Reacting to Functional Changes in Service-Oriented Enterprises

Xumin Liu † Athman Bouguettaya †‡
† Department of Computer Science, Virginia Tech, USA
‡ CSIRO ICT Center, Canberra, ACT, Australia
{xuminl,athman}@vt.edu

Abstract

In this paper, we focus on the changes that trigger the modification of a Service-Oriented Enterprise’s functionality. We present a framework that helps an SOE automatically modify its functional schema based on a change specification. The central component of this framework is a change reaction manager. It provides mechanisms to interpret a change specification, modify the member services in an SOE, and rearrange their cooperation correspondingly. A domain knowledge provider offers the semantics that facilitates the change reaction process. We also use a logging mechanism to keep track of the change reaction process.

I. Introduction

The Web service technologies have offered the promises of global accessibility and standardization. This has greatly encouraged more business entities to deliver their functionalities on the Web. This trend is motivating a paradigm shift in enterprise structure from the traditional single, monolithic entity to a collaboration of Web services [7]. Such Service Oriented Enterprises (SOEs) have the potential to open the door of entrepreneurship to all Web users. An SOE is an on-demand and dynamic collaboration between autonomous Web services that collectively provide a value added service to users.

SOEs are gaining momentum as the next generation of loosely coupled enterprise [7]. Current application domains of SOEs include the aircraft industry, tourism industry, computer industry, scientific community, and automobile industry [4], [9]. Adopting the service oriented architecture provides many benefits over traditional enterprises. First, Web services are readily available for integration. Directly outsourcing the functionalities of existing services can reduce the time and cost of starting a new business. Second, as the number of businesses have increased on the Web, it will be possible to select the “best” services from a pool of “similar” services [10]. Finally, there is no geographic boundary that restricts the selection of business partners. Partners may be integrated from any geographic location across the “global village”.

Example I.1 We use an application from the travel domain as a running example to motivate and illustrate our work. The business goal of a travel SOE is to provide a comprehensive travel package for users, including airline booking, hotel reservation, and car rental. To achieve this business goal, the SOE collects the collaborative efforts from the corresponding Web services in the service space. 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 traveller 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. ⊗

A. Motivation

While SOEs present many advantages, they also introduce some new research challenges. SOEs are dynamic due to the dynamic environment they operate in. That is, market conditions do change; business regulations do evolve; individual Web services come and go at their will; and new technologies may emerge over the time. These all may trigger a change on an SOE with respect to the functionality it provides, the way it works, the partners it collaborates with, and the performance it offers. In SOEs, changes are expected to be the rules unlike the case in traditional enterprises. Therefore, a systematic support for handling changes will play a central role for the successfully deployment of an SOE.
Changes in an SOE can be classified into two categories: **bottom-up changes** and **top-down changes**. Bottom-up changes refer to the changes that are initiated by the outsourced Web service providers. Top-down changes, on the other hand, are initiated by an SOE owner. They are usually the result of new business requirements, new regulations, or new laws. Top-down changes in an SOE can be categorized as **functional changes** and **non-functional changes**. Functional changes refer to those that trigger changes on the functionalities of an SOE. In our running example, adding a POI service into the SOE will bring a new functionality (i.e., retrieving local attractions) to the travel SOE. Non-functional changes refer to those that trigger changes on the performance of an SOE. For example, an SOE owner may want to enhance the customer privacy protection to conform to a new regulation. When compared to non-functional changes, dealing with functional changes is usually more challenging. A non-functional change will normally not cause any modification to an SOE’s schema. Instead, it requires an update from the service providers to satisfy the non-functional requirements. Considering the availability of competing service providers, a non-functional change can be handled by selecting the Web services with desired non-functional properties. Existing works on quality-driven service selection [12], [10] can be leveraged to achieve this purpose. In contrast, functional changes always cause the modification of an SOE’s schema [8]. This may even trigger a process of re-engineering the enterprise. Considering that functional changes could occur very frequently due to the dynamic business environment, it is critical to provide a systematic support for efficiently reacting to functional changes on an SOE. Therefore, in this paper, we focus on dealing with the functional changes.

Current approaches to react to functional changes in an enterprise are mainly performed in a manual or predefined fashion. That is, the new functional schema is manually generated or predefined. This drawback greatly limits their applicability since changes in SOEs are assumed to be frequent. Similarly, it is not possible to anticipate all the changes in an SOE and predefine the new functional schema transitions. Therefore, change reaction is expected to be more automatic and intelligent.

