Change Management of Long Term Composed Services

Xumin Liu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Dr. Athman Bouguettaya, Chair
Dr. Dennis Gracanin, Co-chair,
   Dr. Reza Barkhi
Dr. Shawn Bohner
Dr. Chang Tien Lu

July 28th, 2009
Blacksburg, Virginia, USA

Keywords: Change Management - Long term Composed Services - Ontology - Top-down

Copyright 2009, Xumin Liu
Change Management of Long Term Composed Services

Xumin Liu

(ABSTRACT)

We propose a framework for managing changes in *Long term Composed Services* (LCSs). The key components of the proposed framework include a *Web Service Change Management Language* (SCML), *change enactment*, and *change optimization*. The SCML is a formal language to specify top-down changes. It is built upon a formal model which consists of a *Web service ontology* and a *LCS schema*. The Web service ontology gives a semantic description on the important features of a service, including *functionality*, *quality*, and *context*. The LCS schema gives a high-level overview of a LCS's key features. A top-down change is specified as the modification of a LCS schema in the first place. Change enactment is the process of reacting to a top-down change. It consists of two subcomponents, including *change reaction* and *change verification*. The change reaction component implements the proposed change operators by modifying a LCS schema and the membership of Web services. The change verification component ensures that the correctness of a LCS is maintained during the process of change reaction. We propose a set of algorithms for the processes of change reaction and verification. The change optimization component selects the Web services that participate in a LCS to ensure that the change has been reacted to in the best way. We propose a two-phase optimization process to select services using both *service reputation* and *service quality*. We present a change management system that implements the proposed approaches. We also conduct a set of simulations to assess the performance.
To my husband, Qi,

my daughter, Emily.
Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Athman Bouguettaya, for his continuous support during my entire PhD study. Dr. Bouguettaya’s keen interest and passion in research, his vision in the fields of service computing and data management, and his disciplined research methodology truly inspired me and will guide me throughout my career. It is the tremendous time and effort that he devoted on advising that trained me into a real researcher. His disciplined guidance is the most valuable asset for me to learn how to conduct independent and creative research. Dr. Bouguettaya’s dedication, patience, and inspiration are a great fortune to my PhD study and are highly appreciated.

I want to thank my thesis committee members, Professors Reza Barkhi, Shawn Bohner, Denis Gracanin, and Chang-Tien Lu, for their advice and valuable input on my thesis research, for their support and encouragement in my job search, and their willingness to change their schedules to accommodate the exams and meetings.

I want to thank my collaborators: Salman Akram, Athman Bouguettaya, Denis Gracanin, Zaki Malik, Brahim Medjahed, Mourad Ouzzani, Abdelmounaam Rezgui, Jemma Wu, Xu Yang, Qi Yu, Xiaoyu Zhang, and Li Zhou for their productive and enjoyable collaborations. Many thanks go to Li Zhou who helped implement the user interface for the change management system.

I am grateful to the colleagues at ECEG research lab in Virginia Tech: Salman Akram, Weiping He, Zaki Malik, Brahim Medjahed, Mourad Ouzzani, Abdelmounaam Rezgui, Xu Yang, Qi Yu, and George Zheng. I appreciate them for providing such an inspiring working environment and offering numerous help for both my research and life.

The continuous support from my family is instrumental to both my life and my research. I want to thank my parents, my sisters, and my brother for their constant love and support. I want to thank my daughter, Emily, for the happiness, sunshine, and enjoyable moments she has brought to my life. Last but not least, my most special appreciation goes to my husband Qi Yu. It is his continuous encouragement, everlasting support, and deep love that made me reach this far.
Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Motivation ................................................................. 2

1.1.1 Long term Composed Services ....................................... 3

1.1.2 Change Management ................................................... 4

1.2 A Travel Scenario .......................................................... 7

1.3 Research Issues ............................................................ 9

1.4 Summary of Contributions ............................................... 10

1.5 Dissertation Organization ................................................. 12

2 A Overview of an End-to-end Change Management Framework 14

2.1 The Formal Model .......................................................... 14

2.2 Change Model .............................................................. 16

2.3 Change Enactment .......................................................... 16

2.4 Change Optimization ....................................................... 17

3 Web Service Ontology 19

3.1 Overview of the Web Service Ontology ................................ 19
3.2 Service Concepts ......................................................... 20
  3.2.1 Functionality ....................................................... 21
  3.2.2 Quality ............................................................. 22
  3.2.3 Context ............................................................. 24
3.3 Service Relationships .................................................. 25
  3.3.1 Inheritance Relationship .......................................... 25
  3.3.2 Dependency Relationship ......................................... 26
3.4 Service Ontology Query Infrastructure .............................. 27

4 A Supporting Infrastructure of a LCS ................................. 33
  4.1 An Overview of a LCS ................................................ 33
  4.2 LCS Schema .......................................................... 35
    4.2.1 Schema Graph ................................................... 35
    4.2.2 LCS Quality ...................................................... 37
    4.2.3 LCS Context ...................................................... 37
  4.3 Schema Correctness ................................................... 38
    4.3.1 Structural Correctness ......................................... 38
    4.3.2 Semantic Correctness ......................................... 39

5 The SCML Language ....................................................... 41
  5.1 Change Taxonomy .................................................... 41
    5.1.1 Functional Changes ............................................. 42
    5.1.2 Non-functional changes ...................................... 47
  5.2 Change Operators ................................................... 48
    5.2.1 Functional Change Operators ................................ 48
    5.2.2 Non-functional Change Operators ............................. 50
  5.3 Change Language .................................................... 50
5.3.1 Create Command ........................................... 51
5.3.2 Select Command ........................................... 53
5.3.3 Alter Command ........................................... 53
5.3.4 Update Command ........................................... 55
5.3.5 Drop Command ........................................... 55

6 Change Enactment and Change Optimization .......... 56
  6.1 Change Enactment ........................................... 56
    6.1.1 Select Service Nodes ................................... 57
    6.1.2 Update LCS schema Graphs ................................... 57
    6.1.3 Verify Changes ........................................... 62
    6.1.4 Compile LCS Instances .................................... 67
  6.2 Change Optimization ....................................... 69
    6.2.1 Phase I: Reputation Based Optimization ................. 69
    6.2.2 Phase II: QoWS Based Optimization ...................... 72

7 Implementation .............................................. 74
  7.1 User Interface ............................................. 74
  7.2 Ontology Support .......................................... 76
    7.2.1 Ontology Providers ..................................... 76
    7.2.2 Web Service Registry .................................... 77
    7.2.3 WSMO APIs .............................................. 77
  7.3 Change Management ......................................... 78
  7.4 Visualization .............................................. 81
  7.5 System Usage ............................................... 84

8 Performance Study ........................................... 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Change Enactment Performance</td>
<td>89</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Effect of the LCS Size</td>
<td>90</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Effect of Change Size</td>
<td>92</td>
</tr>
<tr>
<td>8.2</td>
<td>Ontology Query Optimization</td>
<td>94</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Depth of the Service Ontology</td>
<td>95</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Fanout of the Service Nodes</td>
<td>95</td>
</tr>
<tr>
<td>8.3</td>
<td>Change Optimization</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>Related Work</td>
<td>99</td>
</tr>
<tr>
<td>9.1</td>
<td>Change Management in Web Service Community</td>
<td>99</td>
</tr>
<tr>
<td>9.2</td>
<td>Adaptive Workflow System</td>
<td>100</td>
</tr>
<tr>
<td>9.3</td>
<td>Change Management in Traditional Databases</td>
<td>102</td>
</tr>
<tr>
<td>9.4</td>
<td>Ontologies for Web Services</td>
<td>102</td>
</tr>
<tr>
<td>9.4.1</td>
<td>OWL</td>
<td>103</td>
</tr>
<tr>
<td>9.4.2</td>
<td>OWL-S</td>
<td>104</td>
</tr>
<tr>
<td>9.4.3</td>
<td>WSMO</td>
<td>105</td>
</tr>
<tr>
<td>10</td>
<td>Conclusion</td>
<td>107</td>
</tr>
<tr>
<td>10.1</td>
<td>Research Summary</td>
<td>107</td>
</tr>
<tr>
<td>10.2</td>
<td>Directions for Future Research</td>
<td>109</td>
</tr>
<tr>
<td>B</td>
<td>Bibliography</td>
<td>111</td>
</tr>
<tr>
<td>A</td>
<td>Acronyms and Abbreviations</td>
<td>118</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Change reaction for adding a car rental service .................................. 7
2.1 The change management framework .................................................. 15
3.1 A Web service ontology ................................................................. 20
3.2 An example of an airline service ontology in OWL-S ......................... 21
3.3 A context transported in a SOAP header block .................................. 24
3.4 A Travel Domain Ontology ........................................................... 26
4.1 The architecture of a LCS ............................................................... 34
4.2 An example of a LCS schema graph ............................................... 36
5.1 A taxonomy of top-down changes in a LCS ..................................... 42
5.2 Four types of process constraints .................................................. 47
6.1 The potential conflict and redundancy .......................................... 58
6.2 The updated travel agency LCS schema ....................................... 66
6.3 Two-phase optimization process .................................................... 69
7.1 The change management system .................................................... 75
7.2 Importing a Web service ontology ............................................... 81
7.3 Importing a LCS and a change ..................................................... 82
7.4 The west-oriented view control flow of a LCS after change management ... 83
## List of Tables

3.1 Measurement of QoWS parameters ........................................... 23

4.1 QoWS for a LCS instance ....................................................... 37

4.2 The LCS schema correctness criteria ....................................... 40

5.1 Top-down change model ....................................................... 48

8.1 Experiment parameters ........................................................ 90
Chapter 1

Introduction

The last few years have witnessed an explosion of activity around Web services [50, 63, 66, 37, 28, 21, 64, 9, 80, 65]. Web services are distinguished from other traditional applications by two major features: global availabilities and standardization. First, Web services take advantages of the powerful communication paradigm of the Web to provide global availabilities [60]. Second, they are built upon XML-based standards as a vehicle for exchanging messages across heterogeneous Web applications [33, 69, 70, 73, 68, 72, 71, 56, 10].

Web services enable application integration on a large scale and in a distributed, heterogeneous, and dynamic environment. A Web service is defined as a software system designed to support interoperable machine-to-machine interaction over a network [8]. It is the central component in a Web service architecture. A Web service architecture consists of three roles: service providers, service consumers, and service brokers. A service provider offers business functionalities. A service consumer uses Web services. A service broker mediates between a service provider and a service consumer. It enables a service provider to publish services and a service consumer to discover services. A Web service architecture also incorporates three key XML-based standards to support the roles and their interaction. These standards include Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description Discovery and Integration (UDDI). WSDL defines a Web service as a set of endpoints operating on messages [73]. It is used by a service provider to publish their services. SOAP defines a communication protocol that can be used by a service consumer to access Web services [69]. UDDI defines a registry for advertising and discovering Web services [70].

