Ontology Support for Managing Top-Down Changes in Composite Services*

Xumin Liu[1] and Athman Bouguettaya[2]

[1] Department of Computer Science, Virginia Tech, USA
xuminl@vt.edu
[2] CSIRO ICT Center, Canberra, ACT, Australia
Athman.Bouguettaya@csiro.au

Abstract. We present a foundational framework to manage changes in composite services. The framework takes as input a change specification and reacts to the change in an automatic and efficient manner. We propose a service ontology that provides systematic support for the change management process. We also propose a set of algorithms that enable us to efficiently query the proposed service ontology. With the ontology support, desired service functionalities can be accurately, efficiently retrieved and composed to react to changes. We use a Service-Oriented Enterprise (SOE) as an application of composite services to motivate and illustrate the proposed solution. We evaluate the performance of the proposed algorithms with a set of experiments.

Keywords: top-down changes, change management, composite service, ontology, service oriented enterprises.

1 Introduction

The emerging service oriented computing and the enabling technologies facilitate efficient functionality outsourcing on the Web. This is enabling a paradigm shift in business structures allowing them to outsource required functionality from third party Web-based providers through service composition [3]. A *composite Web service* is therefore an on-demand and dynamic collaboration between autonomous Web services that collectively provide a value added service. Each autonomous service specializes in a core competency, which reduces cost with increased quality and efficiency for the business entity and its consumers. While there has been a large body of research in the automatic composition of Web services, managing the changes during the lifecycle of composite Web services has so far attracted little attention [3,19,4].

We use a *Service Oriented Enterprise* (SOE) as a typical application of composite services to motivate and illustrate our work. An SOE is a Web-based Virtual Enterprises [11]. It outsources the functionalities from autonomous Web services, whose providers may be geographically distributed and organizationally

* This work was supported by the National Science Foundation under the CNS - Cyber Trust program with contract 0627469.

E. Bertino and J.B.D. Joshi (Eds.): CollaborateCom 2008, LNICST 10, pp. 760–777, 2009.

independent. It is expected to promote entrepreneurship and introduce new business opportunities through dynamic alliances.

Example 1.1. We use an application from the travel domain as a running example throughout this paper. Consider a travel SOE that aims to provide a comprehensive travel package by outsourcing functionalities from different service providers, including Airline services, Hotel services, and Car rental services. Suppose that a new market report shows that Point of Interest (POI) services are very popular recently. A POI service is expected to retrieve the local attractions based on user interests given a geographical location. Using this service, a traveler can very easily get the information, like restaurants, museums, music centers, around the hotel he/she chooses to stay in during the trip. Therefore, the owner of a travel agency SOE, say John, wants to add a new POI service into the travel package to attract more market interests. To react to this change, a POI service needs to be added and composed with other services. Meanwhile, suppose that another market report shows that users put more attention to the service's reputation when they choose a travel package. In this case, John wants to ensure that all the outsourced service providers have a high reputation. □

Fully realizing SOEs lies in providing support to improve their adaptability to the dynamic environment, i.e., to deal with changes during the life-time of an SOE as rules and not as exceptions [2,11]. For instance, market conditions may change, business regulations may evolve, individual Web services may come and go at will, or new technologies may emerge over time. These all may trigger a change in an SOE with respect to the functionality it provides, the way it works, the partners it is composed of, and the performance it offers. Unlike traditional enterprises, where changes are "exceptions", in SOEs changes are likely the norm. Therefore, a systematic solution for handling changes is a fundamental issue in SOEs.

Changes in SOEs can be classified into two categories: *top-down changes* and *bottom-up changes* [12]. Top-down changes refer to those that are initiated by an SOE owner. These are usually the result of new business requirements, new regulations, or new laws. For example, the owner of a travel agency SOE may want to add a taxi service to the travel package. Bottom-up changes refer to those that are initiated by the outsourced Web service providers. For example, an airline reservation service provider may change the functionality of the service by adding a new operation for checking a flight status, or a traffic service provider may decide to increase the invocation fee of the service. In this paper, we focus on dealing with top-down changes.

Change management in the context of composite services poses a set of research issues. A composite service outsources its functionality from independent service providers. There are no central control mechanisms that can be used to monitor and manage these service providers. Therefore, the challenge of managing changes lies in providing an end-to-end framework to introduce, model, and manage a top-down change in a way that best reacts to the change.

Among the most challenging issues is the automation of the process of change reaction. We expect that changes in an SOE occurs frequently due to the dynamic

business environment it interacts with. Thus, it is challenging to manage all changes in a manual way. There are existing frameworks proposed for managing changes in other fields, such as software systems, database systems, and workflow systems [8,14,16]. In these frameworks, automating change reaction mainly relies on predefined schemas or policies, whose availability cannot be guaranteed in an SOE.

Therefore, it is not sufficient to manage changes in an SOE by simply applying the approaches adopted in existing frameworks.

In this paper, we leverage the machine-processable semantics delivered by a Web service ontology to support automatic change management in an SOE. Web service ontologies have been proposed to semantically enrich the description of Web services, such as their functionality, invocation, quality, etc [18,5]. They also capture the semantics of the interactions between different communities of Web services. The semantics enable software agents to automatically locate, access, and compose Web services without human interference. Therefore, we expect that the semantic support will also play a key role in the automatic change management process. We assume that the development, agreement, and management of ontologies can be achieved through the existing ontological supporting tools [9].

We summarize our major contributions as follows.

