Component Service Model with Semantics (CoSMoS): A New Component Model for Dynamic Service Composition

Keita Fujii and Tatsuya Suda
University of California, Irvine
{kfujii, suda}@ics.uci.edu

Abstract

Dynamic service composition provides the potential for future applications to adapt to various user preferences and contexts. Despite its difficulties, dynamic service composition provides several benefits, namely, flexibility, adaptability, and availability. However, existing dynamic service composition systems lack extensibility and understandability. Also some systems cannot support operation dependency and soft-state information. To overcome the problems of the existing systems, this paper proposes a new component model called Component Service Model with Semantics, or CoSMoS. CoSMoS resolves the problems of the existing systems by applying semantic graphs, ontologies and an object-oriented model. This paper describes the structure of CoSMoS and shows how dynamic service composition can be implemented using CoSMoS.

1. Introduction

Future applications are expected to adapt to various user preferences and contexts. Dynamic service composition, a set of techniques to compose a service (or application) on demand through combining multiple components, has the potential to realize such applications. Despite its difficulties, dynamic service composition provides several benefits to the emerging applications, namely, flexibility, adaptability, and availability. It provides flexibility as a number of useful applications may be composed from a set of available components. It provides adaptability as applications can be composed according to the user preference and context. A new application that was not envisioned at design time may emerge. It achieves higher availability as malfunctioning components in the application can be substituted with alternative components. By discovering and combining a proper set of components for a given request on demand, dynamic service composition can compose a flexible, adaptable and robust application.

There are some existing systems that implement dynamic service composition. This paper classifies the existing systems into three categories based on their approaches, namely, template-based systems, interface-based systems, and logic-based systems, and discusses the features and issues of each category. This paper claims that logic-based systems are more adaptable than others, yet they lack extensibility and understandability, and that the component model applied in many of the existing systems lacks the notion of operation dependency and soft-state (or session) information, which are necessary to compose an application with complex interactions.

After the discussions of the existing systems, this paper proposes a new component model called Component Service Model with Semantics, or CoSMoS. CoSMoS overcomes the problems of the existing systems by applying semantic graphs, ontologies and an object-oriented model. CoSMoS introduces semantic graphs and ontologies to define and annotate semantics of a component. The semantic graphs and ontologies bring an extensible and intuitive representation of semantic information. CoSMoS applies an object-oriented model that allows outputs of an operation to be components that may have operations. By supporting the object-oriented model, CoSMoS represents operation dependency and soft-state information naturally. By using CoSMoS, applications with complex interactions can be dynamically composed from an intuitive request.

The remainder of this paper is organized as following. Section 2 discusses the classification of existing dynamic service composition systems and their problems. Section 3 describes the structure of CoSMoS and explains how the problems of the existing systems are solved in CoSMoS. Section 4 briefly shows the overview of a service composition algorithm based on CoSMoS. Section 5 describes the implementation of CoSMoS. Section 6 concludes this paper.

2. Discussion of existing systems

Although dynamic service composition is a relatively new topic, several systems that implement dynamic service composition have been proposed. This section classifies the existing systems into three categories, namely, template-based systems, interface-based systems, and logic-based systems, and discusses the features and issues of each category.
Template-based systems

Template-based systems [1, 2] compose an application from a given service template. A service template defines (1) types or rules of the components required for composing an application, and (2) a structure of the application, such as an order of execution of the components. In the template-based systems, a user requests an application either by choosing a service template from a repository or by creating a template by himself. The requested application is composed through selecting the components specified in the template and combining them according to the structure described in the template. Figure 1(a) shows an example of composing a “map printing service” using the template-based approach.

The template-based systems can compose applications that involve complex interactions between components and can provide some flexibility to the emerging applications by choosing different sets of components. However, the template-based systems cannot compose applications whose templates are not available. For example, a user cannot print with his printer if he does not have a template to use a printer. Also, relying on users to create service templates is not always acceptable because creating a template requires technical knowledge and experiences. Therefore, the template-based systems have limited adaptability.