Pushed by the leading IT companies, the Web is poised to take a new giant step to be a central repository for the ever increasing number of Web services [51]. This will have
the effect of transforming the Web from a \textit{data-oriented} repository to a \textit{service-oriented} repository, called the \textit{service Web}. Significant efforts are invested in developing Web services from both academia and industry.

Service Oriented Computing (SOC) is emerging as the new paradigm to facilitate 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 [7]. A \textit{composed 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. There are two types of composed Web services: \textit{short term} and \textit{long term}. In a short term composed service, the partnership between different services are transient. The services are usually composed for the purpose of a specific and temporary business goal. Once the goal is reached, the collaboration will be dissolved. An example of such short term composed Web services is a travel planner, which comes up with a trip plan for users by collaborating with different travel services, such as airlines, hotels, and car rentals. Once the travel is finished, the travel planner will no longer be used and the collaboration between these services will be dissolved. In a \textit{Long term Composed Service} (LCS), the partnership between different services are permanent. The services are composed to fulfill a business goal in the long run. A typical application of a LCS is a \textit{Service Oriented Enterprise} (SOE) [39], which has a long term commitment to its users. Long term composed services (LCSs) have attracted a lot of attention since they provide a powerful tool to offer value-added and customized services. While there has been a large body of research in the automatic composition of Web services, managing the changes during the lifecycle of LCSs has so far attracted little attention [7, 84, 14].

\section*{1.1 Motivation}

A LCS consists of several autonomous outsourced Web services, but acts as a \textit{virtually} coherent entity. Business entities, in the form of Web services, are often geographically distributed and organizationally independent. While LCSs have great promises of introducing new business opportunities through dynamic alliance, the challenges of fully realizing a LCS lie in managing changes during its lifecycle.


1.1.1 Long term Composed Services

LCSs aim at making a full usage of online services by combining them to provide value-added services. As more and more business entities publish their functionalities on the Web via Web services, LCSs have been rapidly adopted as a new paradigm that enables flexible and on-demand collaboration between different business entities. This envisions the wide application domains of LCSs, including scientific computing, tourism industry, computer industry, automobile industry, etc. We summarize several representative benefits that LCSs provide as below.

- First, the provisioning of Web services drastically reduces the capital required to start a business. Web services are readily available for integration and orchestration. Thus, investments made by other businesses may be reused for mutual profit. Moreover, the provisioning of Web services reduces the time to market as well. Since Web services exist prior to their selection, their market competitiveness and reputation will naturally benefit the LCSs that outsource them.

- Second, the partners of LCSs can be selected dynamically. Web services provide machine-processible APIs that enable themselves to be invoked and orchestrated automatically [30]. Hence, LCSs enable an on-demand, project-driven alliance between different business entities. Moreover, there is no geographical boundary that restricts the selection of business partners. Hence, LCSs enable a wide-integration of business entities from the “global village”.

- Third, as more and more business entities have brought their functionalities on the Web, it will be possible that multiple service providers compete to offer the similar services with different user-centric quality [81]. The “best” services can be selected from those providers to form LCSs. Therefore, the consumer or end user of LCSs will benefit from the open competition between businesses.

The lifecycle of a LCS is a series of stages through which it passes from its inception to its termination. There are four phases in a LCS lifecycle: planning, composition, orchestration, and dissolution. The planning phase is the first stage, where the LCS is described at a high level. It is initiated when the owner of the LCS establishes a need for a business objective [53]. The composition phase deals with integrating the selected Web services [42]. After this, the selected Web services are orchestrated to provide the value-added service. The dissolution phase occurs when the owner of the LCS decides that the execution of the LCS is no longer required.
1.1.2 Change Management

Fully realizing LCSs lies in providing support to improve their adaptability to the dynamic environment, i.e., to deal with changes during the lifetime of a LCS. Because of the dynamic nature of Web service infrastructure, changes should be considered as the rule and managed in a structured and systematic way [34]. Changes are usually introduced by the occurrence of new market interests, business regulation, new technologies, etc. Such changes are always associated with a requirement on the modification of a LCS with respect to the functionality it provides, the way it performs, the partners it is composed of, and the performance it delivers. Once a change occurs, a LCS needs to quickly adjust itself to fulfill the requirement introduced by the change. The adjustment also needs to be performed in an automatic manner considering the frequent occurrence of the changes. By doing this, a LCS can maximize the market interests it attracts, optimize the way it outsources its functionality, and thus maintain its competitiveness among its peers.

Changes in a LCS can be classified into two categories: top-down changes and bottom-up changes. Top-down changes refer to those that are initiated by the owner of the LCS. Bottom-up changes refer to those that are initiated by the outsourced service providers.

- Top-down changes refer to those that are initiated by the owner of a LCS. They are usually the result of business policies, business regulations, or laws. For example, a LCS outsources its functionality from services in travel domain to offer a comprehensive travel package (referred to as a travel LCS). According to a new market report, the LCS may want to expand its business by adding a new local activity service.

- Bottom-up changes refer to those that are initiated by the outsourced service providers. They are usually the result of an error, an exception, or an alternation in a Web service space. Web services may change their features independently without the consent of the LCS that utilizes their functionalities. For example, a map service may become unavailable due to a network failure. For another 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 dissertation, we focus on dealing with top-down changes. Top down changes are triggered by a set of change initiators, which can be business policies, business regulations (or laws), and the LCS’s owner.
The business policies constitute a specification of what a LCS must deliver or accomplish. They reflect a LCS’s goals. For example, a travel agency LCS may have the policy of “maintaining its annual income over 20 millions”. A business policy can be specified in terms of assertions [74]. A top-down change might be triggered to ensure a LCS to enact the policy. For example, if the LCS fails to achieve over 20 million income annually, it needs to increase its profit by adding a new service to attract more business opportunities or deleting a service to decrease the expenses. Meanwhile, business policies change with the business environment, which may also trigger top-down changes. For example, suppose that customers consider the privacy protection when choosing a travel agency. In this case, the LCS may adopt a new policy stating that all the outsourced service providers should provide good privacy protection. To enforce this policy, the services that provide poor privacy protection will be replaced by those that provide good privacy protection.

The business regulations or laws may also trigger top-down changes. For example, suppose that there is a new regulation that the online payment should be performed by a third party service. This will subsequently initiate the process of adding an online payment service to the LCS, such as VeriSign, Paypal, or Google Checkout.

In addition, a LCS’s owner can make any changes as his or her intention.

Change initiators introduce changes to a LCS by enforcing a new requirement, functional or non-functional. A functional requirement is enforced on what a LCS offers. For example, a travel agency LCS may be required to add a traffic service to attract more customer interests. A non-functional requirement is enforced on how well a LCS performs. For example, a travel agency LCS may be required to respond to its user request more quickly.

There are some existing standards providing related functionalities that can be utilized to deal with changes of services. For example, Web Services Eventing (WS-eventing) and Web Services Notification (WSN) [46, 75] focus on providing an event-based framework that monitors the activities of Web services. Once there is an event occurs in a service, it can be sent to the other services via message passing. By affiliating a change with an event, the change on one or more member services of a LCS can be propagated to the other LCS participants. Nevertheless, the design focus of this framework is not for change management purpose. Furthermore, it does not support for dealing with top-down changes at all, which are initiated by the dynamic business environment, not by the individual services.

Change management has been a major research topic in the area of workflow management. Workflow Management Systems (WfMSs) aim at coordinating activities between different
business processes (systems) so that they can run in an efficient manner. This is done by automating the business processes and invoking appropriate resources in a sequence. According to [18], “workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve or contribute to an overall business goal.” There are many frameworks proposed for supporting adaptive workflow systems [23, 62, 13, 36, 45]. These works mainly address the issue of handling the running instances when the schemata has been modified. Approaches are proposed to maintain correctness between process types and instances. A process type represents a particular business process described by a schema. A process instance is a real time execution of the process type. Changes to a process type occur when the process schema is modified in response to the environment. For example, a business process may adapt to comply with a new legislation, or it may be optimized for performance reasons. The respective process type changes must be propagated to the process instances.

The major research issues of change management in LCSs are different from the ones in workflow systems. This is due to the different contexts and assumptions, which are described as follows.

- LCSs have the promise of enabling flexible collaboration between different business entities. They are expected to be capable of adapting to the dynamic environment in a prompt way. In this case, the modification of the collaboration is expected to be frequent and requires a systematic support. In contrast, the collaboration between different entities of workflow systems is usually within an organization. The environment that the systems interact with are relatively “static”. Changes on the schemata are treated as “exceptions” and usually performed in a manual way.

- LCSs outsource their functionalities from autonomous service providers. There is no central control mechanism that a LCS can rely on to monitor and manage changes. In contrast, the entities in workflow systems are within an organization, which enables a centralized approach to monitor and manage the changes.

Therefore, change management in LCSs has different focuses from the one in adaptive workflow systems. It is not sufficient to manage changes in a LCS by simply applying the approaches adopted in existing frameworks.
1.2 A Travel Scenario

In this section, we use an application from the travel domain as a running example to motivate and illustrate our work.

Consider a travel agency LCS that offers several types of functionalities, including airline, hotel, taxi, weather, and online payment (depicted in Figure 1.1). Typical examples of such travel agency LCSs may include Priceline [52] and Expedia [24], etc. Suppose that the travel agency LCS aims to provide a comprehensive travel package that outsources the functionalities from different service providers. Users can thus book airlines, reserve hotels, rent cars, reserve taxis, or check weather information by directly accessing this LCS. These services can be independently invoked. Thus, users need to provide the information that are required to invoke the services. When services are combined together, there might be some dependency relationship between them. These dependencies determine the composition of the services. In this case, users do not necessarily need to provide the information for each service. The
input of some services can be derived from the dependency relationship. For example, when
invoking an airline service and a hotel service at the same time (i.e., they are included in
the same travel package), the input information for the hotel service is determined by the
flight information, which is the output of the flight service. More specifically, the city and
the check in-and-out dates of the hotel service are all determined by the flight information.
These dependency relationships need to be enforced when composing Web services together.

Due to the LCS’s dynamic business environment, it is obviously that changes may occur
in LCSs. Suppose that a market survey shows that car rental services have attracted more
interest than taxi services serving as the ground transportation. In this case, the owner of
the travel agency LCS may want to replace the taxi service by a car rental service. Moreover,
users that choose car rental services probably also take interest in the local traffic information,
such as the driving direction from the airport to the hotel. The traffic information may keep
changing for some reason such as some roads may become under construction. Therefore,
the traffic service outsourced is expected to provide the up-to-date traffic information. In
addition, users may tend to include local activities to their travel packages nowadays. For
example, when a user plans a trip to Orlando, s/he may also want to visit the local activities,
such as the Universal Orlando and the SeaWorld Orlando. In this case, s/he may want to
reserve the tickets for these activities via the travel agency. If the travel agency LCS does not
incorporate a local activities Web service into the enterprise, it takes the risk of becoming
obsolete and loosing business.