- First, we propose an integrated change management framework that enables an SOE to systematically react to a top-down change. The framework takes a change specification as input and reacts to the change by modifying the composition of the member services of an SOE.
- Second, we enrich the change management framework with ontology support for automatically reacting to the changes. This has the effect of transforming a change specification into a corresponding service ontology query. By answering the query, the desired functionality related to the change can be efficiently retrieved from the tree-structured service ontology.

The reminder of this paper is organized as follows. In Section 2, we introduce three-layer top-down changes based on an SOE's architecture. In Section 3, we propose an ontology-based framework that manages top-down changes in an SOE. In Section 4, we define a service ontology with a tree-like structure to provide the sufficient semantics for change management. In Section 5, we propose a set of algorithms to efficiently query the service ontology. We report our experimental results in Section 6. We briefly overview some related work in Section 7 and conclude in Section 8.

2 Preliminary

In this section, we briefly introduce a supporting infrastructure of an SOE by identifying its key components. Based on this infrastructure, we then describe a layered top-down changes that might occur to an SOE. Top-down changes are initiated at the top components and propagated to the lower ones.

2.1 A Supporting Infrastructure of an SOE

There are two key components and two supporting components in an SOE infrastructure. The key components include an *SOE schema* and an *SOE instance*. An SOE is associated with an SOE schema, which describes its high-level business logic. An SOE schema consists of a set of abstract services and the relationships among these services. An abstract service specifies one type of functionality provided by Web services. It is not bounded to any concrete service. It is defined in terms of a Web service ontology. An SOE instance is an orchestration of a set of concrete services, which instantiates an SOE schema. It actually delivers the functionality and performance of an SOE. The two supporting components include *ontology providers* and *Web service providers*. The ontology provider manages and maintains a set of ontologies that semantically describe Web services. An SOE outsources ontologies from an ontology provider to build up its schema. The Web service providers offer a set of Web services, which can be outsourced to form SOE instances.

The underpinning of the proposed supporting infrastructure is a standard *Service Oriented Architecture* (SOA) [7]. The service providers use WSDL to describe their services. Web service registries, such as UDDI, can be used as a directory for an SOE to look for Web services. After locating a Web service, SOAP messages are exchanged between an SOE and the service providers for invoking the service. Beyond this, semantic Web service technologies can be used by the ontology providers to define their service ontology, such as OWL-S and WSMO [5,18]. The composition between selected services can be defined using service orchestration language, such as BPEL [10].

2.2 The Change Layers

Changes in an SOE can be categorized into three layers: *business requirement changes*, *SOE schema changes*, and *SOE instance changes*. The uppermost layer reflects the dynamic environment that an SOE is exposed to. An SOE schema gives a high-level abstraction of an SOE's functionality and invocation. An SOE instance consists of a set of concrete Web services, which are composed together to instantiate an SOE's schema. A business requirement change could be led by new technologies, new business strategies, new market requirements, or new regulations and laws. In our running example, the owner of the travel agency SOE wants to add a POI service to the travel package. The business requirement change can be interpreted as the modification of SOE's functionality, invocation, (i.e., an SOE schema change) and the updated performance requirement (i.e., an SOE instance change). That is, the travel agency SOE needs to add a POI functionality. At the same time, the invocation among the member services (i.e., the airline service, the hotel service, and the taxi service) needs to be modified, respectively. An SOE schema change will be propagated to the SOE instance changes, to implement the changes in practice. An SOE instance change can be specified as the modification of the list of the concrete services and the way they cooperate.

3 Ontology-Based Framework for Change Management

In this section, we propose an ontology-based framework to support automatic change management. We first propose a formal model that captures the key features of top-down changes. Based on this model, we then propose our framework (depicted in Figure 1). The framework consists of two major components: a *change manager* and an *ontology manager*. The change manager is used for managing changes in an SOE. It takes a change specification as input and generates a new SOE schema and instance as output. The ontology manager is used for managing ontologies to provide semantic support for the change manager. It also provides an interface to query the semantics.

3.1 Top-Down Change Model

Generally, the process of change reaction is modifying an SOE's functionality and/or performance to fulfill the requirement introduced by the change. Therefore, we model a change by specifying its functional and non-functional (i.e., performance) requirement.

Definition 3.1. *A top-down change $\mathcal{C}$ is a binary $\{\mathcal{C}_\mathcal{F}, \mathcal{C}_\mathcal{P}\}$, where $\mathcal{C}_\mathcal{F}$ is the functional requirement enforced by introducing the change (referred to as a* functional change*), and $\mathcal{C}_\mathcal{P}$ is the performance requirement enforced by introducing the change (referred to as a* non-functional change*).* □

In the above definition, $\mathcal{C}_\mathcal{F}$ specifies the requirement on the modification of an SOE's functionality enforced by introducing $\mathcal{C}$, such as adding a new functionality and/or removing an existing functionality. In our running example, $\mathcal{C}_\mathcal{F}$ refers to that an SOE should add a POI service.