Interface-based systems

Interface-based systems [3] use interface information of components instead of service templates in order to compose an application. Interface information of a component consists of operations, each of which is a set of inputs (or arguments) and outputs (or return values). An operation is the same concept as the function, procedure or method in programming languages. In the interface-based systems, a user requests an application by submitting interface information (i.e., a set of inputs and outputs) of the application he is requesting. The requested application is composed through combining components such that the combination of the components accepts the requested inputs and generates the requested outputs. Figure 1(b) shows an example of composing the “map printing service” using the interface-based approach.

Unlike the template-based systems, the interface-based systems can compose an application as long as there exist proper components and an appropriate request is provided. Thus, the interface-based systems provide higher adaptability than the template-based systems. However, the interface-based systems have several problems. First, certain applications cannot be represented as a set or inputs and outputs. For example, an email sending service requires an email address and some text as inputs, but does not output any data. Therefore, it is not easy to model an email sending service as a set of inputs and outputs. Also, interface information often lacks semantic information about the application. For example, consider a direction generation service which depicts a graphical map showing a direction from one address to another. This direction generation service accepts two addresses as its inputs and generates an image as its output. However, without any extra notation, one cannot tell which address of the two inputs is the starting point, and which address is the destination. This kind of semantic information is either implicitly annotated as argument names (e.g., “generate(Address start, Address destination”) or explicitly described as comments so that programmers can recognize. But in order to compose an application autonomously, this semantic information should also be represented in machine-understandable format.

Logic-based systems

Logic-based systems [4, 5] extend the interface-based approach by adding extra information (i.e., precondition and postcondition) into interface information using first-order logic. A user requests an application by submitting a first order formula representing the logic that must be satisfied by the application. The requested application is composed through combining components such that the combination of the logics specified in the components is equivalent to the logic specified by the user. Figure 1(c) shows an example of the logic-based approach.

First-order logic can represent applications that interface information (i.e., a set of inputs and outputs) cannot represent. For example, the email sending service can be modeled as “known(address(email)) & known(content(email)) → sent(email)”. Also, first-order logic can represent semantic information about the application. For instance, the direction generation service can be represented as “address(A) & address(B) → direction(A,B) & start(A) & destination(B)”. Thus, the
logic-based systems are more adaptable than the template-based since they do not require service templates, and support more varieties of service than the interface-based systems.

However, the logic-based systems are not extensible and not suitable for a distributed environment because of the following reason. Logic-based systems require all components (or service providers) and users to agree and share the same logic definition (i.e., a set of functions and variables). This requirement prohibits a service provider from implementing a new type of service if the service cannot be represented in the existing logic definition, leading the logic-based systems to be application domain specific. Also, this requirement raises the issue of how to distribute the logic definition in a distributed environment. In addition, first-order formulas may not be easy to understand. For example, the statement “male and female are opposite” is represented as “∀x male(x) -> ¬ female(x)” in first-order formula, but this formula is not easy for normal users to understand.

**Procedure-oriented Component Model**

There are also other problems in the existing systems that arise not from their approach but from their component model. Some existing systems [3, 4, 5] are based on a procedure-oriented model, where the output of an operation of a component (i.e., return value) is not allowed to have an operation. This restriction is crucial in achieving dynamic service composition due to the following issues.

One issue of the procedure-oriented model is the lack of dependency information between operations. It is often the case that a component has multiple operations and there exist some dependencies between them, such as invocation order or mutual exclusion. For example, assume a printer component that spools several documents and supports cancellation of the spooled documents. The printer component should have two operations: **print** and **cancel**. For human designers, it is intuitive that **cancel** should be invoked only after **print** is invoked. However, without any explicit notation, it is impossible for computers to mechanically derive this dependency information. This dependency information is necessary when composing an application that involves multiple operations.

Another issue of the procedure-oriented model is the lack of soft-state information to handle sessions. In the procedure-oriented model, users have to keep soft-state information to associate a sequence of interactions (i.e., a session). For example, in the previous example of the printer component, each user has to keep a session ID of the spooled document so that he can only cancel the document he has requested. Thus, the printer component has to implement **print** to return a session ID of the input document, and implement **cancel** to take the session ID as an input. However, no information tells users to keep the session ID returned by **print** and provide it to **cancel**. Soft-state information is important when composing an application that handles multiple interactions as a single session.