To implement the above changes, the travel agency LCS needs to add three types of services
including: car rental, traffic, and local attraction. It also needs to remove the taxi service. Once
there is an update on the participation of services in a LCS, the service composition needs
to be re-generated. Therefore, the owner of the LCS, say John, needs to first examine the
specification of each type of participating service. He then needs to design the composition
between these services, including their data transfer and invocation order. He then needs to
locate concrete Web services that provide the desired functionality. Moreover, when adding a
service, there may be multiple service providers that compete with each other to provide the
similar functionality. These service providers are different from each other in terms of their
delivered quality, such as the fee they charge and the duration of invoking the service, etc.
In this case, John needs to find the one that provides the best quality. Each of these service
providers has promised their quality. However, there might be some violations between the
promised quality and the quality they actually deliver. For example, a car rental service A
may be advertised that its price is $30 per day However, it actually charges $35 per day. John
may not know most of these services before and there is no central monitoring mechanisms
on the Web that he can rely on. As a result, he needs to use some mechanisms to determine
whether the service is trustable or not.

This scenario justifies that the process of change reaction could be very complicated, which makes it impractical to manage changes in a manual way. Therefore, a systematic framework is required to provide disciplined support for change management in LCSs.

1.3 Research Issues

Change management in the context of LCSs poses a set of research issues. A LCS 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 specify and manage a top-down change in a way that best reacts to the change. We summarize the major research issues of managing changes in LCSs as follows.

R1: *Semantics support:* To improve the automation of change management, it is essential that software agents can understand the problems they are dealing with and the approaches they can use. Specifically, they need to have sufficient knowledge about LCSs and their outsourced services. In databases, the automation of data queries and updates relies on metadata (i.e., data types and data schema). Similar meta-information is also required to provide a semantic support for the process of change management.

R2: *Change model:* Changes need to be first captured and identified so that they can be understood and processed. Different changes are different in terms of their motivations, requirements, behaviors, etc. This naturally triggers a need to differentiate changes in that different strategies can be deployed. Traditional approaches of specifying top-down changes are *ad-hoc* or *informal*. This will involve intensive human interventions. Therefore, change should be modeled in a *formal* way that can be understandable and processable by machines.

R3: *Change reaction:* A top-down change is always associated with a new requirement on a LCS. It may require the LCS to add a new functionality or improve privacy protection. To respond to a change, a LCS may need to change the services that it outsources, and the way that the outsourced services cooperate with each other. It would introduce a high cost if the process is performed in a manual way. To automate this process, the information about the outsourced services and their collaboration needs to be
understood by machines. In such a way, the proper modification on a LCS can be automatically performed to fulfill the requirement introduced by the change.

**R4: Change verification:** The process of change reaction needs to be verified. Put differently, any modification on a LCS needs to be verified to ensure that the LCS is evolved from a correct configuration to another correct configuration. For example, in a travel agency LCS, the hotel service may rely on an airline service to provide the customers’ check-in/check-out information. Suppose that a change requires to remove the airline service. If the hotel service cannot get the information from the user or the other services in the LCS, it will not be able to be invoked properly. For the verification purpose, it needs to first define the correctness of LCSs. Based on the correctness, it then needs to evaluate the process of change reaction.

**R5: Change optimization:** The result of change reaction needs to be optimized to ensure that the change is managed in the best way. Considering the large amount of Web services expected to be available on the Web, there may be multiple services that compete to provide similar functionalities with different performance. For example, there are multiple service providers that offer airline services, such as United, Delta, Airway, etc. Therefore, when there is a need of replacement and addition of Web services, there may be multiple candidates. This triggers a need to choose the “best” service from those candidates.

### 1.4 Summary of Contributions

In this section, we outline our end-to-end approach for managing top-down changes in LCSs. We first define a *formal model* that provides the grounding semantics for describing, reacting to, verifying, and optimizing changes in LCSs. Based on the formal model, we present a change language, which specifies top-down changes. We then propose a multiple-level process of change reaction. Different types of changes will be implemented on different reaction levels. We then verify the modification on LCSs from different perspective to ensure their correctness. After that, we propose a multiple-phase optimization process to generate the “best” result. We summarize our contribution as follows.

**C1:** *Web service ontology:* We define a formal model to provide the necessary grounding semantics to support the automation of change management. The context of changes are LCSs and their outsourced services. Therefore, the formal model contains the
machine-understandable description of LCSs and Web services. We first propose a Web service ontology to effectively capture the key features of a Web service. A service ontology can also be used as a meaningful organization of Web services based on their functionalities. The semantics it provides enable automatic service discovery and integration.

**C2: Ontology query:** Considering that a large amount of Web services is expected to be available on the Web, a Web service ontology may contain a large body of information about the Web services within a domain. Therefore, we propose a set of algorithms to efficiently retrieve required semantics from a Web service ontology.

**C3: LCS schema:** We propose a LCS schema that gives a high-level description of a LCS. A LCS schema captures what a LCS offers and how it works. It plays a similar role in a LCS as what a data schema plays in a database. Using a LCS schema, change management can be performed in a top-down fashion. The general idea of what a change is and how to manage it can be first generated. It will then be translated to the specific situation, where more technical details are involved.

**C4: Web Service Change Management Language (SCML):** We propose a change management language, SCML, to specify top-down changes in LCSs. A change is always associated with a requirement on a LCS. To fulfill such a requirement, a LCS needs to modify its features, functional or non-functional. The functional features are related to the functionality that a LCS offers. The non-functional features are related to the performance that a LCS delivers. Therefore, we propose a taxonomy of changes, which classifies changes into different categories based on their requirements. Based on the taxonomy, different change functions are then defined. These functions constitute a change management language, which describes a change in a formal and declarative way.

**C5: Two-level change reaction:** We propose a process of change reaction which implements the change language on two levels: schema-level and instance-level. During the first level of change reaction, a LCS schema will be modified to conform to the change requirement. There are two major types of modifications at this level. First, a LCS may change the list of outsourced services. For example, it may find a new service to outsource or terminate its partnership with an outsourced service. Second, a LCS may change the way that it outsources services. For example, it may want to serialize the invocation order between two services, instead of parallelizing it. During the second level of change reaction, a new instantiation of a LCS schema will be generated. Web
services that fulfill the change requirement will be selected and orchestrated. Considering that there may be multiple service providers competing to offer the similar functionality, multiple instances of the LCS can be generated.

C6: Change verification: We propose a process to verify the process of change reaction to ensure that the change has been responded properly. We first define a set of criteria that measure the correctness of a LCS’s configuration. We then use the criteria to check the process of change reaction. Once there is a modification made on a LCS schema, the modification will be verified. If there is an error detected, it will be corrected by further modifying the LCS. We analyze the impact of the order of verifying modifications on the process of change verification. We then devise an order of checking correctness of LCSs to avoid the modification that would cause potential errors.

C7: Two-phase change optimization: We propose a two-phase optimization process to optimize the result of change management. Web services are provided by independent service providers. They may be new or no prior history [42, 49]. Web services may make promises about the provided service and its associated quality but may fail partially or fully to deliver on these promises. Therefore, during the first phase of optimization, we use a service reputation as a filter to remove the Web services that have a low reputation. During the second phase, we use Quality of Web Services (QoWS) to define a cost model, which represents the users’ preference on their choices of Web services. We then use this cost model as the second filter to choose the best one from the remaining trustworthy services.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we give an overview of the end-to-end change management framework. In Chapter 3, we present a Web service ontology. In Chapter 4, we propose a formal definition of the LCS schema. The proposed service ontology and LCS schema are used as a formal model to provide the semantics for change management. Based on this formal model, we propose a Web Service Change Management Language (SCML) to specify top-down changes. The SCML language consists of a set of change operators, which are described in Chapter 5. In Chapter 6, we propose a set of steps to enact changes. We first propose a set of algorithms to implement the change operators. We then propose a process to verify the changes. We also propose a two-phase optimization approach to get the best result of change management. In Chapter 7,
we present an initial prototype that implements our proposed approaches. In Chapter 8, we present our experimental study on the performance of the proposed approaches. In Chapter 9, we overview the related work. The dissertation is concluded in Chapter 10 by summarizing the major contributions and discussing some future directions.
Chapter 2

A Overview of an End-to-end Change Management Framework

In this chapter, we give a brief overview of the proposed end-to-end framework for change management, which is depicted in Figure 2.1. The framework aims to identify the key components to introduce, model, and manage changes to ensure that the changes can be dealt with in a proper way. The bottom of the framework is a Web service space, which consists of the actual service providers in a certain domain, a.k.a., service instances. LCSs are on top of the service space. They outsource their functionalities from services in the service space. Above LCSs is a formal model, which is expected to provide the sufficient semantic support for the change management. Based on the formal model, change model is used to declaratively specify top-down changes that might occur in LCSs. The change management component modifies a LCS to react to the changes.

2.1 The Formal Model

The formal model is expected to provide the grounding semantics to support change management. It includes a LCS schema and a Web service ontology.

A LCS schema gives a high-level overview of a LCS, such as the functionality it provides and the performance it delivers. It helps software agents capture the important information about a LCS, such as what it offers, how it works, the environment it works with, and how well it performs. It then helps software agents understand what a change is intended to make to a LCS and how to modify the LCS to react to the change. A LCS schema defines
a LCS’s *functional* features (i.e., functionality) and *non-functional* features (i.e., context and performance). The description of a LCS’s functionality contains the information about the types of services that a LCS outsources its functionality from and the collaborative relationships between these services. We call a type of services as an *abstract service*. An abstract service specifies one type of functionality provided by Web services. Examples of such functionality include *airline*, *hotel*, and *car rental* in a travel domain. They are not bounded to any concrete services. In such a way, a LCS can always choose the “best” services to outsource.

An abstract service is defined in terms of a Web service ontology. A Web service ontology contains the semantic description of the common features of Web services within a domain. A Web service ontology also defines a taxonomy of service instances based on their functionality. In addition to functionality, a service ontology also models other features of Web services, such as their quality and context. A Web service ontology is offered by *ontology providers*, which can be knowledge experts in a domain. Ontology providers maintain and manage the domain-specific knowledge about the services and their relationships within a domain. The semantics contained in a Web service ontology can be used to define a LCS schema. Moreover, by the nature of an ontology, Web services can be classified into different categories.
based on their functionality. Therefore, a Web service ontology can be used as a meaningful organization of Web services.

2.2 Change Model

A change model is used to specify top-down changes. Changes may be specified in a literal way at first place, which is informal and sometimes vague. Examples of such change specifications include “increase a LCS’s profit” and “stop outsourcing the service that has a low reliability”. This type of specification obviously lacks sufficient formalization and semantics to support change management in a systematic way. Therefore, change specifications need to be unambiguous, formal, and disciplined, which are described as follows.

- **Unambiguous:** A change is always associated with a specific goal on a LCS. During the process of change management, the LCS will be modified and the goal will be reached ultimately. Therefore, a change specification should be unambiguous so that the goal can be deterministic.