To fulfill the requirement defined in $\mathcal{C}_\mathcal{F}$, the first step is to change the SOE's schema, which defines the SOE's functionality. Therefore, the change analyzer

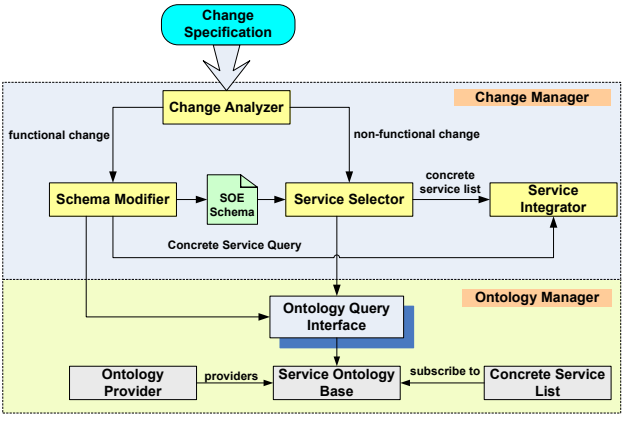

Fig. 1. An Ontology-based Framework For Change Management in SOEs

needs to specify $\mathcal{C}_{\mathcal{F}}$ in the same way in defining an SOE's functionality. Since an SOE outsources its functionality from one or more Web services, it is natural to use the combination of a set of abstract services to define an SOE's functionality [12]. Each abstract service defines a type of functionality, such as transportation, lodge, information, etc. Therefore, we have the formal definition of $\mathcal{C}_{\mathcal{F}}$ as follows.

Definition 3.2. *A functional change $\mathcal{C}_{\mathcal{F}}$ is a binary $\{\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{D}}\}$, where $\mathcal{S}_{\mathcal{A}}$ is the set of abstract services that define the functionality required to be added to an SOE, and $\mathcal{S}_{\mathcal{D}}$ is the set of abstract services that define the functionality required to be removed from an SOE.* □

$\mathcal{C}_{\mathcal{P}}$ specifies the requirement on the modification of an SOE's performance enforced by introducing $\mathcal{C}$, such as improving the SOE's reliability, reducing the cost, reducing the invocation duration, increasing the service provider's reputation, etc. In our running example, $\mathcal{C}_{\mathcal{P}}$ refers to that all the outsourced service providers should have a reputation with the degree of high.

$\mathcal{C}_{\mathcal{P}}$ can be defined in terms of a set of quality constraints. These constraints can be enforced on a single outsourced service or the SOE. Examples of such constraints include: *the charge of the* Lodge *service should be less then 80 dollars per night*; *the reputation of each service provider in the SOE should be high*; *the overall reliability of invoking the SOE should be high*. We have the formal definition of $\mathcal{C}_{\mathcal{P}}$ as follows.

Definition 3.3. *A functionality change $\mathcal{C}_{\mathcal{P}}$ is a set $\{\gamma_1, ..., \gamma_i, ..., \gamma_n\}$, where each γ_i is a quality constraint and defined as a triple $\{p_i, c_i, S_i\}$. p_i is a quality parameter, such as reputation, reliability, cost, etc; c_i is a conditional formula, such as "$<$ \$80", "$=$ high", etc. S_i is a set of abstract services whose performances are enforced by γ_i. Since a quality constraint can be enforced on either single service or an SOE, we use $\mathcal{E}$ to represent the SOE. Therefore, if $S_i = \mathcal{E}$, it means that γ_i is enforced on the entire SOE.* □

Example 3.1. In our travel SOE example, the functional change can be specified as $\mathcal{C}_{\mathcal{F}}$={ {POI}, {}} based on Definition 3.2. Similarly, based on Definition 3.3, the non-functional change can be specified as $\mathcal{C}_{\mathcal{P}}$={$\gamma_1$}, where γ_1={"reputation", "=high", {Airline, Hotel, Car rental, POI} }. □

3.2 Change Manager

The change manager consists of a set of components, including a *change analyzer*, a *schema modifier*, a *service selector*, and a *service integrator*. The change analyzer takes as input a change specification in the format that can be understood and processed by other components. A change specification conveys the information about the requirement on modifying an SOE's functionality and/or performance, which are determined by the SOE's schema and the outsourced Web services respectively. Therefore, the schema modifier may need to update the SOE's schema to fulfill the functional requirement of the change. The service selector may need to locate Web services to fulfill both the functional and performance requirement

of the change. Finally, the service integrator may compose the outsourced Web services to generate the new SOE instance.

A **change analyzer** provides a user interface that takes as input a top-down change specification. The information contained in a change specification is used for reacting to the change, including functional requirement ($\mathcal{C}_{\mathcal{F}}$) and performance requirement ($\mathcal{C}_{\mathcal{P}}$) of a change. $\mathcal{C}_{\mathcal{F}}$ will be used as the input of the schema modifier to update the functionality of an SOE. $\mathcal{C}_{\mathcal{P}}$ will be used as the input of the service selector to locate the service that delivers the desirable quality.

The **schema modifier** changes an SOE's schema to fulfill the requirement specified in $\mathcal{C}_{\mathcal{F}}$. A set of SOE schema templates can be predefined and stored in a domain-specific knowledge base to facilitate the schema updating process. Thus, once there is a requirement on changing an SOE's schema, a schema modifier first searches the knowledge base for a predefined SOE schema that matches the requirement. If there is not such a match, the schema modifier will automatically generate a new semantically correct service schema based on $\mathcal{C}_{\mathcal{F}}$ to compose different services [13].

The **service selector** locates Web services to generate an instantiation of the new SOE's schema. It takes as the input of the SOE's new schema and $\mathcal{C}_{\mathcal{P}}$ to guarantee that the newly generated SOE's instance meets both the functional and the performance requirement of $\mathcal{C}$. Specifically, a service selector follows two steps: *functionality-based Web service discovery* and *quality-based service selection*. The first step is to find Web services that provide the functionality specified in the new SOE's schema. It requires a functionality-based service registry to achieve this purpose. Since there might be competing providers that offer the similar functionality, the service selector may get multiple services. Thus, it needs to select a service based on the quality requirement. The service selector first removes the services that do not meet $\mathcal{C}_{\mathcal{P}}$. It then chooses the service with the best quality. The output of the service selector is a list of Web services (referred to as $\mathcal{CS}$) whose composition is expected to meet both $\mathcal{C}_{\mathcal{F}}$ and $\mathcal{C}_{\mathcal{P}}$.

