performs(P, O) & outputs(O, C) & represents(C, S) → targets(P, S)	For an operation O , a predicate P, a component C, and a noun S, if O performs P, O outputs C, and C represents S, then, S is the target of P.
Rule 2	Component(C) & Component(P) & Noun(Sc) & Noun(Sp) & hasPropertyOf(C, P) & represents(C, Sc) & represents(P, Sp) → applyTo(Sp, Sc)	For components C, P, and nouns Sc, Sp, if C has a property P, C represents Sc, and P represents Sp, then, apply Sp to Sc.
Rule 3	Component(X) & Component(Y) & Noun(Sx) & Noun(Sy) & usedBy(Y, X) & represents(X, Sx) & represents(Y, Sy) → applyTo(Sx, Sy)	For components X, Y, and nouns Sx, Sy, if Y is used by X (i.e., Y provides the input X), X represents Sx, and Y represents Sy, then, apply Sx to Sy.
Rule 4	Concept(Sx) & Concept(Sy) & Concept(Sz) & Concept(L) & Wildcard(W) & L(Sx, W) & L(Sy, Sz) & applyTo(Sx, Sy) → applyTo(W, Sz)	For concepts Sx, Sy, Sz, L, and a wildcard concept W, if Sx has a link L to W, Sy has the same link L to Sz, and Sx is applied to Sy, then, apply W to Sz.
Rule 5	Concept(Sx) & Concept(Sy) & Concept(L) & Resource(X) & applyTo(Sx, Sy) & L(X,Sx) → L(X, Sy)	For concepts Sx, Sy, L, and a resource X (i.e., any node X), if Sx is applied to Sy, and X has a link L to Sx, then, add the same link L from X to Sy, too.



(a)



(b)

Fig. 14. Workflows created after input complement. (a) Print direction from home to restaurant. (b) Print direction from restaurant to home.

workflow in Fig. 14(a) into a semantic graph by connecting {"HomeAddress," "originComp"}, {"RestAddress," "dest Comp"}, and {"image," "dirImage"} with "usedBy" links.

After converting the workflow into a semantic graph, SemanticsAnalyzer then adds new links to the semantic graph such that the resulting semantic graph models the semantics of

```
Apply Rule 1:
    performs('print: Operation', 'print: Predicate') &
    outputs('print: Operation', 'paper: RealObject') &
    represents('paper: RealObject', 'w: Wildcard')
    → targets('print: Predicate', 'w: Wildcard')
Apply Rule 3:
    usedBy('dirImage: Component', 'image: Component') &
    represents('image: Component', 'w: Wildcard') &
    represents('dirImage: Component', 'direction: Noun')
    → applyTo('w: Wildcard', 'direction: Noun')
Apply Rule 5:
    applyTo('w: Wildcard', 'direction: Noun') &
    targets('print: Predicate', 'w: Wildcard')
    → targets('print: Predicate', 'direction: Noun')
```

Fig. 15. Example of applying semantics retrieval rules (1).

```
Apply Rule 2:
    hasPropertyOf('Home: Component',
                  'HomeAddress: Component') &
    represents('Home: Component', 'home: Noun') &
    represents('HomeAddress: Component', 'address: Noun')
    → applyTo('address: Noun', 'home: Noun')
Apply Rule 3:
    usedBy('HomeAddress: Component',
           'originComp: Component') &
    represents('originComp: Component', 'origin: Noun') &
    represents('HomeAddress: Component', 'address: Noun')
    → applyTo('origin: Noun', 'address: Noun')
Apply Rule 5:
    applyTo('origin: Noun', 'address: Noun') &
    from('direction: Noun', 'origin: Noun')
    → from('direction: Noun', 'address: Noun')
Apply Rule 5:
    applyTo('address: Noun', 'home: Noun') &
    from('direction: Noun', 'address: Noun')
    → from('direction: Noun', 'home: Noun')
```

Fig. 16. Example of applying semantics retrieval rules (2).