- **Formal:** To improve the automation of change management, it is important that the software agents understand what change is intended to make to a LCS. Therefore, a change specification should contain machine-understandable semantics, such as pre-defined keywords and logic-base expressions.

- **Disciplined:** To ensure that a change is feasible, it is important that a change specification contains all the necessary information. For example, if a change requires to remove a service that has a low reliability, it needs to specify at what degree a service’s reliability is considered as low. Therefore, a change specification should be disciplined. For different types of changes, different types of information are required to be contained in a change specification.

2.3 Change Enactment

Change enactment is to implement the changes defined by change models. It consists of two subcomponents: change reaction and change verification.

Change reaction is the process of modifying a LCS to make its features conform to the requirement of the change. The features of a LCS include its functionality, performance,
and context. They are determined by the services that a LCS outsources and the way that the services cooperate with each other. Therefore, during the process of change reaction, a LCS may change the partnership of a service, i.e., starting or stopping a partnership. This will depend on whether the service offers the required functionality, performance, or context. A LCS may also change the way that it outsources its functionality, i.e., the collaboration between the outsourced services. A LCS schema gives a high-level overview of a LCS which facilitates a strategic and operational analysis on how to react to changes. Change reaction should be performed in a top-down fashion. More specifically, the modification should be started at the schema level and then propagated to the service instance level.

Change verification is the process of checking whether a LCS schema still defines a correct configuration once it has been modified after the phase of change reaction. For example, if a service is added to a LCS, it needs to ensure that the service can be invoked properly. If not, the LCS will be considered as having an incorrect configuration. Once an error is detected, the incorrect configuration will be fixed by modifying the LCS schema. To do this, the correctness criteria of a LCS’s configuration should be defined. The correctness criteria will then be used as a guidance for change verification. Since the correctness of a LCS needs to be verified from different perspectives, the order of examining different correctness criteria may affect the verification process itself. That is, the modification that is made to fix a wrong configuration may cause another wrong configuration. This type of situation needs to be avoided by carefully analyzing the impact of the examination order on the verification process.

2.4 Change Optimization

Change optimization is to optimize the result of change management. This phase focuses on the change that requires efficient selection and replacement of outsourced Web services.

A change may trigger the modification of a LCS’s functionality, performance, context, or all of them. It may then trigger the change of a LCS’s instance (i.e., re-orchestration of Web services). Thus, a LCS may add new services to fulfill a new functionality, deliver the desired performance, or contain a required context. Since the volume of Web services is expected to grow, the process of selecting new services from the service space needs to be optimized, so that the “best” service from a pool of competitive services will be selected. The Quality of Web Services (QoWS) has become a central criterion for differentiating competing service providers [84, 83].
To conform to the criterion of QoWS, Web service providers may make the promises about the QoWS they offer. The information about the QoWS of the services can be used for efficient service selection. Considering that Web services are autonomous (i.e., provided by independent service providers), highly volatile (i.e., low reliability), and a priori unknown (i.e., new or no prior history) [42, 49], Web service providers may fail partially or fully to deliver the promises they make on QoWS they offer. The trustworthy services should be then identified and selected. Therefore, change optimization should be performed at two phases. During the first phase, the services that have bad reputation will be filtered out. The remaining services are all considered as trustworthy services. During the second phase, the ones that provide the best QoWS will be selected from the trustworthy services.
Chapter 3

Web Service Ontology

In this chapter, we propose a Web service ontology which adds semantics to Web service description. The semantics it delivers are expected to enable automatic service selection and integration, which are necessary for the automation of managing changes during a LCS’s lifecycle.

3.1 Overview of the Web Service Ontology

As depicted in Figure 3.1, a Web service ontology consists of a set of service concepts. A service concept defines a type of Web services within a domain. It captures the common features of Web services. A service concept can be viewed as an abstract service, which can be instantiated by concrete Web services. By the nature of an ontology, it can classify Web services into different categories based on their functionalities. Using the service concepts of the ontology, Web services’ functionalities are defined in a way that is clear and unambiguous to software agents. For example, a Web service instance \( w \) can be classified with the “AirBooking” node of the ontology to imply that it is an air reservation service. This ontology also facilitates a LCS’s owner to establish its LCS. He can choose a node of this ontology as an abstract service to define the LCS schema.

A Web service ontology also captures the relationships between different types of services within a domain. There are two types of relationships: inheritance relationship and dependency relationship. Inheritance relationship is a common relationship between two concepts within an ontology. It lies between two service concepts if one inherits its features from another one. Dependency relationship lies between two types of services if one’s invocation
relies on another’s invocation. It exists between two types of services when the services are combined together.

In the sequel, we first give the definition of service concepts. We then describe the relationships between different concepts.

## 3.2 Service Concepts

In this section, we define a service concept which captures the common features of Web services within a domain. There are three major features of a Web service: *service functionality* ($F^S$), *service context* ($C^S$), and *service quality* ($Q^S$). $F^S$ specifies what a service offers to its users. $C^S$ constructs the environmental situation that a service operates with. $Q^S$ defines a set of parameters for service evaluation and selection. By using the information delivered by these three features, a LCS can determine whether to choose a service as its member (i.e. whether the service provides the desirable functionality with satisfactory context and quality). These three features are elaborated on in the following sections.
3.2.1 Functionality

A service functionality ($F_S$) is collectively delivered by a set of service 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 dependency relationships between different service operations. For example, an Airline service may provide several operations, such as user_login, flight_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 flight_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 actually affected by the outside with a set of input and responses with a set of output [4].

---

Figure 3.2: An example of an airline service ontology in OWL-S
Therefore, we can define a service functionality as a binary: service data \((D)\) and service operations \((OP)\). \(D\) is an essential aspect of the behavior of a service. A service can be viewed as a transducer that translates a set of input items into a set of output items \([4]\). Besides of these input \((I)\), output \((O)\) data, \(D\) also contains internal data \((N)\), which the service generates and consumes them by itself. \(OP\) consists of a set of operations and their dependency relationships.

There are several ontology languages such as OWL-S and WSMO \([17, 78]\) that we can use to describe the proposed ontology. Take OWL-S as an example. There is a mapping between the proposed service ontology and the OWL-S definition. A service data can be mapped to the property of hasInput, hasOutput, hasLocal. A service operation can be mapped to the definition of an atomic process. The relationship between two operations can be defined by a composite process in an OWL-S ontology.

Suppose that an airline service provides a flight ticket reservation functionality. It has the service data as follows:

\[ (I) = \{\text{user\_information, departure\_date, arrival\_date, departure\_city, arrival\_city}\} \]

\[ (O) = \{\text{flight\_information}\} \]

\[ (N) = \{\text{login\_confirmation, payment\_information}\} \]

The airline service also has the service operations as follows:

user\_Login: It takes user\_information as input and generates the output of login\_information and payment\_information.

flight\_Reservation: It takes the input of login\_confirmation, payment\_information, departure\_date, arrival\_date, departure\_city, arrival\_city and generates the output of flight\_information.

It can be specified in OWL-S as depicted in Figure 3.2.

### 3.2.2 Quality

The service quality \((Q^S)\) is the main criterion to evaluate a service \([20, 44, 48, 31]\). A Web service is always associated with a service in the real world (i.e. an airline company, a hotel, etc). Therefore, we define the service quality in twofold: general quality \((Q_G)\) and business quality \((Q_B)\).

- General quality consists of a set of domain-free parameters that measure the quality of
Table 3.1: Measurement of QoWS parameters

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Definition</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Latency</td>
<td>Time\textsubscript{process}(op) + Time\textsubscript{results}(op) where Time\textsubscript{process} is the time to process op and Time\textsubscript{results} is the time to transmit/receive the results</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>N\textsubscript{success}(op)/N\textsubscript{invoked}(op) where N\textsubscript{success} is the number of times that op has been successfully executed and N\textsubscript{invoked} is the total number of invocations</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>Availability</td>
<td>UpTime(op)/TotalTime(op) where UpTime is the time op was accessible during the total measurement time Total-Time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>Encryption</td>
<td>Equal to 1 iff messages are encrypted</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>Authentication</td>
<td>Equal to 1 iff consumers are authenticated</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>Reputation</td>
<td>The degree of confidence that a service will deliver the quality it promises, ranging from 0 to 1.</td>
<td>number</td>
</tr>
<tr>
<td>Business (travel domain)</td>
<td>Fee</td>
<td>Dollar amount to execute the operation</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>comfort_Lodge</td>
<td>The degree of comfort provided by a lodge service, ranging from 0 to 1</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>convenience_Internet</td>
<td>The degree of convenience of accessing internet, ranging from 0 to 1</td>
<td>number</td>
</tr>
</tbody>
</table>

the service that is consumed through the Web. It mainly presents the measurement of properties that are related to the access of an operation op of a Web service. Examples of these parameters include availability, reliability, and latency. The general quality also includes the security requirements that are compliant with an operation op of a Web service. Examples of the parameters include encryption, authentication, and reputation.

- Business quality consists of a set of domain-specific parameters that measure the quality of the service that is consumed in the real world. For example, in a travel domain, the parameters can be the fee that is charged to access a service, comfort\_Lodge for how comfort is a lodge service, and convenience\_Internet for how convenient the access to the internet.

We propose a QoWS model as depicted in Table 3.1. It is not meant to be exhaustive and can be extended. It gives the formal definitions of a set of representative quality parameters. Other quality parameters can be added to extend the current model.
3.2.3 Context

The context of a Web service ($C^S$) models the environmental situation that a service resides in. It can be any metadata that explicitly describes the meaning of the data items exchanged between Web services. For example, $context=(currency, dollars)$ defines a context of currency in dollars. Therefore, $C^S$ contains the important information for correctly interacting with a service and needs to be modeled in a formal way.

Since a service context can be of any type of metadata, there are many types of contexts, such as $location$, $time$, $currency$, $temperature$, $weight$, $length$, $users$, etc. Therefore, we define $C$ as a set of context types that are related to the service invocation. $C = \{c_1, \ldots, c_n\}$, where $c_i$ is a context type.

WS-context is an OASIS standards to manage the lifecycle of a Web service context [47]. In WS-Context, the context lifecycle is as follows. First, to initiate an activity, a service requests a new context from the WS-Context service via a begin message and specifies a timeline for sharing the context. The begin action will then return a begin message plus a context. In this case, the interaction with the service will be associated with the context delivered in a SOAP message [69]. The context-aware service interaction will be terminated once by timing out or by explicitly instructing the context service to end. Figure 3.3 shows an example of a SOAP message that delivers a context information.
3.3 Service Relationships

A Web service ontology also captures the relationships between different types of services within a domain. There are two types of relationships: inheritance relationship and dependency relationship. Inheritance relationship lies between two service concepts if one inherits its features from another one. Dependency relationship lies in between two types of services if one’s invocation relies on another’s invocation. We elaborate on these two types of relationships as follows.

3.3.1 Inheritance Relationship

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 important 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.