The **service integrator** generates a new SOE's instance by composing the services in $\mathcal{S}$. It takes as input the service list $\mathcal{CS}$ and the updated SOE's schema. It specifies the execution order and data flow among the services in $\mathcal{S}$ that conforms to the cooperation patterns defined by the updated SOE schema. It also coordinates the interaction between different services. Some existing Web service standards such as WS-Coordination, BPEL, etc, can be leveraged to implement the service integrator [15,10].

3.3 The Ontology Manager

The ontology support components include a *service ontology base* and an *ontology query interface*.

The **service ontology base** stores the ontology definitions of Web services within a specific domain. A node in a service ontology defines a type of functionality offered by a service. Examples of the nodes in a travel domain include Airline, Taxi, and Hotel. By the nature of ontology, Web services can be classified into categories based on their functionalities. Therefore, each node in a service

ontology is associated with a list of services that provide the defined functionality. This association enables a service selector to perform functionality-based service discovery by first locating the corresponding node in a service ontology, then locating the Web services that subscribe to the node. Beside of functionality, a service ontology also models the relationship between different Web services, which can be used to guide their composition.

The **ontology query interface** supports two types of queries in the ontology base: *functionality query* and *Web service query*. The functionality query is to locate a node in the service ontology and retrieve the related information, such as its relationships with other nodes. It can be performed in the following ways: (1) operation-based query, (2) data-based query, and (3) the combination of (1) and (2). An operation-based query is to traverse the service ontology and retrieve the related information about the nodes which provide the operation. A data-based query is to traverse the ontology and retrieve the related information about the nodes which provides the matched input and output. The Web service query is to find a list of Web services that provide a specific functionality, which is identified by a certain node in a service ontology. The corresponding Web services can be retrieved by checking whether they are subscribing to the node.

Ontology support components are central for automatic change management process. We will elaborate them in the following sections.

4 Web Service Ontology

In this section, we propose a *Web service ontology* that provides semantic support for automatic change management. We first identify a set of key semantics that are described by the service ontology. We then define the structure of the service ontology which will be used as the basis for ontology querying.

4.1 Ontology Definition

The semantics provided by the service ontology aims to help automatically generate a new SOE's schema. Generating the new schema requires two steps. First, it needs to identify the functionality being added to or removed from an SOE. Second, it needs to gracefully compose the newly updated service list. To achieve this, a service ontology needs to capture two types of semantics: *service functionality* (which helps achieve the first step), and *service dependency* (which helps achieve the second step).

The functionality of a service can be modeled from two aspects: the operations that a service provides (i.e., *service operation*) and the data that a service operates on (i.e., *service data*), which also corresponds the two type of functionality query we proposed in Section 3.3.

A service functionality is collectively delivered by a set of operations. The process of accessing a service is actually invoking one or more operations provided by the service. The operations consume the service input and generate the output of the service. It is worthy to note that there may be dependent relationships

between different service operations. For example, an Airline service may provide several operations, such as `user_login`, `airline_reservation`, etc. Typically, an user needs to login before (s)he can reserve an air ticket. Therefore, there is a dependency between `user_login` and `airline_reservation`. The dependencies between service operations (referred to as *operation-level dependencies*) need to be strictly enforced when accessing a service.

A service data is also an essential aspect of a service functionality. A service is affected by the outside with a set of input and responses with a set of output [1]. From the external perspective, a service data therefore consists of two data sets: *input* ($\mathcal{I}$) and *output* ($\mathcal{O}$) data.

A service ontology also needs to capture the dependent relationships defined in term of composite services. A Web service is independent and autonomous in nature. A user can directly access a Web service without relying on other services. However, when multiple services are composed together by an SOE, certain dependency constraints can be defined within the generated composite service. For example, the Hotel service usually depends on the Airline service when they are both included in a travel package since the city and the check in-and-out dates of the Hotel service are usually determined by the flight information. As a result, the invocation of the Hotel service should be performed after the Airline service is invoked.

4.2 Ontology Structure

The structure of ontology is hierarchical and extensible by nature. Each node in this structure corresponds to a type of service functionality. Once the ontology structure becomes large with the increment of the available service functionalities, the process of identifying a proper piece of functionality would turn out to be time consuming. It is of importance and beneficial for change management to make this process efficient and accurate. An intuitive way is to leverage the relationship between a node and its children to guide the search of the ontology. We identify two types of *parent-child* relationships in a service ontology: *Is-a* and *Has-Of*, as depicted in Figure 2.

An *Is-a* relationship lies in between a node and its parent node if the node is one type of the parent node. That is, the child node has all the properties (i.e., operations and service data) of its parent node. Beside, it may also has the properties that the parent node does not have. For example, in Figure 2, an Airline service is a child of a Transportation service. It provides the transportation service with a special feature, i.e., through a flight. Therefore, there is an *Is-a* relationship between the Airline service and the Transportation service, shown by a line between them.