**B. Contribution**

In this paper, we propose a framework for handling functional changes in SOEs. The framework aims to automatically update an SOE schema for reacting to a change. This process is facilitated by the semantics offered via a domain knowledge provider. We summarize the key components of this framework, which also constitute the major contribution of this paper, as follows.

- **Domain Knowledge Provider**: We will design and include a domain knowledge provider in the proposed framework. The domain knowledge provider is expected to supply the sufficient semantics for software agents to automatically modify an SOE’s functional schema. It offers two types of semantics. The first one describes the key functionalities of Web services within a domain. The second one describes the relationship between those services.
- **Change Reaction Manager**: We will propose a set of mechanisms to react to functional changes in SOEs. The mechanisms are expected to process a change specification and correspondingly modify the functional schema of an SOE.
- **Change Log Manager**: We will propose a change log manager to keep track of the process of change reaction. In case that there is an exception or failure occurs, the reaction process can be rolled back based on the logged information.

The remainder of this paper is organized as follows. In Section II, we discuss some related work. In Section III, we give a high level overview of the proposed change reaction framework. In Section IV, we present the knowledge base provider and discuss the semantics it provides for change reaction. In Section V, we present the change reaction manager that processes a change specification and reacts to the change. We describe the change log manager in Section VI and conclude in Section VII.

**II. Related Works**

Change reaction is an active research topic in the field of traditional database management, knowledge engineer, and software revolution. Besides, some research efforts are underway to provide change management in a Web service community and adaptive workflow system [3], [2], [6]. In this section, we will discuss some important works and differentiate our work with them.

[3] focuses on handling exceptional changes that can be raised inside workflow-driven Web applications is proposed. It first classifies these changes into behavioral (or user-generated), semantic (or application), and system exceptions. The behavior exceptions are driven by improper execution order of process activities. The semantic exceptions are driven by unsuccessful logical outcome of activities execution. The system exceptions are driven by the malfunctioning of the workflow-based Web application, such as network failures and system breakdowns. It then proposes a framework to handle these changes. The framework consists of three major components: capturing model, notifying model, and handling model. The capturing model captures events and store the exceptions data in the workflow model. The notifying model propagates the...
occurred exceptions to the users. The handling model defines a set of recovery policy to resolve the exception. Different recovery policies apply to different types of exceptions.

[2] focuses on managing bottom-up changes in service-oriented enterprises. It first presents a taxonomy that classifies bottom-up changes into categories. Changes are distinguished between service level and business level: triggering changes that occur 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. Ontologies are used for locating services from an exploratory service space. Agents are employed to assist in detecting and managing changes to the enterprises.

[3], [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. In our paper, we focus on the functional changes, which are expected to change the functionality of an SOE.

[6] 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, we will propose a solution that automatically generates the new version of schema to react to a change.

III. An Overview of the Framework

In this section, we give a high level overview of the framework we propose for handling functional changes in SOEs. A key idea behind our solution is to adopt semantic Web service technologies to facilitate automatic change reaction. As depicted in Figure 1, the change reaction manager is the central component in the proposed framework. It is expected to interpret a change specification and automatically update the functional schema of an SOE. In addition, there are some supporting components in the framework: the domain knowledge provider, the schema container, and the change log manager.

The domain knowledge provider is expected to provide the sufficient semantics for describing Web services within an application domain. It contains the definitions of a set of service ontologies and the relationships among these services. A service ontology is used to capture the functionality a Web service offers. It specify the information about the data items that a service operates on and the operations that a service offers. By the nature of ontologies, Web services can be classified into categories based on their functionalities. The service relationship manager stores the relationship between services in a domain. Web services are self-contained and can be independently invoked. However, once they work together, there will be some dependency constraints on their cooperation. These constraints are domain specific. For example, in our running scenario, a hotel reservation usually depends on the airline ticket booking. That is, the lodging information is determined by the flight information. To enforce this constraint, a hotel service needs to get the output from an airline service when they cooperate. A similar kind of dependency may lie between a car rental service and an airline service. We will use the dependency constraints for automatically generating new functional schema for a change.

The schema container stores the functional schema of an SOE. An SOE functional schema gives a high-level description of an SOE’s functionality (i.e., what it offers) and cooperation patterns between services (i.e., how it works). A service ontology is imported from the ontology definitions provided by the domain knowledge provider. It describes a type of functionality provided by Web services within a domain, such as airline booking, flight status checking, etc [5]. Therefore, we use a set of service ontologies to specify an SOE’s functionality. In our running example, the service list is specified as \{ S_{OA}, S_{OH}, S_c \}, where S_{OA}, S_{OH}, and S_c are the ontologies that describe the functionality of an airline service, a hotel service, and a car rental service. A service ontology can be instantiated by concrete Web services. We also use control flow and data flow to specify how the services cooperate in an SOE.