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. In addition, it may also have the properties that the parent node does not have. For example, 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 in Figure 3.4.

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, 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, 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, as depicted in Figure 3.4.

For example, a travel service can be classified as an airline service, a train service, or a car rental service, etc. Within the airline service category, services can be further classified as checking airline availability services, check-in flight services, booking airline services, and
3.3.2 Dependency Relationship

It is worth to note that although Web services are autonomous and they can be invoked independently, there may be some dependency constraint formed when they are combined together. The dependency needs to be enforced when invoking the services.

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 dependency relationship between \(S_1\) and \(S_3\).

The dependency relationships between different types of services within a domain are usually generated by domain experts. They are predefined and stored in a domain knowledge base. We can use a set \(DR\) to denote the set which contains a set of such dependency relationships. \(DR\) is always associated with a Web service ontology \(O\). We define a \(DR\) as follows.

**Definition 3.1.** Dependency Relationship (DR) – A DR is a set \(\{DR_1, DR_2, \ldots, DR_n\}\) which is associated with a Web service ontology \(O\), where \(DR_i\) is a triplet \(\{S_{i_1}, S_{i_2}, D_i\}\). \(S_{i_1} \in O, S_{i_2} \in O, D_i \subset S_{i_1}.Output, and D_i \subset S_{i_2}.Input\)
3.4 Service Ontology Query Infrastructure

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. Meanwhile, the dependency relationship between different types of services also needs to be retrieved for composing these services together. Therefore, there are four types of ontology queries that may be required as follows.

- **Functionality-based service query**: It is to traverse a Web service ontology to find a service node which offers a specified functionality. Since a service functionality has two facets: service operations and service data, there are two types of queries here: operation-based and data-based.

- **Non-functionality service query**: It is to traverse a Web service ontology to find a service node based on the non-functional features. It includes quality-based service queries and context-based service queries.

- **Dependency relationship query**: It is to check the dependency relationship between two given service nodes.

- **Web service query**: It is to find a list of Web services that offer the functionality defined by the given service node.

We will elaborate on the different types of queries as follows.

In an operation-based functionality query, the tree-like structure of an ontology is traversed to find the node that provides a specified operation. Considering the tree-like structure of the service ontology, we leverage *path expressions* as an effective tool to declaratively and efficiently query the service ontology [67]. Therefore, we support two types of operation-based functionality queries: by operation name and by a path expression.
Algorithm 1 Operation-based Query Using Names

Input: an operation op; a service ontology tree $T(r)$
Output: a service node $N_S$

1: $n=r$; find=false;
2: if op $\notin n.OP$ then
3: find=true;
4: end if
5: if find=false then
6: $N=get_{-}Is_{-}a_{-}Children(n);$  
7: $s=null;$
8: for all $n \in N$ do
9: $s=HBFS(op, T(n));$ // Heuristically breadth first search $s$ in subtree $T(n)$
10: if $s=null$ then
11: return ERROR; // Fail to find the specified service
12: end if
13: end for
14: return $s$;
15: end if
16: Function HBFS($op, T(n)$)
17: $N=get_{-}Children(n);$  
18: if op $\notin n.OP$ then
19: $N=get_{-}Is_{-}a_{-}Children(n);$  
20: if $N=null$ then
21: return null;
22: end if
23: for all $t \in N$ do
24: return HBFS($t, OP, T(t)$);
25: end for
26: end if
27: return $n$;

The process of performing an operation-based query using name is depicted in Algorithm 1. The input is a string which specifies the name of the required operation and a service ontology tree. The output is a service node.

The query process starts at the root node of the ontology tree. It first checks whether the current node contains the specified operation. If yes, it will return the current node. If not, it will perform a heuristic breadth first search on the child nodes of the current node. Put differently, the algorithm only searches the child nodes that potentially have the specified operation. To narrow down the query space, the query leverages the difference between the two types of inheritance relationships between a node and its child nodes. If the current node does not provide a specified operation, its child nodes with Has_of relationships will not provide the operation either. Therefore, the query will only be continued on the current node’s Is_a child nodes (Line 6-15). The computation complexity of this algorithm is $O(|N|)$, where $N$ is the number of the nodes in the service ontology tree.

An operation-based query can be also specified in terms of a path expression. Path expressions used in service ontology queries for the sake of change management are different
from those used in querying semistructured data and object-oriented databases [58, 35, 16]. Traditional path expressions result in a set of paths that match the specified patterns [3]. In contrast, a path expression in a service ontology query is used to return the functionalities (in terms of services and operations) required by the change reaction process. In what follows, we give our definition of path expressions, which will be used in the queries on the service ontology.

**Definition 3.1.** *Path Expression –* A path expression $\mathcal{P}E$ is an expression over an alphabet $\Sigma$. $\Sigma$ is defined as the union of path variables and operations, i.e., $\Sigma = \mathcal{C} \cup \mathcal{S} \cup \mathcal{OP}$, where $\mathcal{C} = \{/ , //, [\] \}$, $\mathcal{S}$ is the set of service nodes in the ontology, and $\mathcal{OP}$ represents the set of operations offered by the service nodes. Specifically, $\mathcal{C}(\mathcal{P}E_i)$ represents the set of path variables in $\mathcal{P}E_i$, $\mathcal{S}(\mathcal{P}E_i)$ represents the set of services in $\mathcal{P}E_i$, and $\mathcal{OP}(\mathcal{P}E_i)$ represents the set of operations in $\mathcal{P}E_i$. ‘/’ and ‘//’ take the standard semantics of the path expressions. ‘[’ is used to specify the operation that is required to retrieved from the service node. A path expression is always expected to end with an operation name, which immediately follows the service node that offers the operation.

**Example 3.1.** A path expression $\mathcal{P}E$ that retrieves FlightStatus operation from the Airline service can be expressed as follows: $\mathcal{P}E = //\text{Transportation/Airline}[\text{FlightStatus}]$, where $\mathcal{C}(\mathcal{P}E) = \{/ , //, [\] \}$, $\mathcal{S}(\mathcal{P}E) = \{\text{Transportation, Airline}\}$, and $\mathcal{OP}(\mathcal{P}E) = \{\text{FlightStatus}\}$.

The process of performing an operation-based query using path expressions is depicted in Algorithm 2. 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 7-15), 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 breadth first search procedure (line 16-24 and line 32-42), 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 25-29).

$^{1}$‘/*’ is not used here because a path expression in service ontology queries always starts from the root node of the ontology tree.
Algorithm 2 Operation-based Query Using Path Expressions

Input: a service ontology query \( Q \) (a path expression); a service ontology tree \( T(r) \)
Output: a service node \( N_S \);

1: \( C = Q.C; S = Q.S; OP = Q.OP; \)
2: \( n = r; \)
3: while \( C \neq \phi \) do
4: \( c = C.pop(); \)
5: \( s = S.pop(); \)
6: if \( c=’/’ \) then
7: \( \text{find=false,} \)
8: if \( n.name \text{ matches } s \) then
9: \( N = n; \text{ find=true;} \)
10: end if
11: if \( \text{find==false then} \)
12: \( \text{return ERROR; // Fail to find the specified service} \)
13: end if
14: end if
15: if \( c=’/’ \) then
16: \( N = \text{get_Is_a_Children}(n) \)
17: for all \( n \in N \) do
18: \( R=HBFS(s, OP, T(n)); // \text{Heuristically breadth first search } s \text{ in subtree } T(n) \)
19: if \( R=\text{null then} \)
20: \( \text{return ERROR; // Fail to find the specified service} \)
21: end if
22: end for
23: end if
24: if \( c=’/’ \) then
25: if \( OP \in N.OP \) then
26: return \( N \); \)
27: end if
28: end if
29: end if
30: end while
31: Function \( \text{HBFS}(s, OP, T(n)) \)
32: if \( OP \notin n.OP \) then
33: \( N=\text{get_Is_a_Children}(n); \)
34: end if
35: for all \( t \in N \) do
36: if \( t.name \text{ matches } s \) then
37: return \( t; // \text{Find the service} \)
38: else \( \text{HBFS}(t, OP, T(t)) \)
39: end if
40: end for
41: return \( \text{NULL}; \)
42: end

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 3, 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 $D_O$). The matching criterion can be defined as: a node is matched if its output covers the required output, i.e., $S_O \supseteq D_O$, and its input can be covered by the given input, i.e., $S_I \subseteq D_I$.

**Algorithm 3 Data-based Service Functionality Query**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function CHECK($s, D_I, D_O, T(s)$)</td>
</tr>
<tr>
<td>2</td>
<td>if $s.I$ $\subseteq$ $D_I$ then</td>
</tr>
<tr>
<td>3</td>
<td>if $s.O$ $\supseteq$ $D_O$ then</td>
</tr>
<tr>
<td>4</td>
<td>$s = s'$; // Find the matched service node</td>
</tr>
<tr>
<td>5</td>
<td>$L_{OP} = s$.get_Operation_By_Output($D_O$)</td>
</tr>
<tr>
<td>6</td>
<td>return $L_{OP}$;</td>
</tr>
<tr>
<td>7</td>
<td>end if</td>
</tr>
<tr>
<td>8</td>
<td>end if</td>
</tr>
<tr>
<td>9</td>
<td>if ($s.I$ $\subseteq$ $D_I$) $==$ true then</td>
</tr>
<tr>
<td>10</td>
<td>if ($s.O$ $\supseteq$ $D_O$) $==$ false then</td>
</tr>
<tr>
<td>11</td>
<td>$C_L = \text{get}<em>{-\text{Is}</em>{-a}}_{\text{-Children}}(s)$;</td>
</tr>
<tr>
<td>12</td>
<td>for all $s' \in C_L$ do</td>
</tr>
<tr>
<td>13</td>
<td>return CHECK($r, D_I, D_O, T(r)$);</td>
</tr>
<tr>
<td>14</td>
<td>end for</td>
</tr>
<tr>
<td>15</td>
<td>end if</td>
</tr>
<tr>
<td>16</td>
<td>end if</td>
</tr>
<tr>
<td>17</td>
<td>if ($s.I$ $\subseteq$ $D_I$) $==$ false then</td>
</tr>
<tr>
<td>18</td>
<td>if ($s.O$ $\supseteq$ $D_O$) $==$ true then</td>
</tr>
<tr>
<td>19</td>
<td>$C_L = \text{get}<em>{-\text{Has}</em>{-of}}_{\text{-Children}}(s)$;</td>
</tr>
<tr>
<td>20</td>
<td>for all $s' \in C_L$ do</td>
</tr>
<tr>
<td>21</td>
<td>return CHECK($s', D_I, D_O, T(s)$)</td>
</tr>
<tr>
<td>22</td>
<td>end for</td>
</tr>
<tr>
<td>23</td>
<td>end if</td>
</tr>
<tr>
<td>24</td>
<td>end if</td>
</tr>
<tr>
<td>25</td>
<td>return ERROR; // Fail to find a matched service node</td>
</tr>
</tbody>
</table>

The query starts at the root node. It first checks whether the current node matches the requirement. If yes, it will return the current node as the result (Line 2-8). 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 9-16). 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 17-24). By leveraging the two types of relationships to guide the search, the algorithm performs more efficiently by only checking the potential nodes that provide the specified service data.
The non-functionality service queries is to find a service node based on its non-functional features in a Web service ontology tree. Therefore, these queries can be quality-based service query and context-based service query.