the workflow. More precisely, SemanticsAnalyzer applies the predefined rules called *semantics retrieval rules* (Table I) onto the semantic graph. The *semantics retrieval rules* retrieve the semantics of the workflow by adding new links to the semantic graph converted from the workflow. For example, Figs. 15 and 16 illustrate how the *semantics retrieval rules* add the two links, "targets('print: Predicate,' 'direction: Noun')" and "from('direction: Noun,' 'home: Noun')," onto the semantic

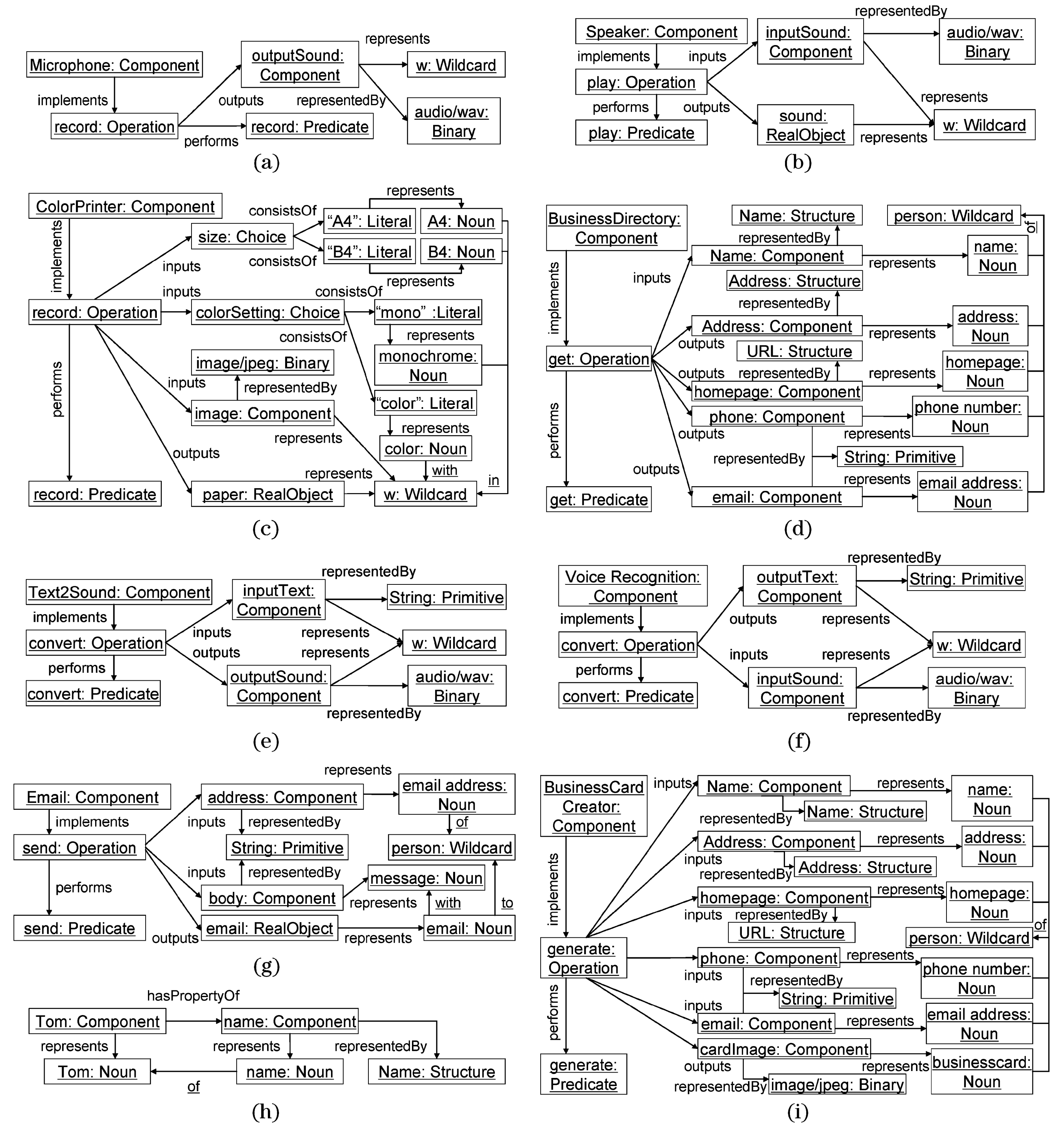


Fig. 17. Components implemented in SeGSeC. (a) Microphone. (b) Speaker. (c) Color printer. (d) Business directory. (e) Text-to-sound converter. (f) Voice recognition. (g) E-mail sender. (h) Tom. (i) Business card creator.