The change reaction manager implements a set of mechanisms to modify an SOE functional schema on demand based on a change requirement. These include a change specification manager, a message exchange mediator and an execution manager. The change specification manager provides a user interface to input a change specification. To maintain the correctness of an SOE schema, the change specification should first ensure the consistency between the data flow and the control flow of an SOE. Second, Web services are interacted through message exchanges. The data flow should guarantee that each member service will receive all the required input. Therefore, the proposed framework will first determine the new data flow and use it to generate the new control flow. As depicted in 1, the
message exchange mediator modifies the data flow of the SOE based on the specification. The execution manager then updates the control flow to make it consistent with the data flow.

The whole process of change reaction will be recorded by the log manager in the order of time. Information stored in the log manager can be used for a rollback process when a failure or an exception occurs.

IV. Domain Knowledge Provider

In this section, we will describe the semantics offered by a domain knowledge provider. These semantics are domain-specific. They describe the features of the Web services within an application domain and the relationships between these Web services.

Ontology provided by the service ontology manager is expected to enrich the machine-understandable semantics in a service description. Many semantic Web service frameworks have been proposed to define a service ontology, such as OWL-S and WSMO [1], [11]. Since the process of change reaction we propose is mainly based on message exchange among Web services, we will focus on the service data information provided by a service ontology.

A service ontology describes the data that a service operates on as two sets: \(input(I)\) and \(output(O)\). The input of a service consists of a set of data items that reflects how the outside affects a service. It triggers the service invocation. The output consists of a set of data items that shows how the service reacts to the outside. It is the result of the invocation.

There are domain-specific dependency constraints between Web services. These constraints need to be enforced only when these services cooperate together. A dependency constraint can be specified as a triplet \(\{S_1, S_2, D\}\). It means that the invocation of \(S_1\) depends on the invocation of \(S_2\) when they cooperate together since \(S_2\)’s output will be used as the input of \(S_1\) with respect to the data items included in \(D\). The dependency constraint is not necessarily transitive. That is, if \(S_1\) depends on \(S_2\) and \(S_2\) depends on \(S_3\), \(S_1\) does not necessarily depend on \(S_3\). Suppose only \(S_1\) and \(S_3\) attend the cooperation, the dependency constraints \(\{S_1, S_2, D_1\}\) and \(\{S_2, S_3, D_2\}\) do not need to be enforced since \(S_2\) is not involved in the cooperation. In this case, there may be no dependent relationship between \(S_1\) and \(S_3\).

V. Change Reaction Manager

We present the change reaction manager in this section. The key mechanisms of the change reaction manager are provided by its three components: a change specification manager, a message exchange mediator, and an execution manager.
A. Change Specification Manager

A change specification manager provides a user interface to input a specification of a functional change. The specification carries the information about what change is expected to make to the functionality of an SOE. Since the functionality of an SOE can be specified as its member service list, a functional change can be defined as the modification of its member service list. Moreover, the change specification manager will also get the information of the new input and output of the SOE. Therefore, a change specification contains the following information: (1) the services that will be added, denoted by \( S_A \); (2) the services that will be removed, denoted by \( S_R \); (3) the new input, denoted by \( I_N \); and (4) the new output, denoted by \( O_N \).

The change specification interface layouts two types of information: (1) ontology of all the services within the domain; (2) a list of the services included in an SOE. The SOE owner can choose the services from the ontology to add to the SOE. These services will be included in \( S_A \). The SOE owner can also remove the services from the member service list. These services will be included in \( S_R \). These two lists will be used as the input of the other components in the change reaction manager. As a result, the new member service list will be \( S_{\text{new}} = (S_{\text{old}} \cup S_A) - S_R \). After the member service list has been changed, the data flow and the control flow should be modified to maintain the correctness of an SOE. In addition to the newly added or removed services, the change specification interface also layouts a set of data items that are related to the services in \( S_{\text{new}} \) (i.e., their input and output). The SOE owner can specify \( I_N \) and \( O_N \) by selecting the corresponding items.

In our running example, the change requires to add a POI service to the SOE. As a result, the new input should include the category of the POI the user is interested in. The new output should include the list of the POIs. Therefore, it can be specified as:

\[
\begin{align*}
S_A &= \{ \text{POI} \}; \\
S_R &= \phi; \\
I_N &= \{ \text{start_date, end_date, from_city, to_city, interest_categories} \}; \\
O_N &= \{ \text{airline_info, room_info, car_info, POI_list} \}.
\end{align*}
\]

B. Message Exchange Mediator

The message exchange mediator maintains the message exchanges among member services in an SOE. It modifies the data flow when there is an update on the member service list.