The quality-based service query is to traverse a Web service ontology tree to retrieve a node that supports a specified quality parameter. The query is performed in a similar way as the functionality-based service query. It starts at the root node of the tree. The query then performs a heuristic breadth first search on the child nodes of the current node to find the desired service node. The query space will be narrowed down by ignoring the current node's Has_of child nodes.

The context-based service query is to traverse a Web service ontology tree to retrieve a node that supports a specified context type. The query is performed in the same way as the quality-based service query.

The dependency relationship query is to retrieve the dependency between two types of services in a Web service ontology tree.

**Algorithm 4 Retrieving Dependency Relationships**

<table>
<thead>
<tr>
<th>Input: two services (s_1) and (s_2); a dependency relationship set (DR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: a set of data (D);</td>
</tr>
<tr>
<td>1: for all (1 \leq i \leq n) do</td>
</tr>
<tr>
<td>2: (\text{if } DR_i.S_1\text{ matches } s_1.name \text{ AND } DR_i.S_2\text{ matches } s_2.name) then</td>
</tr>
<tr>
<td>3: (\text{return } DR_i.D);</td>
</tr>
<tr>
<td>4: end if</td>
</tr>
<tr>
<td>5: end for</td>
</tr>
<tr>
<td>6: return null;</td>
</tr>
</tbody>
</table>

As depicted in Algorithm 4, the dependency relationship query is to go through the dependency relationship (DR) set and check whether there is a match between the elements in DR and the given service pair. If yes, the associated data set will be returned (Line 2-3). 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 and the Web service list.
Chapter 4

A Supporting Infrastructure of a LCS

In this chapter, we propose a supporting infrastructure of a LCS. We first give an overview of a LCS’s architecture. It mainly consists of two key components: LCS schema and LCS instance. It also contains two supporting components: ontology providers and Web service providers. A LCS schema is the kernel of a LCS since it defines its high-level business logic. It guides the composition of outsourced Web services to perform the functionality of the LCS. We give the formal definition and analyze the correctness of the schema.

4.1 An Overview of a LCS

Figure 4.1 depicts the architecture of a LCS. There are two key components and two supporting components in this architecture. The key components include a LCS schema and a LCS instance. The two supporting components include ontology providers and Web service providers.

- **LCS schema**: A LCS schema consists of a set of abstract services and the relationships among these services. An abstract service specifies one type of functionality provided by the Web services. They are not bounded to any concrete services. They are defined in terms of service concepts in a Web service ontology.

- **LCS instance**: A LCS instance is a composition of a set of concrete services, which instantiates a LCS schema. It actually delivers the functionality and performance of a LCS.
• **Ontology providers**: The ontology provider manages and maintains a set of ontologies that describe the semantics of Web services. A LCS outsources semantics from an ontology provider to build up its schema.

• **Web service providers**: The Web service providers offer a set of Web services, which can be outsourced to form LCS instances.

The underpinning of the proposed LCS architecture is a standard Service Oriented Architecture (SOA) [2]. The service providers use WSDL to publish Web services [73]. The WSDL description specifies the address and the arguments to invoke the service. Web service registries, such as UDDI, can be used as a directory for a LCS to look for Web services [70]. After locating a Web service, SOAP messages are exchanged between a LCS and the service providers for invoking the service [69]. Beyond this, semantic Web service technologies can be used by the ontology providers to define their service ontology, such as OWL-S and WSMO [17, 78]. The composition between selected services can be defined using service orchestration languages, such as BPEL [33].
4.2 LCS Schema

A LCS can be considered as a special type of Web services. It then has the similar features as Web services, which include functionality (i.e., what it offers), quality (i.e., quality), and context (i.e., under what environment). The difference between a LCS and common Web services lies in that a LCS outsources its functionality from third-party service providers, instead of providing the functionality by itself. Therefore, it also needs to capture the collaboration between the outsourced Web services within a LCS.

A LCS schema is the basis for specifying a change. Moreover, once there is a change required to be made to a LCS, such as its functionality, it will be more efficient to start at the schema level than directly go to the instance-level where concrete Web services and detailed orchestration are concerned. In this section, we define a LCS schema, which describes a LCS’s features at a high-level. We use a schema graph to capture the functionality of a LCS as well as the collaboration between the outsourced services. We then formally define the other two features of a LCS: quality and context.

4.2.1 Schema Graph

A LCS outsources its functionality from individual and autonomous services. Therefore, its functionality can be specified by two types of information: (1) the services it outsources (2) the composition of the services. It is worth to note that Web services are autonomous and independent. They may change their features, or come and go at their will. Considering this dynamic situation, we define a LCS’s functionality using a set of abstract services (i.e., the service concepts in a Web service ontology), instead of using concrete Web services. Each abstract service corresponds to one type of functionality, such as airline service, hotel service, etc. The composition of different services specifies how they affect each other by exchanging messages. It can be defined in terms of a data flow and a control flow. The data flow specifies the data transfer between different services. The control flow specifies the invocation order among the services. Therefore, we can use a directed graph (DG) to define a LCS schema, where nodes represent abstract services and edges represent the data flows and the control flows. In particular, we define a LCS’s functionality (referred to as schema graph) as follows.

Definition 4.1: A LCS schema graph is the directed graph that has two types of edges $DG=\{N, DE, CE\}$, where:

$N$ is a set of nodes. $N=\{n_e, n_1, n_2, \ldots, n_n, n_\omega\}$, where $n_e$ and $n_\omega$ are two special nodes that represent the user of the LCS. $n_e$ only has outgoing edges and $n_\omega$ only has incoming edges.
Figure 4.2: An example of a LCS schema graph

$n_i$ represents an abstract service ($1 \leq i \leq n$).

$DE$ is a set of edges. $DE=\{de_1, de_2, ..., de_s\}$, where $de_i=\{n_f, n_t, d_i\}$ represents that $n_f$ sends a message containing data $d_i$ to $n_t$. $n_f, n_t \in N$. If $n_f$ is $n_ε$, it means the data $d_i$ is part of the input of a LCS getting from the users. If $n_t$ is $n_ω$, it means the data $d_i$ is part of the output of a LCS returning to the users.

$CE$ is a set of edges. $CE=\{ce_1, ce_2, ..., ce_t\}$, where $ce_i=\{n_b, n_a, c_i\}$ represents that $n_a$ will be invoked after $n_b$ is invoked if condition $c_i$ is fulfilled. $n_b, n_a \in N$. If $n_b$ is $n_ε$, it means that the invocation of the LCS starts from invoking $n_a$. If $n_a$ is $n_ω$, it means that the invocation of the LCS ends with invoking $n_b$.

Figure 4.2 shows the schema of the travel agency LCS in our running example. The LCS provides a comprehensive travel package by outsourcing functionality from an Airline service, a Hotel service, a Taxi service, a Weather service, and a Payment service. Users are expected to provide their personal information (i.e., user_information), such as their names and billing addresses. They also need to provide the trip information (i.e., travel_information), such as the departure date, arrival date, departure place, and arrival place. An Airline service will be invoked first. After that, it will send the flight information to the Hotel service and the Taxi service. The hotel service will be invoked. After that, it will send its output which contains the hotel location information to the Taxi service. The Taxi service will take the arrival time of the flight, the location of the airport, and the location of the hotel as input. It then processes the taxi reservation which contains the reservation time and the fee based on the input. After that, the Payment service will be invoked. It receives the data from the three services (i.e., the Airline, Taxi, and Hotel services) generating the fee information from each
of them and charges the payment. It then generates the payment confirmation information which will be returned to the user. The Weather service will be invoked after the invocation of the Hotel service since it will take the hotel location and reservation time as its input and generate the weather information as its output.

4.2.2 LCS Quality

The quality of a LCS consists of a set of quality parameters, such as reliability, fee, invocation duration, reliability, etc. These parameters constitute a quality model that is used to evaluate how well a LCS performs. The quality model is domain-specific. A LCS outsources its functionality from multiple services. Meanwhile, it also uses quality models from these services. Therefore, we define a LCS’s quality model as follows.

**Definition 4.2 LCS Quality:** A LCS’s quality is a set $Q = \{q_1, q_2, ..., q_n\}$, where $q_i$ is a quality parameter. Meanwhile $Q \in (\bigcup_{n_i \in N} n_i.Q)$.

Since a LCS’s quality is actually delivered by the Web services it outsources, the quality thus can be determined by these services. Since a LCS instance contains multiple services, we need to aggregate the QoWS parameters from different services. Table 4.1 lists the aggregate functions for the QoWS parameters.

<table>
<thead>
<tr>
<th>QoWS parameter</th>
<th>Aggregation function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>$\sum_{i=1}^{n} latency(ws_i)$</td>
</tr>
<tr>
<td>Reliability</td>
<td>$\prod_{i=1}^{n} rel(ws_i)$</td>
</tr>
<tr>
<td>Availability</td>
<td>$\prod_{i=1}^{n} av(ws_i)$</td>
</tr>
<tr>
<td>Fee</td>
<td>$\sum_{i=1}^{n} fee(ws_i)$</td>
</tr>
<tr>
<td>Encryption</td>
<td>$\frac{1}{n} \sum_{i=1}^{n} enc(ws_i)$</td>
</tr>
<tr>
<td>Authentication</td>
<td>$\frac{1}{n} \sum_{i=1}^{n} aut(ws_i)$</td>
</tr>
<tr>
<td>Non-repudiation</td>
<td>$\frac{1}{n} \sum_{i=1}^{n} nrep(ws_i)$</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>$\frac{1}{n} \sum_{i=1}^{n} con(ws_i)$</td>
</tr>
</tbody>
</table>

4.2.3 LCS Context

The context of a LCS consists of a set of context types, such as location, time, user, travel type, etc. The context types structure the environment that affects how a LCS performs its
functionality. They contain important information for the interaction between a LCS and its users. Since a LCS outsources its functionality from multiple services, its context structure can be determined by these services. Therefore, we define a LCS’s context as follows.

**Definition 4.3 LCS Context:** A LCS’s context is a set \( C = \{c_1, c_2, ..., c_n\} \), where \( c_i \) is a context type. Meanwhile \( C \in (\bigcup_{n_i \in N} n_i.C) \).

### 4.3 Schema Correctness

A LCS schema defines its configuration with respect to its functionality. It needs to be correct so that the LCS can function well. For example, an outsourced service should make a direct or indirect contribution to the overall functionality of a LCS. Otherwise, it will be meaningless to be included. It should be guaranteed that it can retrieve all its required input to be invoked. In this section, we define a set of criteria for a LCS schema from both structural and semantic correctness perspectives.