A *Has-of* relationship lies in between a node and its parent node if the node is one part of the parent node. That is, the child node has part of the properties of its parent node. For example, in Figure 2, a Flight Quote service is a child of an Airline service. It only provides the service of getting the quote of a flight, but not other airline-related services, such as checking flight status, reserving a flight,

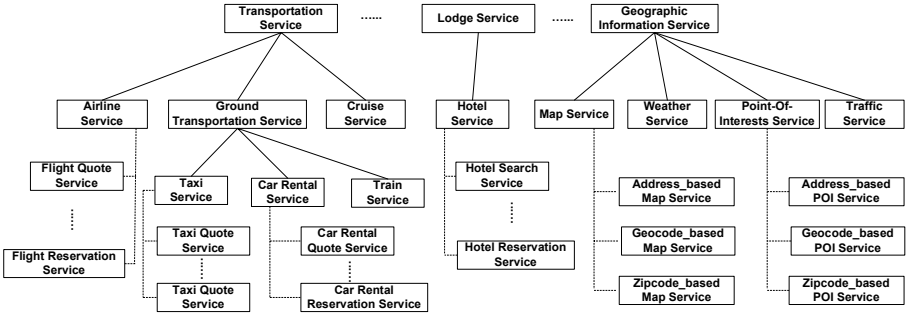

Fig. 2. A Travel Domain Ontology

electronic check in, etc. Therefore, there is a *Has-of* relationship between the Flight Quote service and the Airline service, shown by a dashed line between them.

5 Service Ontology Query Infrastructure

As stated in Section 3, reacting to a change requires to refer back to the hierarchical structure of the corresponding service ontology. As the ontology structure may become very large due to the increase of the services in the domain, there is a need to efficiently query the service ontology to retrieve the required service nodes and locate Web services that subscribe to them at the same time. Considering the tree-like structure of the service ontology we proposed in Section 4, we leverage *path expressions* as an effective tool to declaratively and efficiently query the service ontology [17].

To use path expression in the service query, we add a root node to our ontology to make it have a well-formed tree structure. The root node is an abstract one that does not have any properties. For any service node that does not have a parent node will take the root node as its parent. An *Is-a* relationships lies in between the root node and its child nodes. By using the abstract node, it is guaranteed that the query can be specified and performed from the root of the ontology tree.

5.1 Processing the Service Ontology Queries

We present algorithms to process the queries on the service ontology for the purpose of change management. As discussed in Section 3, there are mainly two types of ontology queries needed to be performed: Web service query and functionality query.

The Web service query is required when a service selector needs to find a list of Web services that provide the specific functionality. In this case, the query process takes a service node as a input and return the list of Web services that subscribe to it. It can be easily performed through the subscription between the ontology base and the Web service list in our proposed framework. Therefore, we will focus on functionality query in this section.

The functionality query is required under two situations. First, when an SOE's owner wants to add a new functionality to the SOE, the corresponding node and the related operation in the service ontology need to be retrieved. Meanwhile, the information about both operation-level and service-level dependencies also needs to be retrieved for automatically generating the new SOE's schema. In this case, an operation-based functionality query should be performed. Second, the composition of the new member services may need to outsource another functionality which is not specified in a change requirement. This will happen when the completeness of the data flow is violated by adding new services or removing existing ones. New services need to be added to fill the blank [12]. In this case, data-based functionality query should be performed. We will present algorithms for these two types of functionality queries as follows.

Operation-based Functionality Query. In an operation-based functionality query, the tree-like structure of an ontology is traversed to find the node that matches the given functionality. The query is specified in terms of a path expression. It returns the required service node and the operation as well as both the service-level and operation-level dependencies if there is any.

As depicted in Algorithm 1., the input is a service ontology tree and a path expression, which is specified in terms of a string. It first extracts the elements from the path expression, such as the path variables($\mathcal{C}$), the service nodes($\mathcal{S}$), and the operation($\mathcal{OP}$). It then takes different steps for different path variables. If the path variable is '/', the algorithm leverages a simple search procedure (line 7-17), which only looks up the immediate children of the current node being processed. On the other hand, if the path variable is '//', the algorithm leverages a heuristic breath first search procedure (line 18-25 and line 32-38), which only searches the child nodes that potentially have the desired operation. For example, if the parent node does not provide the targeted operation, its *Has_of* children will not provide it either. In this case, the algorithms will not explore these children. This will greatly improve the performance of the search process. When the algorithm hits a path variable of '[', it gets the target operation. Then the information of the node and the operation will be retrieved (line 26-29).

Data-based Functionality Query. The data-based functionality query takes as input the desired service data (i.e., input and output). It then traverses the ontology tree to locate the service node that provides the specified service data. Instead of exhaustively going through the entire ontology tree, the algorithm takes advantage of the two types of relationships between a node and its children to effectively narrow down the searching scope.

As shown in Algorithm 2., we use a recursive procedure to query the tree-like structure of a service ontology. The data-based functionality query is performed by matchmaking between two sets of data: service data (including S_I and S_O) of the node in an ontology and the required data (including D_I and D_O) given by the schema modifier. The matching criteria can be defined as: a node is matched if it's output covers the required output, i.e., $S_O \supseteq D_O$, and it's input can be covered by the given input, i.e., $S_I \subseteq D_I$.