graph converted from the workflow shown in Fig. 14(a). In addition, if a component in the workflow provides a logic (i.e., if the semantic graph representation of a component in the workflow contains an "implies" link connecting two links that specify the condition and the consequence of the logic), and if the semantic graph converted from the workflow satisfies the condition of the logic (i.e., if the semantic graph converted from the workflow contains a link specified as the condition of the logic), SemanticsAnalyzer adds a new link as specified as the consequence of the logic onto the semantic graph. For example, if a component in the workflow provides the logic "implies [in('paper,' 'grayscale,') in('paper,' 'monochrome')]," shown in Fig. 7, and if the semantic graph converted from the workflow contains a link "in('paper,' 'grayscale')," SemanticsAnalyzer adds a new link "in('paper,' 'monochrome')" in the semantic graph.

After adding new links to the semantic graph converted from the workflow, SemanticsAnalyzer checks whether all the links

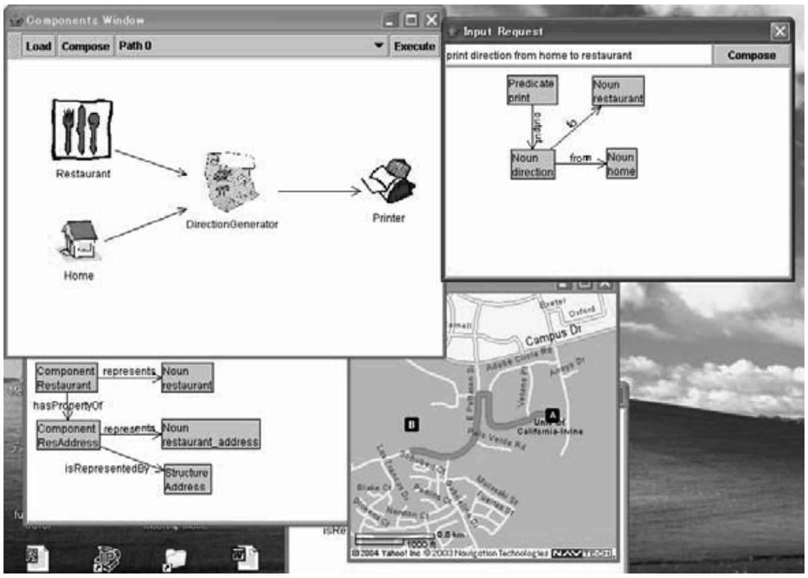


Fig. 18. User interface of the SeGSeC implementation.

in the user request also appear in the semantic graph (i.e., the semantic graph that models the semantics of the workflow). If all the links in the user request appear in the semantic graph, SemanticsAnalyzer concludes that the semantics of the workflow satisfies the user request. In the example scenario described in Section IV-A, if SemanticsAnalyzer receives the workflow shown in Fig. 14(a) and the user request shown in Fig. 12, SemanticsAnalyzer concludes that the semantics of the workflow satisfies the user request because all the links in the user request [i.e., "targets('print: Predicate,' 'direction: Noun')," "from('direction: Noun,' 'home: Noun')," and "to('direction: Noun,' 'restaurant: Noun')"] appear in the semantic graph converted from the workflow. However, if SemanticsAnalyzer receives the workflow shown in Fig. 14(b), it concludes that the semantics of the workflow does not satisfy the user request, because the semantic graph converted from the workflow does not contain the "from('direction: Noun,' 'home: Noun')" and "to('direction: Noun,' 'restaurant: Noun')" links.

If SemanticsAnalyzer concludes that the semantics of the workflow satisfies the user request, it notifies the result to ServiceComposer. ServiceComposer then 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 SemanticsAnalyzer concludes that the semantics of the workflow does not satisfy the user request, or if the user replies negatively, ServiceComposer tries to create other workflows based on the user request.