The message exchange mediator maintains two message sets for a member service \( S \): \( S_M \) for incoming messages and \( S_M^O \) for outgoing messages. A message can be specified as \( \{ P, D \} \), where \( P \) represents the entity where the message comes from or goes to, and \( D \) is a set of the data items carried by the message. \( P \) can be either a user or another member service. If it is a service, \( P \) will be specified as the service identification. Otherwise, \( P \) will be specified as \( P_u \). In \( S_M \), \( D \) should be a subset of the input of \( S \). In \( S_M^O \), \( D \) should be a subset of the output of \( S \). These message sets thus construct the data flow of an SOE.

Once the member service list has been changed, the message exchange mediator will update the \( M_S \) and \( M_S^O \) for its member services by following three steps. First, it will remove all the messages that involve the services in \( S_R \). It then generates a set \( S_L \), which consists of the

---

**Algorithm Message Exchange Generation**

**Input:** \( I_N, O_N, S_{\text{new}}, S_L \)

**Output:** \( M_E, O_N, S_{\text{new}}, S_L \)

1. \( M_E = \phi \)
2. for each service \( s \) in \( S_L \)
   3. \( S_{\text{left}} = S_{\text{new}} - s \)
   4. \( s = \text{check dependency constraint}(s, S_{\text{left}}, D_I) \)
   5. while \( s ^ \neq \phi \)
     6. generate a new message \( m, m.P = s \), \( m.D = s.O \cap S_{\text{left}} \)
     7. \( M_E = M_E \cup \{ m \} \)
     8. \( D_I = D_I - m.D \)
   9. for each \( s_i \in S_{\text{left}} \)
     10. \( s = \text{check dependency constraint}(s_i, D_I) \)
   11. if \( (I_N \cap D_I) = \phi \)
     12. generate a new message \( m, m.P = P_u \), \( m.D = I_N \cap D_I \)
     13. \( M_E = M_E \cup \{ m \} \)
     14. \( D_I = D_I - m.D \)
     15. while \( D_I \) is not empty and \( S_{\text{left}} \) is not empty
     16. choose \( s' \) from \( S_{\text{left}} \)
     17. if \( s'.O \cap D_I = \phi \)
     18. generate a new message \( m, m.P = s' \), \( m.D = S_{\text{new}} \cap D_I \)
     19. \( M_E = M_E \cup \{ m \} \)
     20. \( D_I = D_I - m.D \)
     21. if \( D_I \) is not empty
     22. insert an error message \( m \) to \( M_E \) if \( m \) contains the identification of \( s \) and \( D_I \)
     23. \( L_O = O_N \)
   24. for each service \( s \) in \( S_{\text{new}} \)
     25. if \( s.O \cap O_N \) is not empty
     26. generate a new message \( m, m.P = P_u \), \( m.D = S_{\text{new}} \cap O_N \)
     27. \( M_O = M_O \cup m \)
     28. \( L_O = L_O - m.D \)
     29. if \( L_O = \phi \)
     30. insert an error message \( m \) to \( M_E \) if \( m \) contains the identification of \( P_u \) and \( L_O \)

---

Fig. 2. Algorithm of Message Exchange Generation
services that used to receive messages from the services in $S_R$. Let $D_{Is}$ denote the input data of $S$ that is not covered by the data items delivered in $S$’s incoming messages. Second, for each service in $S_A$, it will create $M_{Si}$ and $M_{So}$, which are initially empty. Let $S_L = S_L \cup S_A$, $S_L$ will consist of every service $S$ where $D_{Is}$ is not empty. Finally, the mediator will generate new message exchanges between services to make sure that each one can get the required input. These message exchanges are based on the input/output matching between member services.

Figure 2 depicts the algorithm for automatically generating message exchanges for reacting to a change. The input is derived from the change specification. This algorithm will update the incoming and outgoing message sets of the member services. The result of performing this algorithm is a message set $M_E$. The message in $M_E$ contains the information about the services that cannot get all the required input either from the user or from the other member services. If $M_E$ is empty, it means that the data flow has been successfully updated. Otherwise, the SOE owner should consider to either require more input from the user side or add another service from the domain to fix the problem.