Algorithm 1. Operation-based Service Functionality Query

Require: a service ontology query $\mathcal{Q}$ (a path expression); a service ontology tree $T(r)$
Ensure: a service node N_S; N_S's depending service nodes L_S; an operation op; op's depending operationL_{op}

```
1: C = Q.C; S = Q.S; OP = Q.OP;
2: N = r;
3: while C ≠ φ do
4:    c = C.pop();
5:    if S ≠ φ then
6:       s = S.pop();
7:       if c=='/' then
8:          find=false,
9:          for all n ∈ N do
10:            if n.name matches s then
11:               N = n; find=true;
12:            end if
13:         end for
14:         if find==false then
15:            return ERROR; {Fail to find the specified service}
16:         end if
17:      end if
18:      if c=='//' then
19:         for all n ∈ N do
20:            N=HBFS(s, OP, T(n)); {Heuristically breath first search s in subtree T(n)}
21:            if N=null then
22:               return ERROR; {Fail to find the specified service}
23:            end if
24:         end for
25:      end if
26:      if c='[' then
27:         N_S = N; L_S=N.getDependingService();
28:         op = N.getOperation(OP);
29:         L_op=N.getDependingOperation(op);
30:      end if
31:   end if
32: end while
33: Function HBFS(s, OP, T(n))
34: N=get_Children(n);
35: if OP ∉ n.OP then
36:    N=get_Is_a_Children(n);
37: end if
38: for all t ∈ N do
39:    if t.name matches s then
40:       return t; {Find the service}
41:       Else HBFS(t, OP, T(t))
42:    end if
43: end for
44: return NULL;
```

The query starts at the root node. It first checks whether the current node matches the requirement. If so, it will return the current node as the result (Line 3-9). If not, there will be two cases. First, the current node requires more input than the specified one. In this case, the child nodes that follow an *Is-a* relationship will be pruned since they require no less input than the current node (Line 10-17). Second, the current node does not fully provide the specified output. In this case, the child nodes that follow a *Has-of* relationship will be pruned since they provide no more output than the current node (Line 18-25). By leveraging the two types of relationships to guide the search, the algorithm performs more efficient by only checking the potential nodes that provide the specified service data.

Algorithm 2. Data-based Service Functionality Query

Require: Required input $\mathcal{D}_I$; Required output $\mathcal{D}_O$, a service ontology subtree $\mathcal{T}(r)$
Ensure: a service node $\mathcal{S}$; the related operation list L_{OP}

```
1: Function CHECK(s,D_I,D_O, T(s))
2: if s.Input ⊆ D_I then
3:    if s.Output ⊇ D_O then
4:       S = s; {Find the matched service node}
5:       L_OP=s.get_Operation_By_Output(D_O)
6:       return L_OP;
7:    end if
8: end if
9: if (s.Input ⊆ D_I)==true then
10:    if (s.Output ⊇ D_O)= false then
11:       C_L=get_Is_a_Children(s);
12:       for all s' ∈ C_L do
13:          return CHECK(r, D_I,D_O,T(r)});
14:       end for
15:    end if
16: end if
17: if (s.Input ⊆ D_I)==false then
18:    if (s.Output ⊇ D_O)= true then
19:       C_L=get_Has_of_Children(s);
20:       for all s' ∈ C_L do
21:          return CHECK(s', D_I,D_O,T(s)})
22:       end for
23:    end if
24: end if
25: return ERROR; {Fail to find a matched service node}
```

5.2 Automatic SOE Schema Modification

In this section, we propose an integrated process to automatically modify an SOE's schema. The updated schema is guaranteed to fulfill $\mathcal{C}_\mathcal{F}$. This process is essentially enabled by the proposed service ontology and the corresponding query support.

As depicted in Figure 3, the process starts from generating the new functionality list, taking $\mathcal{C}_\mathcal{F}$ as its input. The functionality that is expected to add to an SOE is specified in terms of a path expression.

In the second step, it performs an operation-based service ontology query (i.e., algorithm 1.) to retrieve the related service node and the operations for each

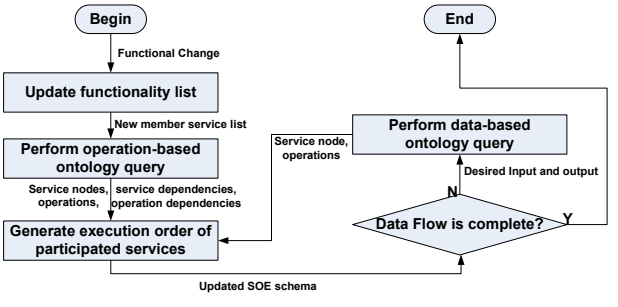

Fig. 3. The Diagram of Modifying an SOE's Schema

path expression. The query result will be used for service selection. Meanwhile, the related service dependencies and operation dependencies are also retrieved. These dependencies will be enforced when composing different services together.

In the third step, the invocation order of the member services and their operations are generated based on their dependencies. Specifically, it first follows the service-level dependencies to generate a service-level order. For example, if invoking a service A depends on the invocation of service B, A will be invoked after the invocation of B. It then follows the operation-level dependencies to generate an operation-level order. As a result, the member services are compose together, which actually defines the new SOE's schema.

In the fourth step, it checks whether the data flow of the updated schema is complete. If not, it performs a data-based query on the service ontology to find the service node which can fill the blank of the data flow (refers to algorithm 2. for details). The returned service nodes are then added to the SOE and composed with other services (i.e., by taking the third step). If the data flow is complete, the process terminates with the output of a new SOE's schema.

6 Experiments

We conducted a set of experiments to assess the performance of the proposed service ontology query algorithms. We run our experiments on a cluster of Sun Enterprise Ultra 10 workstations under Solaris operating system. In order to evaluate the query efficiency, we need to first build a complete service ontology, referred to as $\mathcal{O}$, upon which the query can be applied. We describe the key parameters and illustrate how each of these parameters are used to construct the service ontology. Table 1 shows the definitions of the parameters and their values.