#### 4.3.1 Structural Correctness

We define a LCS’s functionality as a directed graph, which consists of a set of nodes and two types of edges. From a structural point of view, we define the correctness of a LCS as follows.

**Correctness Criterion 1** – Let \( \mathcal{M} \) be a LCS schema. If there is a node \( n \in \mathcal{M}.DG.N \) and \( n \) does not have any incoming or outgoing edges, \( \mathcal{M} \) is not a correct schema.

The criterion means that there should not be any isolated node in a LCS schema graph. A LCS schema contains two sets of edges: \( DE \) and \( CE \). These edges reflect the relationships between different services, i.e., how they are combined together. If there is a node that is not connected to any other node, the node has no interactions with others. Therefore, the LCS schema is incorrect.

**Correctness Criterion 2** – Let \( \mathcal{M} \) be a LCS schema. If there is a node \( n \in \mathcal{M}.DG.N \) and there is not an edge \( e \), where \( e \in \mathcal{M}.DG.CE \) and \( e \) goes to \( n \), \( \mathcal{M} \) is not a correct schema.

The criterion means that there should not be any node in a LCS schema graph that there is no incoming \( CE \) edge. Edges in \( CE \) show the invocation order among different services. An incoming \( CE \) edge of a node \( n \) means that \( n \) should be invoked after the invocation of the node where the edge comes from. For a service that needs to be invoked first, it will have
an incoming \( CE \) edge comes from \( n_e \), which represents the user of the LCS. Therefore, if a node does not have a \( CE \) incoming edge, it cannot be invoked. In this case, the LCS schema is incorrect.

**Correctness Criterion 3** – Let \( \mathcal{M} \) be a LCS’s schema. If there is a node \( n \in \mathcal{M}.DG.N \) and there is not an edge \( e \), where \( e \in \mathcal{M}.DG.DE \) and \( e \) comes from \( n \), \( \mathcal{M} \) is not a correct schema.

The criterion means that there should not be any node in a LCS schema graph that there is no \( DE \) edge that comes from it. Edges in \( DE \) show the data transfer among different services. An incoming \( DE \) edge of a node \( n \) shows that \( n \) will get input from the node where the edge comes from. An outgoing \( DE \) edge of \( n \) shows that \( n \) will generate the output and send it or part of it to the node where the edge goes to. Each service in a LCS should make a direct or indirect contribution to the overall output of a LCS. Therefore, a node should send its output to other services or the user of a LCS (i.e., \( n_\omega \)). Otherwise, \( \mathcal{M} \) is not a correct schema.

### 4.3.2 Semantic Correctness

Each node represents an abstract service in a LCS schema graph. The abstract services are described by a Web service ontology. The ontology contains sufficient semantics such as the input/output of a Web service for automatic service description, discovery, and composition [17, 78]. A LCS schema should be correct from the semantic point of view. We define the semantic correctness as follows.

**Correctness Criterion 4** – Let \( \mathcal{M} \) be a LCS schema. If there is a node \( n_i \in \mathcal{M}.DG.N \) and one of its input data item \( d \) does not be fed by other nodes, \( \mathcal{M} \) is not a correct schema.

The criterion means that each service in a LCS should have all its input fed by other services or the user of the LCS. Otherwise, the LCS schema is not a correct schema. A service is expected to be invoked by a set of messages, which contain the necessary information for the invocation. For example, the information about the date and location is always required to access an airline service. Therefore, the input of a service should be covered by its incoming \( DE \) messages so that it can be invoked. Otherwise, \( \mathcal{M} \) is not a correct schema.

**Correctness Criterion 5** – Let \( \mathcal{M} \) be a LCS schema. If there is a \( DE \) edge between \( n_i \) and \( n_j \), there should be a \( CE \) path between \( n_i \) and \( n_j \). Otherwise, \( \mathcal{M} \) is not a correct schema.

The criterion means that the control flow should be consistent with the data flow. If a node \( n_i \) needs an input from another node \( n_j \), the invocation of \( n_i \) should depend on the invocation
of \( n_j \). There are two basic types of service compositions: horizontally and vertically [41]. If \( n_i \) and \( n_j \) are combined horizontally, the invocation of \( n_i \) should be after the completion of the invocation of \( n_j \). If \( n_i \) and \( n_j \) are combined horizontally, \( n_i \) and \( n_j \) need input from each other. In this case, there should be an invocation cycle between \( n_i \) and \( n_j \). \( n_i \) may be first invoked, it then send messages to \( n_j \) and \( n_j \) will be invoked. After that, \( n_j \) will generate output and send messages to \( n_i \). \( n_i \) will continue to be invoked. \( \mathcal{M} \) should guarantee that the order of data transfer and service invocation are the same. Otherwise, it is not a correct schema.

The summary of the above criteria is depicted as Table 4.2.

<table>
<thead>
<tr>
<th>id</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>structural</td>
<td>(( \exists n )(( n \in \mathcal{M}.DG.N ) \land \neg (( \exists e )(( e \in (\mathcal{M}.DG.DE \lor \mathcal{M}.DE.CE) ) \land (( e.from = n ) \lor (e.to = n))))) \Rightarrow \mathcal{M} ) is not a correct schema.</td>
</tr>
<tr>
<td>2</td>
<td>structural</td>
<td>(( \exists n )(( n \in \mathcal{M}.DG.N ) \land \neg (( \exists e )(( e \in \mathcal{M}.DG.CE ) \land (e.to = n))))) \Rightarrow \mathcal{M} ) is not a correct schema.</td>
</tr>
<tr>
<td>3</td>
<td>structural</td>
<td>(( \exists n )(( n \in \mathcal{M}.DG.N ) \land \neg (( \exists e )(( e \in \mathcal{M}.DG.DE ) \land (e.from = n))))) \Rightarrow \mathcal{M} ) is not a correct schema.</td>
</tr>
<tr>
<td>4</td>
<td>semantic</td>
<td>(( \exists n )(( n \in \mathcal{M}.DG.N ) \land (( \exists d )(( d \in n.input ) \land \neg (( \exists e )(( e \in \mathcal{M}.DG.DE ) \land (e.to = n) \land (d \in e.data)))))) \Rightarrow \mathcal{M} ) is not a correct schema.</td>
</tr>
<tr>
<td>5</td>
<td>semantic</td>
<td>(( \exists e )(( e \in \mathcal{M}.DG.CE ) \land \neg (( \exists p )(( p \in Path(\mathcal{M}.DG.DE) ) \land (p.start = e.from) \land (p.end = e.to)))) \Rightarrow \mathcal{M} ) is not a correct schema.</td>
</tr>
</tbody>
</table>
Chapter 5

The SCML Language

Changes need to be machine comprehensible so that they can be automatically and correctly enacted. Therefore, a change specification should achieve the following: (1) It should be unambiguous about what a change intends to be implemented. (2) It should be formal so that it can be processed by machines. (3) It should be disciplined to ensure that sufficient information is provided. In this chapter, we present a formal language, the Service Change Management Language (SCML), to specify top-down changes. Since a change introduces new requirements on the LCS’s features, we first present a taxonomy that classifies top-down changes based on their requirements. The taxonomy is built on top of the proposed LCS schema. Based on the change taxonomy, we then present the SCML that enables the formal specification of changes.

5.1 Change Taxonomy

A primary task to manage top-down changes in a LCS is to identify a clear classification of these changes. Thus, different reaction policies can be developed to deal with different types of changes. As depicted in Figure 5.1, we use change requirement as a dimension to classify changes. The change requirement reflects the purpose of introducing a change, which could be a result of business policies, business regulations, laws, or just the intension of LCS owners.

Using change requirement as a dimension, changes can be classified based on the key features of a LCS. This conforms to the classical change taxonomy approaches from the fields of software engineering and workflow systems [62, 40]. A LCS outsources the functionality from
multiple Web services. Each component service offers a type of functionality. A long-term relationship is formed among the component services for their cooperation. The features of a LCS can be classified into *functional* and *non-functional*. The functional feature refers to the *functionality* of a LCS. The non-functional features include *context* and *quality*. Top-down changes are expected to modify one or more of the features of a LCS. Therefore, we classify changes based on these features. We elaborate on each type of changes as follows.

### 5.1.1 Functional Changes

Functional changes are those that require to modify the functionality of a LCS. A LCS’s functionality is specified by two types of information: the *abstract services* it outsources and their *composition*. 
5.1.1.1 Changes of Abstract Services

A LCS may change the type of services it outsources for the purpose of enacting a business policy. The change includes adding, removing, and replacing a functionality. This could happen for the purpose of fulfilling three types of requirements: functional requirements, context type requirements, and quality model requirements.

**Functional Requirement:** A LCS’s outsourced services may be changed to fulfill a functional requirement. It can be adding a service to the business model to enrich its functionality. For example, a travel agency may need to outsource a Point-Of-Interest (POI) service to attract more customers. A LCS may also want to remove a service from its business model. For example, consider that a travel agency LCS may outsource an airline service, a hotel service, a train service, a taxi service, and a car rental service. Suppose that the train service does not make satisfactory profit for the LCS. In this case, it may need to be removed from the LCS. A LCS may also want to replace a service in its business model. For example, a travel agency LCS may use an online payment system for reserving a trip. The payment system only supports online bank wire transfer. Considering that users may prefer to use credit card for online transactions, a credit card supported payment service will be used to replace the original one.

As we defined in a Web service ontology, a functionality has two facets: operations and data. For the first facet, the intended service should provide the specified operations. An example of such a change is “adding a service that provides flight status checking operation”. For the second facet, the intended service should provide the ability of transducing data. Put differently, it should be able to generate the specified output by using the given input. An example of such a change is “adding a service that can generate the weather information given a zip code”. We define a functional requirement as follows.

**Definition 5.1 Functional Requirement:** A functional requirement \((f)\) is a triplet \((OP, D_I, D_O)\), where \(OP\) is a set of operations that a service should provide, \(D_I\) and \(D_O\) are two sets of data items stating that a service should be able to generate \(D_O\) by using \(D_I\).

**Context Type Requirement:** A LCS’s outsourced services may be changed due to a new context type requirement. Each abstract service is associated with a set of context types, which constitute the environment structure of the service. Suppose that a LCS is required to support a new context, such as history data. It then needs to ensure that each outsourced service is able to embed the history data information in the SOAP message during the interactions. This may trigger the change of “removing the service that does not support a context type of history data”.
Quality Model Requirement: A LCS’s outsourced services may be changed due to a new quality requirement. Each abstract service is associated with a quality model, which includes the parameters for service evaluation. For example, a top-down change may require a new quality parameter to evaluate the outsourced services, such as privacy. This may trigger the change of “removing the service that does not include privacy in its quality model”.