4) ServicePerformer Module: Upon receiving a workflow from ServiceComposer, ServicePerformer executes the workflow by invoking the operations of the components and retrieving the properties of the components as specified in the workflow.

B. SeGSeC Implementation

In order to demonstrate the feasibility of the proposed semantic-based dynamic service composition, SeGSeC has been implemented in Java.[3] The SeGSeC implementation allows a user to deploy components on a local host and request a service using a natural language. The current SeGSeC implementation can compose several services from various components such as home (Fig. 10), restaurant (Fig. 3), direction generator (Fig. 1), printer (Fig. 11), microphone [Fig. 17(a)], speaker [Fig. 17(b)], color printer [Fig. 17(c)], business directory[4] [Fig. 17(d)], text-to-sound converter [Fig. 17(e)], voice recognition [Fig. 17(f)], e-mail sender [Fig. 17(g)], a component modeling a person named Tom [Fig. 17(h)], and business card creator[5] [Fig. 17(i)]. For instance, it can compose the direction printing service described in Section IV-A. Fig. 18 shows screenshots of the user interface of the current SeGSeC implementation.

C. Empirical Evaluation

This section describes the empirical performance evaluation of SeGSeC. The performance of SeGSeC is affected by the number of components deployed. When a small number of components are deployed, SeGSeC composes the requested service in a short period of time because it only needs to discover and identify necessary components among a small number of components. However, as the number of components deployed becomes larger, SeGSeC may require a longer period of time for composing the requested service, as it needs to discover and identify necessary component among a larger number of components.

In order to examine how the performance of SeGSeC depends on the number of components deployed, a series of performance measurements have been conducted[6] using the current SeGSeC implementation, the 13 components described

[3]The current implementation is available for download at [24].

[4]Given a name of a person, the business directory provides the address, phone number, homepage, and e-mail address of the person.

[5]Given a name, address, phone number, e-mail address and homepage of a person, the business card creator creates an image of a business card of the person.

[6]The measurement was conducted on a Pentium 4 1.7 GHz machine with 384 MB memory and J2SE ver.1.5.0.

TABLE II
RESULTS OF THE FIRST SET OF MEASUREMENTS

User request	Components deployed	Service composition time
print direction from home to restaurant	Home, Restaurant, Direction Generator, Printer	18.1 msec
play direction from home to restaurant	Home, Restaurant, Direction Generator, Text2Sound, Speaker	61.3 msec
print direction from restaurant to Tom	Restaurant, Tom, Direction Generator, Business Directory, Printer	77.0 msec
send email to Tom	Tom, Microphone, Voice Recognition, Email Sender	38.1 msec
print businesscard of Tom	Tom, Business Directory, Business Card Creator, Printer	76.3 msec

in Section IV-B and the five example user requests: “print direction from home to restaurant,” “play direction from home to restaurant,” “print direction from restaurant to Tom,” “send e-mail to Tom,” and “print businesscard of Tom.” In the first set of measurements, the service composition time (i.e., the time period from when a user requests a service to when the user is asked whether to execute the composed service) was measured for the case where only the components required for composing the requested service are deployed. The second set of measurements was then conducted after deploying additional components to the configuration used for the first set of measurements. In the second set of measurements, the following metrics were measured: 1) the average time for ServiceComposer to discover components necessary to compose the requested service; 2) the average time for ServiceComposer to perform the Input complement and create workflows; 3) the average time for SemanticsAnalyzer to perform the semantic matching and identify the workflow whose semantics satisfies the user request; and 4) the average service composition time. In addition to examining how these performance metrics depend on the number of components deployed, the second set of measurements also examined how these metrics depend on the number of operations the deployed components implement and the total number of the nodes[7] in the CoSMoS semantic graph representations of the deployed components. Table II shows the results of the first set of measurements, and Fig. 19 shows the results of the second set of measurements.

Table II shows the user requests used in the measurements, as well as the components deployed and the service composition time for each of the user requests. Table II illustrates that the current SeGSeC implementation composes services in a reasonable time (i.e., less than a second) when only the components required to compose the requested service are deployed.