This section explained several existing approaches to realize dynamic service composition, and discussed the problems of each approach. A template-based approach can compose a complex application, but is not adaptable because it requires service templates to compose applications. An interface-based approach is more adaptable than the template-based approach. However, it cannot support all the kinds of services since some services cannot be represented as a set of inputs and output. It also lacks semantics of service. A logic-based approach solves the issues of the interface-based approach by adding first-order logic into interface information, yet it has some problems in its extensibility and understandability. Also, some existing systems are based on the procedure-oriented component model, which induces operation dependency and soft-state problems. The next section introduces a new component model called Component Service Model with Semantics (CoSMoS). CoSMoS is designed to solve those problems that occur in the logic-based approach and the procedure-oriented model, i.e., extensibility, understandability, operation dependency problem, and soft-state problem.

### 3. Component Service Model with Semantics (CoSMoS)

This section introduces a new component model called Component Service Model with Semantics, or CoSMoS. CoSMoS is designed to solve the problems of the logic-based approach and the procedure-oriented model by applying semantic graphs, ontologies, and an object-oriented model. A semantic graph is an intuitive way to represent high-level concepts in a flexible manner. With the support of ontologies, semantics (or logic) of a component can be freely defined, thus providing high extensibility. The object-oriented model solves the operation dependency and soft-state problems caused by the procedure-oriented model. The rest of this section explains how the semantic graphs, ontologies and object-oriented model are used in CoSMoS, which is followed by the detailed design of CoSMoS.

**Semantic Graph**

A semantic graph is an intuitive way to represent high-level concepts in a flexible manner. A semantic graph is represented as a graph of nodes where nodes
represent resources or concepts and links between them represent associations between resources and concepts. For instance, a sentence “Keita Fujii is the owner of this book” can be represented as Figure 2. This graph representation is commonly used, for example, in Conceptual Graph [6] or RDF [7].

![Figure 2: Example of semantic graph](image)

In CoSMoS, semantic graphs are used to (1) define high-level abstracts, such as concepts, states or objects in real world, (2) represent associations between high-level abstracts and components, and (3) define logic about operations of a component. Details of how semantic graphs are used to represent such information will be explained later.

The benefit of using semantic graphs for dynamic service composition is its understandability and flexibility. Service composition based on semantic graphs is expected to be easier than service composition based on service templates or interface information because service graphs are more understandable and thus easier for users to generate and handle. For example, a semantic graph can be obtained from a natural language through syntactic and semantic analysis [8]. Also, semantic graphs are flexible and powerful enough to represent the same amount of details as a first-order formula. Another benefit of using semantic graphs is that standard models and description frameworks (e.g., RDF, N3 [8]) can be applied to represent the graphs. It is also possible to utilize existing techniques and implementation [9] to store and process semantic graphs.

### Ontology

An ontology is a formal representation of the specification of concepts or relationships of different concepts that exist for different communities. In other words, an ontology is a specification of metadata that defines classes or concepts and relationships between them. Anybody can define his own ontology by specifying the mapping between his own ontology and other existing ontologies. An ontology can be defined separately from semantic graphs, and many different ontologies defined by different parties can coexist.

The benefits of applying ontologies on dynamic service composition are following: First, ontologies provide high extensibility since by allowing a component designer to define his own ontology. This is more extensible than the logic-based approach based on first-order formula where all components (or service providers) and users must agree on and share the same logic definition (i.e., a set of functions and variables).

Second, an ontology-based system is suitable for distributed environment, because ontologies can be separately defined from semantic graph and different ontology can coexist by supplying mapping between them. Lastly, existing ontology technologies such as RDF, RDF Schema [10], and OWL [11] can be applied. By applying these standard technologies, semantic and ontology can be used not only for service composition but also for service discovery or some other purposes.