6.1 Constructing the Service Ontology

We define two key parameters to determine the size of $\mathcal{O}$: depth d and total number of service nodes n. We will evaluate the effect of both d and n on the query efficiency. We construct the service ontology level by level. The construction starts from the root, which is a dummy node with an id of 1, representing the entry point of the ontology. The fanout of each node is a randomly generated number with an upper bound of f. For instance, if the fanout is 3, the root node will have three child nodes, whose ids are 11, 12, and 13. The parent-children

Table 1. Parameter Settings

Parameter	Meaning	Values
d	Ontology depth	[6, 12]
n	Total nodes	$[10^3, 10^6]$
f	Node fanout	[5, 10]
k_1	Number of new operations	[1, 5]
k_2	Number of inherited operations	[1, 4]

relationship has two types: t_1 =*is-a* and t_2 =*has-of*. We randomly assign t_1 or t_2 between a child node and its parent. If a child node *cn* holds an *is-a* relationship with its parent *pn*, *cn* will inherent all the operations from *pn*. In addition, k_1 randomly generated new operations will also be assigned to *cn*. If *cn* holds an *has-of* relationship with *pn*, k_2 operations will be randomly selected from the operation set of *pn* and assigned to *cn*. We assign no operations for the root node since it is just an entry point of the service ontology.

We leverage a FIFO queue $\mathcal{Q}$ to facilitate the process of building the service ontology $\mathcal{O}$. We start by generating the root node and inserting it into $\mathcal{Q}$. The root node is then extracted from $\mathcal{Q}$. All its child nodes are generated based on the rationale we described above. These child nodes are then inserted into $\mathcal{Q}$. The node generation stops when the depth or the maximum number of nodes are reached. After that, we continue to extract the node from the queue until it becomes empty.

6.2 Performance Study

We study the performance of the service ontology query algorithm (referred to as OntoQuery) in this section. We also implemented a Depth First Search (referred to as DFS) on the service ontology for comparison purpose. By performance, we report both the node accesses (referred to as *NA*), which is independent of hardware settings, and the actual running time on our experiment machines. We run our experiments on a cluster of Sun Enterprise Ultra 10 workstation with 512 Mbytes Ram under Solaris operating system.

Depth of the Service Ontology. We study the effect of the depth of the service ontology in this section. We keep the maximum fanout as 5, i.e., $f = 5$, and vary the depth from 6 to 12. Figure 4 shows how the number of node accesses varies with the depth of the service ontology. OntoQuery accesses much less number of nodes than DFS. The smallest difference is almost two orders of magnitude. Generally, DFS accesses more nodes as the depth of the service ontology increases. This increase is in line with the increase in the size of the service ontology (in terms of the total nodes). It is worth to note that the size of the created service ontology does not necessarily increase with its depth. This is because that we only specify the upper bound of the fanout of each node and the actual fanout of most nodes in a deeper ontology may be smaller than those in a shallower ontology. The number of node accesses does not necessarily increase with the depth, either. This is because OntoQuery only picks either *is-a* or *has-of* to proceed. Since these two relationships are randomly generated, they may not necessarily increase with the depth. This also accounts for the larger performance difference when the depth increases. Figure 5 shows the actual CPU time, which demonstrates a very similar trends as the number of node accesses.

Fanout of the Service Nodes. We investigate the effect of the maximum node fanout f in this section. We keep the depth of the ontology as 6, i.e., $d = 6$, and vary the maximum fanout from 6 to 15. Figure 6 and 7 show the number of

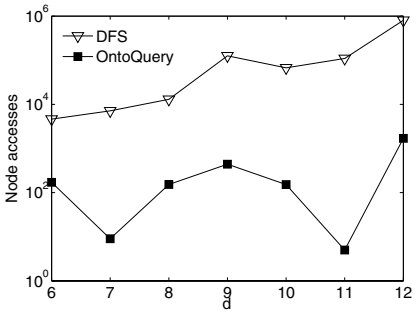

Fig. 4. NA Vs. d

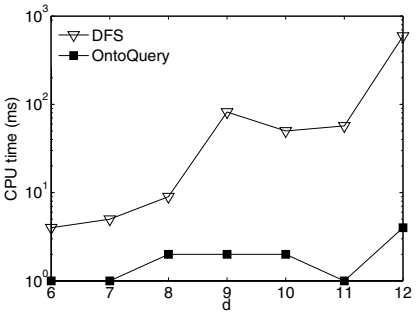

Fig. 5. Time Vs. d

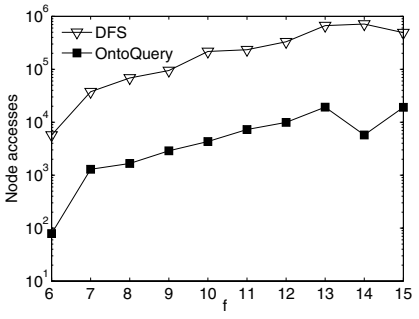

Fig. 6. NA Vs. f

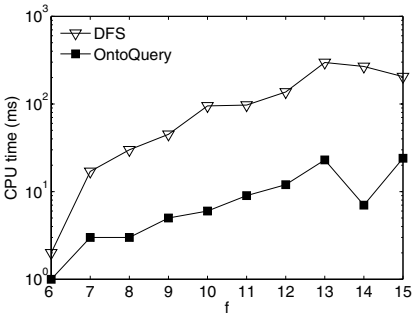

Fig. 7. Time Vs. f

node accesses and the CPU time, respectively. The results are fairly consistent with those from Section 6.2. The results also further confirm the efficiency of the proposed algorithm.