Fig. 19 illustrates the breakdown of the service composition time shown in Table II to investigate which step of SeGSeC dominates the overall performance of SeGSeC in different configurations. Fig. 19 illustrates that the semantic matching performed by SemanticsAnalyzer dominates the overall service composition time when the number of components [Fig. 19(a)], operations [Fig. 19(b)], and nodes [Fig. 19(c)] is small, whereas the overhead of discovering components becomes significant as the number of components, operations, and nodes increases. Fig. 19 also shows that the times required for the input complement and for the semantic matching remain relatively constant even when the number of components, operations, and nodes becomes large. This implies that the input complement and the semantic matching scale to these factors and that the scalability of SeGSeC primarily depends on whether ServiceComposer scales in discovering components.

[7]For example, the number of nodes modeling the microphone component in Fig. 17 is six.

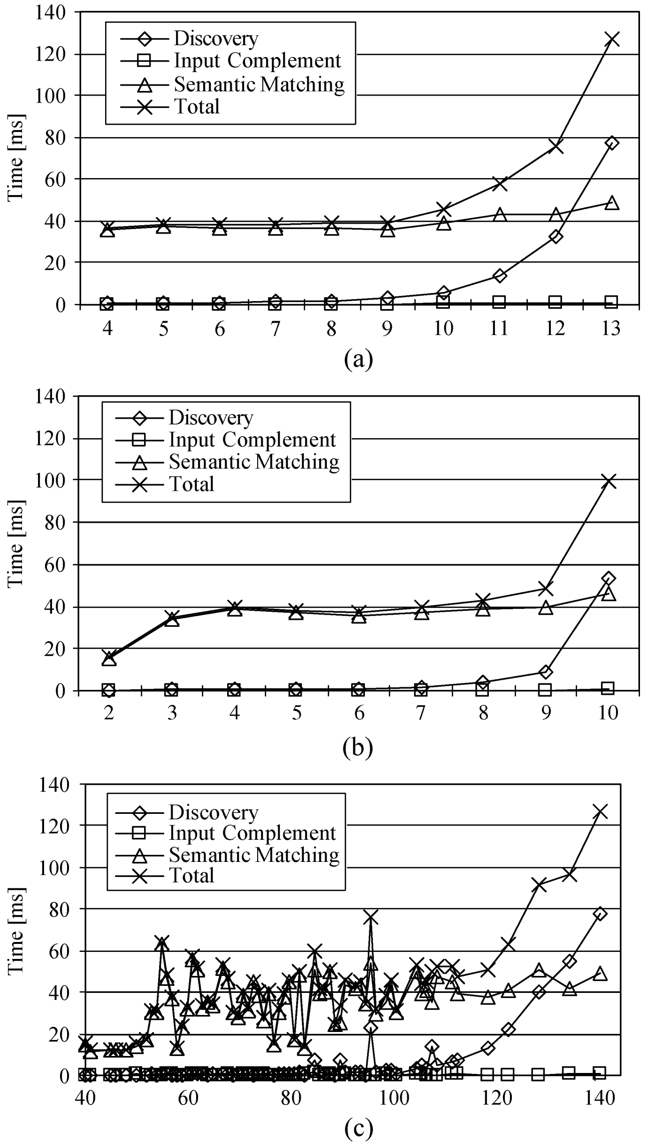


Fig. 19. Results of the second set of measurements. (a) Number of components. (b) Number of operations. (c) Number of nodes.

In summary, the series of measurements have shown that SeGSeC performs efficiently when only a small number of components are deployed and that SeGSeC scales to the number of components deployed if it discovers components efficiently.

V. Conclusion and Future Work

This paper proposes the semantics-based dynamic service composition architecture. The proposed architecture assumes that a user requests a service in an intuitive manner (e.g., using a natural language) and dynamically composes the requested service based on its semantics. The proposed architecture consists of a semantics-aware component model CoSMoS, a middleware CoRE, and a semantics-based service composition mechanism SeGSeC. This paper presents the design, implementation, and empirical evaluation of the proposed architecture.

The empirical evaluation of the proposed architecture presented in the paper provides insights to the behavior of the proposed architecture. It is still necessary to conduct further empirical evaluation of the proposed architecture to investigate its adaptability that is not investigated in this paper. In addition, the proposed architecture may be extended to compose services not only based on the semantics of the requested service but also based on the user evaluation on the services composed in the past in order to improve its adaptability. This awaits further research.