### Object-oriented Component Model

As described in section 2, the procedure-oriented component model has operation dependency and soft-state problems. The operation dependency problem occurs when there are multiple operations and there are some dependencies between them, such as invocation order or mutual exclusion. The soft-state problem refers to the lack of soft-state information to handle sessions. CoSMoS solves these problems by applying an object-oriented model that allows inputs and outputs of an operation to be components that may have operations.

The object-oriented model can represent operation dependency naturally. In the object-oriented model, the printer component in the previous example can be implemented such that the return value of the print operation (called PrintSession) implements the cancel operation (Table 1). In this way, cancel cannot be invoked before print, thus the object-oriented model expresses the dependency between the print and cancel operations.

Also, in the object-oriented model, operations involving soft-state can be modeled as accesses to an object encapsulating soft-state. In the sample codes in Table 1, a PrintSession object encapsulates soft-state (i.e., session ID) of the spooled document, thus users do not have to keep a session ID to cancel a document. Since the object-oriented model is capable of representing operation dependency and soft-state in a natural way without requiring any explicit notion, supporting the object-oriented model simplifies autonomous service composition.

<table>
<thead>
<tr>
<th>Table 1: Procedure-oriented vs. Object-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure-oriented</td>
</tr>
<tr>
<td>Class Printer{</td>
</tr>
<tr>
<td>int print(Document doc){</td>
</tr>
<tr>
<td>…</td>
</tr>
<tr>
<td>//return session ID</td>
</tr>
<tr>
<td>//of this document</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>void cancel(int sessionID){</td>
</tr>
<tr>
<td>// sessionID must be</td>
</tr>
<tr>
<td>// the one returned</td>
</tr>
<tr>
<td>// by print() method</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
CoSMoS Structure

CoSMoS consists of two layers: a semantic layer that represents semantic information (i.e., concepts, states or objects in real world) and a functional layer that represents functional information (i.e., components and their operations). Each layer consists of nodes and links, and a network of nodes and links models a component.

In the semantic layer, a node represents a concept (e.g., “person”, “room”, “read”), a state (e.g., “the printer prints the paper”), or an object in the real world (e.g., “Bob”, “Conference Room X”). Concept nodes are needed to support service composition using high-level abstracts. State nodes and Object nodes allow representing any service as a set of input nodes and output nodes, and thus expand the applicable scope of the interface-based approach. Links between nodes in the semantic layer represent the associations of nodes. For example, Figure 3 shows that the concept “Address” represents the destination of the concept “Map”.

![Figure 3: Example of semantic layer](image)

In the functional layer of CoSMoS, nodes represent components themselves as well as operations of components, and links between nodes represent associations between them. For example, an operation “Convert” which converts a HTML file into Postscript can be modeled as Figure 4.

![Figure 4: Example of functional layer](image)

Links can be established between two layers or between different components by specifying the URIs of the nodes. Figure 5 shows an example of CoSMoS representing a direction generator component that generates a graphical map showing a direction from one address to another.

![Figure 5: Direction Generator in CoSMoS](image)

Table 2: Associations defined in CoSMoS

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>represents</td>
<td>The source resource represents the target concept</td>
</tr>
<tr>
<td>performs</td>
<td>The source interface performs the target proposition</td>
</tr>
<tr>
<td>implements</td>
<td>The source interface implements the target operation</td>
</tr>
<tr>
<td>consistsOf</td>
<td>The source resource consists of the target resource</td>
</tr>
</tbody>
</table>

We chose this two-layer model because it is complex and not intuitive to represent functional information using a vocabulary for semantic information. By separating semantic information and functional information, different description languages can be used to represent each of them. As described later, our description language for CoSMoS, CSDF (Component Service Description Framework), uses RDF Schema for describing semantic layer and WSDL-like language for describing functional layer.

CoSMoS allows a component designer to define arbitrary concept nodes and object nodes as well as arbitrary associations through providing the mapping (i.e. ontology) of his original nodes onto other ones. Since an ontology can also be defined using a semantic graph, ontologies in CoSMoS are defined in the semantic layer. Figure 6 shows an example ontology showing that the concept “read” has the same meaning as the concept “view”.