7 Related Work

Change management is an active research topic in database management, knowledge engineering, and software evolution. Research efforts are also underway to provide change management in a Web service community and adaptive workflow systems [2,6]. In this section, we will elaborate some representative works and differentiate them with our work.

In [2], it focuses on managing bottom-up changes in service-oriented enterprises. Changes are distinguished between service level and business level: *triggering* changes that occurs at the service level and *reactive* changes that occur at the business level in response to the triggering changes. A set of mapping rules are defined between triggering changes and reactive changes. These rules are used for propagating changes. A petri-net based change model is proposed as a mechanism for automatically reacting changes. Agents are employed to assist in detecting and managing changes to the enterprises. [2] mainly focus on devising

handling mechanisms for exceptional changes. An example of such mechanisms is that the system will switch to use an alternative service if a sudden failure occurs to a service. We focus on the top down changes, which are initiated by an SOE's owner in case of the occurrence of new business requirements or new business regulations.

In [6], it focuses on modeling dynamic changes within workflow systems. It introduces a Modeling Language to support Dynamic Evolution within Workflow System (ML-DEWS). A change is modeled as a process class, which contains the information of *roll-out time*, *expiration time*, *change filter*, and *migration process*. The roll-out time indicates when the change begins. The expiration time indicates when the change ends. The change filter specifies the old cases that are allowed to migrate to the new procedure. The migration process specifies how the filtered-in old cases migrate to the new process. In [6], the new version of the workflow schema is predefined. In our work, the new SOE schema is automatically generated. We also propose mechanisms to efficiently select Web services to instantiate the new schema.

8 Conclusion

We presented an ontology-based framework that enables an SOE to efficiently adapt to top-down changes. The proposed service ontology provides sufficient semantic support for automatically updating an SOE's schema when reacting to a change. The tree-like service ontology structure defines two types of parent-child relationships: Is-a and Has-of. The functionalities relevant to a change can be efficiently and accurately identified by following these two types of relationships in the ontology tree. Our experimental results demonstrated the efficiency of the proposed service ontology query algorithms.

References

1. Abiteboul, S., Vianu, V., Fordham, B., Yesha, Y.: Relational transducers for electronic commerce. In: PODS 1998, pp. 179–187. ACM Press, New York (1998)
2. Akram, M.S., Medjahed, B., Bouguettaya, A.: Supporting Dynamic Changes in Web Service Environments. In: First International Conference on Service Oriented Computing, Trento, Italy, pp. 319–334 (December 2003)
3. Baghdadi, Y.: A Web services-based business interactions manager to support electronic commerce applications. In: ICEC 2005: Proceedings of the 7th international conference on Electronic commerce, pp. 435–445. ACM Press, New York (2005)
4. Casati, F., Shan, E., Dayal, U., Shan, M.-C.: Business-Oriented Management of Web Services. ACM Communications (October 2003)
5. Coalition, T.O.S.: Owl-s: Semantic markup for web services. Technical report (July 2004), `http://www.daml.org/services/owl-s/1.1B/owl-s/owl-s.html`
6. Ellis, C.A., Keddara, K.: A workflow change is a workflow. In: Business Process Management, Models, Techniques, and Empirical Studies, London, UK, pp. 201–217. Springer, Heidelberg (2000)

7. Erl, T.: Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. Prentice Hall PTR, Upper Saddle River (2004)
8. Francisco-Revilla, L., Frank Shipman III, M.S., Furuta, R., Karadkar, U., Arora, A.: Managing change on the web. In: Joint Conference on Digital Libraries, Roanoke, United States, June 2001, pp. 67–76 (2001)
9. Gomez-Perez, A., Corcho, O., Fernandez-Lopez, M.: Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web. Springer, Heidelberg (2004)
10. Khalaf, R., Nagy, W.A.: Business Process with BPEL4WS: Learning BPEL4WS, Part 6. Technical report, IBM (2003), http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol6/
11. Khoshafian, S.: Service Oriented Enterprises, 1st edn. Auerbach (October 2006)
12. Liu, X., Bouguettaya, A.: Managing top-down changes in service-oriented enterprises. In: ICWS 2007, Salt Lake City, Utah (July 2007)
13. Liu, X., Bouguettaya, A.: Reacting to functional changes in service-oriented enterprises. In: CollaborateCom 2007, White Plains, NY (November 2007)
14. Nickols, F.: Change management 101: A primer. Technical report, Distance Consulting (September 2004), http://home.att.net/~nickols/change.htm
15. Orchard, D., Cabrera, F., Copeland, G., Freund, T., Klein, J., Langworthy, D., Shewchuk, J., Storey, T.: Web Service Coordination (WS-Coordination) (March 2004)
16. van der Aalst, W.M.P., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science 270(1–2), 125–203 (2002)
17. W3C. XML Path Language (XPath) (November 1999), http://www.w3.org/TR/xpath
18. WSMO Working Group. Web Service Modeling Ontology (WSMO) (2004), http://www.wsmo.org/
19. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.: Quality-driven Web Service Composition. In: Proc. of 14th International Conference on World Wide Web (WWW 2003), Budapest, Hungary, May 2003. ACM Press, New York (2003)