For each service in $S_L$, the mediator will first check the dependency constraint that need to be met between the member services. In Line 04, the check dependency constraint function is invoked. This function takes the input of a service $s$, a service set $SS$, and a data set $DS$. It will check whether there is a dependency constraint between $s$ and the service in $SS$ with respect to the data item in $DS$. If there is, the function will return the first such a service from $SS$. A message exchange will be set up between $s$ and its depending service. This process will be continually performed till there are no more dependent constraints that need to be met. After this, if service $s$ still has some input that needs to be provided, the mediator will check the $IN_s$, as depicted from Line 11 to Line 14. If there is a match between $D_{Is}$ and $IN_s$, a message exchange will be set up. After that, if $D_{Is}$ is still not empty, the mediator will check the other member services to find the resources for the data in $D_{Is}$ and generate the corresponding message exchanges, as depicted from Line 15 to Line 20. If $D_{Is}$ is still not empty after checking all the other services, an error message will be included in $M_E$ to deliver the relative information. Finally, it will check whether the output of the SOE can be generated by its member services. If not, an error message will be sent to $M_E$.

Figure 3 shows the modification of the data flow in our running example. A POI service is added. As a result, there are two message exchanges generated: $e_1$ and $e_2$. The POI service is dependent on the hotel service since users usually tend to get the POI information around the place they live. Therefore, the POI service needs to take the hotel address as its input, which is the purpose of generating $e_1$. $e_1$ consists of two messages: $M1$ and $M1'$, which correspond to the incoming message of the POI service and the outgoing message of the hotel service respectively. $e_2$ shows that the POI service generates the output and returns it to users.

**Fig. 3. The Modification of the Data Flow**

**Fig. 4. The Modification of the Control Flow**

### C. Execution Manager

The execution order between member services are determined by their dependency constraints, which are captured by the data flow. Therefore, the control flow can be modified by the updates of the dependency constraints generated by the message exchange mediator.

The execution manager maintains the execution order among member services in an SOE. The control flow can be specified as a directed acyclic graph (DAG), where vertices denote the member services and edges denote the execution orders. The execution manager follows two steps to modify the control flow. First, based on the change specification, the execution manager will delete all edges that either come from or go to the services that have been removed. It will then add the new vertices for the services that have been added. Second, for a service $s$, the services that provide $s$’s input should be invoked before $s$. We use $B_s$ to denote these services. Meanwhile, the services that wait for $s$’s output as their input should be invoked after
TABLE I. Log records for an SOE modification

<table>
<thead>
<tr>
<th>log_time</th>
<th>action</th>
<th>object_type</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Add</td>
<td>SV</td>
<td>s_P0I</td>
</tr>
<tr>
<td>t2</td>
<td>Add</td>
<td>MSG</td>
<td>e1 = {M1, M1′}</td>
</tr>
<tr>
<td>t3</td>
<td>Add</td>
<td>MSG</td>
<td>e2 = {M2, M2′}</td>
</tr>
<tr>
<td>t4</td>
<td>Add</td>
<td>EO</td>
<td>e3</td>
</tr>
</tbody>
</table>

s. We use $A_s$ to denote these services. Therefore, for each service $s$ in $S_L$, the execution manager will check and make sure that there is an edge coming from the services in $B_s$ and there is an edge going to the services in $A_s$. Figure 4 shows the modification of the control flow in our running example. Since there is a message coming from the hotel service and going to the POI service, a new edge is added in the graph.

VI. Change Log Manager

In this section, we discuss the change log manager in our framework. The purpose of the log manager is to keep track of the entire change reaction process. By doing this, an SOE can be recovered from an exception.

The log manager maintains a set of log files that record the modification of an SOE’s functional schema. A log record contains the information about: log_time, action, object_type, and object. The log_time specifies the time the log is generated. The action specifies the type of the modification. It could be Add for adding or Rem for removing some element. Type of the element is specified by the object_type. A object_type could be SV for services, MSG for message exchanges, EO for execution orders. The object specifies the element that is involved in the modification. In our running example, the following log records will be generated.

Checkpoints can be defined in the log manager. Once there is an error, the change reaction process can be either rolled forward or rolled back based on these checkpoints. Since the modifications are made from a global perspective, it cannot guarantee that an SOE can be back to a correct status at the some intermediate stage of the modification process. When an error occurs during the reaction process, the SOE needs to return to a correct state to ensure its proper functioning. A conservative approach is to cancel all the changes and take the SOE back to the stage where no modification is made. A key extension of this work is to realize a progressive change reaction strategy, which guarantees an SOE to have a correct state at a set of key checkpoints during the entire change reaction process.

VII. Conclusion

In this paper, we have proposed a framework for reacting to functional changes in an SOE. Changes are specified with respect to the updates on the member service list and the new input/output of the SOE. Based on the change specification, several mechanisms are used to automatically modify the data flow and the control flow. By doing this, a new functional schema will be generated to make sure the SOE can function well and reach the goal of introducing the change.

References