![Figure 6: Example of ontology](image)

![Figure 7: Class diagram of CoSMoS](image)
CoSMoS applies the model used in RDF Schema, where nodes are modeled as rdf:Classes, and links are modeled as rdf:Properties. CoSMoS uses existing ontologies (e.g., RDF Schema, N3, OWL), but CoSMoS also defines its own ontology to specify the associations that are not defined in the existing ontologies. Table 2 shows some example associations defined in CoSMoS. Figure 7 shows the class diagram of CoSMoS. The detailed specification of CoSMoS is available at [12].

4. Service Composition using CoSMoS

This section shows the overview of our service composition algorithm based on CoSMoS. We are currently designing a distributed algorithm to efficiently find a composite application upon receiving a service composition request. However, due to the space limitation and the fact that the algorithm is still being refined, this paper only shows the high level overview of our algorithm.

Service composition process starts when a user submits a composition request. A service composition request is represented in CoSMoS. Figure 8 shows an example service composition request that requests a service to print out directions from a home to a restaurant. Based on the provided service composition request, an application is composed using the following techniques.

**Induction**: Induction restructures a portion of the composition request into a different structure without losing the semantics of the request. In other words, Induction takes a set of nodes and links from the request and replaces them with a new set of nodes and links without losing the semantics of the whole structure. (Figure 9(a))

**Replacement**: Replacement replaces a node in the composition request with another (set of) node(s) representing the same concept. (Figure 9(b))

**Interface discovery**: If the composition request contains a proposition node, interface discovery discovers an operation node that performs the proposition, and also checks whether the operation is implemented by a component or by an interface node. If the operation node is implemented by a component, input complement is performed next. Otherwise (i.e., if the operation node is implemented by an interface node, meaning that there is no component that directly implements the proposition), interface matching is performed next. (Figure 9(c))

**Input complement**: Given an operation (i.e., a set of inputs and outputs) of a component, input complement finds a path of components that generates the input(s) of the operation so that the component can be executed. (Figure 9(d))

**Interface matching**: Given an operation (i.e., a set of inputs and outputs), interface matching finds a path of components to convert the input(s) into the output(s). (Figure 9(e))

Figure 10 shows the example of how a direction printing service can be composed from the request depicted in Figure 8 using the above techniques.
5. Implementation

This section briefly describes our current status of CoSMoS implementation. To implement CoSMoS, we designed a description language (framework) for CoSMoS, which is called CSDF (Component Service Description Framework). CSDF is used to help implement a component, and also is used as the format of an advertisement (metadata) of a component. We designed two variations of CSDF: CSDF/XML and CSDF/Javadoc. CSDF/XML uses RDF Schema to describe the semantic layer, and WSDL-like notation to describe the functional layer. CSDF/Javadoc uses RDF Schema embedded in Javadoc to describe the semantic layer, and Java source code to describe the functional layer. Figure 11 shows an example of CSDF/XML. The complete grammar of CSDF and a CSDF compiler are available at [12].

We also implemented a runtime system that supports the service composition algorithm described in Section 4. Components generated through CSDF compiler can be deployed onto the runtime system, and users can compose applications by providing composition requests to the runtime system. As we finalize our algorithm design and implement the completed algorithm onto our runtime system, we will make the runtime system public.

6. Conclusion

This paper proposed a new component model called CoSMoS. CoSMoS achieves extensibility and understandability by applying semantic graphs and ontologies. CoSMoS also solves operation dependency and soft-state problems by applying an object-oriented model. This paper depicted the necessity of these features by discussing existing dynamic service composition systems. This paper also presented the detailed design of CoSMoS, and a dynamic service composition algorithm using CoSMoS. This paper briefly introduced a description framework and a runtime system that were designed and developed for CoSMoS.

Currently we are evaluating the features and issues of CoSMoS both theoretically and empirically. We are also developing a fully distributed service composition mechanism based on the algorithm described in Section 4.

7. References


Figure 11: Example of CSDF/XML