References

[1] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in eFlow," in *Proc. Int. Conf Advanced Inf. Syst. Eng.*, Stockholm, Sweden, 2000.

[2] D. Mennie and B. Pagurek, "An architecture to support dynamic composition of service components," in *Proc. 5th Int. Workshop Component-Oriented Program.*, Sophia Antipolis, France, 2000.

[3] M. Minami, H. Morikawa, and T. Aoyama, "The design and evaluation of an interface-based naming system for supporting service synthesis in ubiquitous computing environment," *Trans. Inst. Electron., Inf. Commun. Eng.*, vol. J86-B, no. 5, pp. 777–789, May 2003.

[4] 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 *Proc. 28th Very Large Database Conf.*, Hong Kong, China, Aug. 2002.

[5] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma, "Dynamic workflow composition using Markov decision processes," in *Proc. 2nd Int. Conf. Web Serv.*, San Diego, CA, Jul. 6–9, 2004, pp. 576–582.

[6] M. Marazakis, D. Papadakis, and C. Nikolaou, "The aurora architecture for developing network-centric applications by dynamic composition of services," FORTH/ICS, Tech. Rep. TR 213, 1997.

[7] P. Pires, M. Mattoso, and M. Benevides, "Building Reliable Web Services Compositions," in *Lecture Notes in Computer Science*. New York: Springer-Verlag, 2003, vol. 2593, Web, Web-Services, and Database Systems 2002, pp. 59–72. ISBN 3-540-00745-8.

[8] P. Traverso and M. Pistore, "Automated composition of semantic web services into executable processes," in *Proc. 3rd Int. Semantic Web Conf.*, Hiroshima, Japan, Nov. 7–11, 2004.

[9] W. Cheung, J. Liu, K. Tsang, and R. Wong, "Toward autonomous service composition in a grid environment," in *Proc. IEEE Int. Conf. Web Serv.*, San Diego, CA, Jul. 2004.

[10] S. Chandrasekaran, S. Madden, and M. Ionescu, "Ninja workflows: An architecture for composing services over wide area networks," Univ. California, Berkeley, CA, CS262 class project writeup, 2000.

[11] S. R. Ponnekanti and A. Fox, "SWORD: A developer toolkit for web service composition," in *Proc. 11th World Wide Web Conf. (Web Eng. Track)*, Honolulu, HI, May 7–11, 2002.

[12] E. Sirin and B. Parsia, "Planning for semantic web services," in *Proc. Semantic Web Services Workshop 3rd Int. Semantic Web Conf.*, 2004.

[13] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, "Automating DAML-S web services composition using SHOP2," in *Proc. 2nd Int. Semantic Web Conf.*, Sanibel Island, FL, Oct. 2003.

[14] B. Limthanmaphon and Y. Zhang, "Web service composition with case-based reasoning," in *Proc. 14th Australasian Database Conf.*, K.-D. Schewe and X. Zhou, Eds., Adelaide, Australia, 2003, pp. 201–208.

[15] S. McIlraith and T. Son, "Adapting Golog for composition of semantic web services," in *Proc. 8th Int. Conf. Knowl. Representation Reasoning*, Apr. 2002, pp. 482–493.

[16] M. Sheshagiri, M. desJardins, and T. Finin, "A planner for composing services described in DAML-S," in *Proc. Workshop Planning Web Services*, Jul. 2003.

[17] J. Rao, P. Kungas, and M. Matskin, "Logic-based web service composition: From service description to process model," in *Proc. IEEE Int. Conf. Web Serv.*, San Diego, CA, Jul. 6–9, 2004, pp. 446–453.

[18] K. Fujii and T. Suda, "Dynamic service composition using semantic information," in *Proc. 2nd Int. Conf. Service Oriented Comput.*, Nov. 2004.

[19] OWL-S 1.0 release. [Online]. Available: http://www.daml.org/services/owl-s/1.0/

[20] Web service modeling ontology. [Online]. Available: http://www.wsmo.org/

[21] A. Eberhart, "Ad hoc invocation of semantic web services," in *Proc. IEEE Int. Conf. Web Serv.*, Jul. 2004, pp. 116–123.

[22] METEOR-S: Semantic web services and processes. [Online]. Available: http://lsdis.cs.uga.edu/Projects/METEOR-S/

[23] J. Peer. Semantic service markup with SESMA version 0.8. [Online]. Available: http://elektra.mcm.unisg.ch/pbwsc/

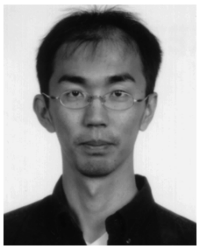
[24] K. Fujii. Dynamic service composition. [Online]. Available: http://netresearch.ics.uci.edu/kfujii/dsc/

[25] T. Nakano and T. Suda, "Self-organizing network services with evolutionary adaptation," *IEEE Trans. Neural Netw. (Special Issue on Adaptive Learning Systems in Communication Networks)*, vol. 16, no. 5, pp. 1269–1278, Sep. 2005.

[26] J. Suzuki and T. Suda, "A middleware platform for a biologically-inspired network architecture supporting autonomous and adaptive applications," *IEEE J. Sel. Areas Commun. (Special Issue on Intelligent Services and Applications in Next Generation Networks)*, vol. 32, no. 2, pp. 249–260, Feb. 2005.

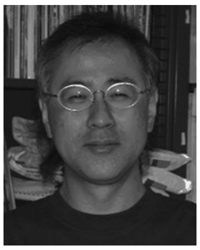
[27] M. Wang and T. Suda, "The bio-networking architecture: A biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications," in *Proc. 1st IEEE Symp. Appl. Internet (SAINT)*, 2001.

[28] G. A. Mann, "BEELINE—a situated, bounded conceptual knowledge system," *Int. J. Syst. Res. Inf. Sci.*, vol. 7, pp. 37–53, 1995.



Keita Fujii (S'04) received the B.E. and M.S. degrees in computer science from Waseda University, Tokyo, Japan, in 1999 and 2001, respectively. He is currently working towards the Ph.D. degree at the School of Information and Computer Science, University of California, Irvine.

His main research interests span various areas in computer communication networks, including distributed computing, ubiquitous computing, agent-based computing, and middleware technologies.



Tatsuya Suda (S'80–M'82–SM'97–F'01) received the B.E., M.E., and Dr.E. degrees in applied mathematics and physics from Kyoto University, Kyoto, Japan, in 1977, 1979, and 1982, respectively.

From 1982 to 1984, he was with the Department of Computer Science, Columbia University, New York, as a Postdoctoral Research Associate. Since 1984, he has been with the Department of Information and Computer Science, University of California, Irvine, where he is currently a Professor. He has also served as a Program Director of the Networking Research Program, National Science Foundation from October 1996 to January 1999. He was a visiting Associate Professor at the University of California, San Diego, a Hitachi Professor at the Osaka University, and currently is a NTT Research Professor. He is an Area Editor of the *International Journal of Computer and Software Engineering*. He is a member of the Editorial Board of the Encyclopedia of Electrical and Electronics Engineering, Wiley. He has been engaged in research in the fields of computer communications and networks, high-speed networks, multimedia systems, ubiquitous networks, distributed systems, object oriented communication systems, network applications, performance modeling and evaluation, and application of biological concepts to networks and network applications.

Dr. Suda is a member of the Association for Computing Machinery (ACM). He received an IBM Postdoctoral Fellowship in 1983. He was the Conference Coordinator from 1989 to 1991, the Secretary and Treasurer from 1991 to 1993, the Vice Chairman from 1993 to 1995, and the Chairman from 1995 to 1997 of the IEEE Technical Committee on Computer Communications. He was also the Director of the U.S. Society Relations of the IEEE Communications Society from 1997 to 1999. He is an Editor of the IEEE/ACM Transactions on Networking, a Senior Technical Consultant to the IEEE Transactions on Communications, and a former Editor of the IEEE Transactions on Communications. He was the Chair of the 8th IEEE Workshop on Computer Communications and the TPC Co-Chair of the IEEE INFOCOM 1997.