Definition 5.2 LCS outsourced abstract service change: An outsourced functionality change of a LCS \((FM)\) is a triplet \(\{F^+, F^-, F^R\}\), where \(F^+\) is a set of abstract services representing the functionality that will be added to the LCS, \(F^-\) is a set of abstract services representing the functionality that will be deleted from the LCS, and \(F^R\) is a set of abstract service pairs. Each pair \(<s, s'>\) corresponds to a service replacement, where \(s\) will be replaced by \(s'\).

5.1.1.2 Changes of Service Composition

A LCS’s composition defines how it performs its functionality. It specifies the coordination of the outsourced services in a LCS.

A LCS’s composition may change under two situations. First, when a new service is added to a LCS or a service is deleted from a LCS, a composition change will be introduced. For example, when adding a payment service to a travel agency LCS, it needs to be combined with other services. Second, a LCS’s owner may want to change the way that the component services are combined together for some purpose, such as optimization. For example, suppose that a hotel service and a car rental service are invoked sequentially. There is no invocation dependency between them since they do not exchange messages with each other. In this case, the LCS’s owner may want to parallelize their invocation to decrease the overall duration time.

The change to a LCS’s composition can occur to both data transfer and invocation order. Change to data transfer among services includes the modification of user input, LCS output, adding or deleting a message between two services.

- **User input:** The user input is obtained from the user of a LCS. It contains the information that is necessary to invoke the services outsourced by the LCS. Once there is a change on the outsourced services, a change of the user input may be introduced. For example, when adding a car rental service, some information is required from the user to invoke the service, such as the car type (i.e., full size, compact, mid-size, economy, etc.). A change of the user input may also be introduced by a LCS’s owner.
For example, a travel agency LCS provides the airline+hotel package. In this package, the information about location and check in/out time is typically determined by the result of invoking the airline service. The owner may now want to change it by letting users provide these information. In this way, users can have more options when they choose their hotels.

- **LCS output:** The LCS output is generated by a LCS and returned to its users. It is contributed directly or indirectly by the services that the LCS outsources from. Once there is a change of the outsourced services, a change of the LCS output may be introduced. For example, when adding a car rental service, the LCS will generate more information, such as the pick up/drop off location, time, date, and charges. A change of LCS output may also be introduced by a LCS’s owner. For example, a travel agency LCS is used to generate the weather information. The owner may want to stop providing such information in the future.

- **Message exchange:** The message exchange is performed between outsourced services in a LCS. A Web service is interacted by its users or partners completely by message exchange. It is invoked by an input message and reacts to the message with an output message. We define a message as follows.

**Definition 5.3 Message:** A message \((m)\) is a tuple \(
\{(s^f, s^t, D)\}\), where \(s^f\) is the service that the data comes from, \(s^t\) is the service that the data goes to, and \(D\) is a set of data items delivered.

Once there is a change of the outsourced services, a change of the message exchange between services may be introduced. For example, when adding a traffic service to a travel package, the LCS owner may want to add the message exchanges from the airline service and the hotel service to the traffic service so that it can generate the corresponding driving direction between the airport to the hotel.

Changes to the invocation order refer to adding a process constraint, which can be sequential, parallel, outsourcing, or conditional selection. Figure 5.2 shows these four types of process constraints. Their definitions are given as below.

**Definition 5.4 Process Constraints:** A process constraint \((p)\) can be either one of the follows:

- **Sequential Constraint:** \(P^{>>}(s_1, s_2)\) means that \(s_1\) is invoked before the invocation of \(s_2\). It usually exists between two services where one service requires the result of another service’s invocation.
- **Parallel Constraint:** \( P|| (s, s_1, s_2) \) means that \( s \)'s invocation is in parallel with the invocation block from \( s_1 \) and \( s_2 \). It usually exists between the services where there are no message exchanges between them.

- **Outsourcing Constraint:** \( P\triangleleft (s_1, s_2) \) means that \( s_1 \) outsources functionality from \( s_2 \). It is always introduced by adding a new service (i.e. \( s_2 \)) to enrich another service's (i.e., \( s_1 \)) functionality. For the sake of simplicity and without the loss of generality, we assume that \( s_2 \) does not have any interaction with other services in a LCS except for \( s_1 \).

- **Conditional Selection Constraint:** \( P\? ((c_1, s_1), \ldots, (c_n, s_n)) \) means that if \( c_i \) is fulfilled, \( s_i \) will be invoked, where \( 1 \leq i \leq n \). It always exists among different services which provide the similar functionality in a coarse granularity. Examples of such functionality include taxi services and car rental services, which both provide the ground transportation.

Adding a new service will naturally introduce the changes to the invocation order among component services in a LCS. The invocation order between the new service and the other services may be specified by the owner of a LCS. It is worth to note that some invocation order for new services can also be automatically generated. It will be determined by the owner of a LCS whether it is necessary to specify the invocation order for new services or not. For the changes of the invocation order between two existing services, “adding” actually does a “replacing” work here. More specifically, when adding a new process constraint on the invocation order between two services, the previous one will be deleted to avoid the conflicts. For example, if the owner of an travel agency LCS wants to change the invocation order of the hotel service and the car rental service from sequential to parallel, he needs to first delete the sequential constraint between these two services and then add a parallel constraint.

We define a composition change as follows.

**Definition 5.5 LCS composition change:** A composition change of a LCS \((FM)\) is a tuple \( \{I^+, I^-, O^+, O^-, MX^+, MX^-, P^+\} \), where \( I^+ \) is a set of input data that will be added to the LCS, \( I^- \) is a set of input data that will be deleted from the LCS, \( O^+ \) is a set of output data that will be added to the LCS, \( O^- \) is a set of output data that will be deleted from the LCS, \( MX^+ \) is a set of data transfers that will be added to the LCS, \( MX^- \) is a set of data transfers that will be deleted from the LCS, and \( P^+ \) is a set of process constraints that will be added to the LCS.
5.1.2 Non-functional changes

Non-functional changes are those that require to change the non-functional features of a LCS, including context and performance changes.

The context of a LCS specifies its environmental information, such as location, time, historical transactions, and etc. A top-down change may require to change the context of a LCS and/or its outsourced services. For example, a travel agency LCS may change its location to another state. In this case, location-sensitive policy may affect the LCS’s function. An example of such a policy is the tax rate. A top down change may also require to change the context of a LCS’s outsourced service. For example, a travel agency LCS may want to change the location of the airline service to take advantage of a new policy. We define a context change as follows.

**Definition 5.6 LCS Context Change:** A context change of a LCS (CM) is a set \( \{\lambda_1, \lambda_2, ..., \lambda_m\} \), where \( \lambda_i \) is a context constraint that is enforced on the outsourced services. It can be specified as \( \{S, e, v\} \), where \( S \) is a set of outsourced abstract service of a LCS, \( e \) specifies the context type, and \( v \) specifies the new value of the context.

The quality of a LCS refers to its non-functional features, such as its reliability, fee, invocation duration, reputation, etc. It evaluates the quality delivered by a LCS. A top-down change may require to modify the quality that a LCS delivers. For example, a LCS’s owner may want to guarantee that the providers of its component services should have a decent reputation.
We define a quality change as follows.

**Definition 5.7 LCS Quality Change:** A quality change of a LCS (QM) is a set \( \{ \delta_1, \delta_2, ..., \delta_n \} \), where \( \delta_i \) is a quality constraint that is enforced on the outsourced services. It can be specified as \( \{ S, i, r, \} \), where \( S \) is a set of outsourced services of a LCS, \( i \) specifies the quality parameter, and \( r \) indicates the requirement on these services with respect to \( i \).

A top-down change can be thus modeled as the modification of the LCS’s features, where we summarize as follows.

### Table 5.1: Top-down change model

<table>
<thead>
<tr>
<th>Top-down Changes</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outsourced Abstract Services Change</td>
<td>( FM = (F^+, F^-, F^R) ).</td>
</tr>
<tr>
<td>Composition Change</td>
<td>( OM = (O^+, O^-, I^+, I^-, M^+, M^-, P^+) ).</td>
</tr>
<tr>
<td>Context Change</td>
<td>( CM = (\lambda_1, \lambda_2, ..., \lambda_m) ).</td>
</tr>
<tr>
<td>Quality Change</td>
<td>( QM = (\delta_1, \delta_2, ..., \delta_n) ).</td>
</tr>
</tbody>
</table>

### 5.2 Change Operators

Based on the proposed change taxonomy, we present two types of change operators: functional and non-functional. We elaborate on these operators in the sequel.

#### 5.2.1 Functional Change Operators

The functionality changes may occur to a LCS’s outsourced abstract services (\( FM \)) and their composition (\( OM \)). We use \( M \) to denote a LCS’s schema. \( FM \) consists of three sets, including the abstract services that are intended to be added (\( F^+ \)), removed (\( F^- \)), and replaced (\( F^R \)). The selection of these abstract services is based on the three types of requirements: functional requirement, context type requirement, and quality model requirement. Therefore, we define the change operators for selecting abstract services as follows.

- \( \Pi^F_{op}(op, O) \): It will traverse the service ontology \( O \) to find the abstract services that provide the specified operation \( op \). This operator takes \( op \) and \( O \) as input and returns an abstract service.
• \( \Pi^F_d(D_I, D_O, O) \): It will traverse the service ontology \( O \) to find the abstract services that can generate the required output of \( D_O \) by using the given input \( D_I \). This operator takes two sets of data, \( D_I, D_O \) as well as a service ontology \( O \) as input and returns an abstract service.

• \( \Pi^C(c, O) \): It will traverse the service ontology \( O \) to find the abstract services that support a context type \( c \). This operator takes \( c \) and \( O \) as input and returns a list of abstract services.

• \( \Pi^Q(q, O) \): It will traverse the service ontology \( O \) to find the abstract services that include a quality parameter \( q \) in its quality model. This operator takes \( s \) and \( q \) as input and returns a list of abstract services.

For the selected service node, we define two change operators as below.

• \( \Delta^S(s, O, M, op) \): It will perform the operation \( op \), by either adding or removing an abstract service to or from a LCS’s schema \( M \). This operator takes \( s, M \), and \( op \) as input and returns a new LCS schema as its output.

• \( \Delta^{S\rightarrow}(s_{old}, s_{new}, O, M) \): It will replace an abstract service \( s_{old} \) with another abstract service \( s_{new} \) in a LCS’ schema \( M \). This operator takes \( s_{old}, s_{new}, O \), and \( M \) as input and returns a new LCS schema as its output.

\( OM \) consists of several sets, including a set of input data that is intended to be added \((I^+)\) or removed \((I^-)\), a set of output data that is intended to be added \((O^+)\) or removed \((O^-)\), a set of data transfers that are intended to be added \((MX^+)\) or be removed \((MX^-)\), a set of process constraints that are intended to be added \((P^+)\). Therefore, we define the following change operators for them.

• \( \Delta^I(M, D, op) \): It will perform the operation \( op \), by either adding the data items in \( D \) to or removing some data items from a LCS’s input.

• \( \Delta^O(M, D, op) \): It will perform the operation \( op \), by either adding the data items in \( D \) to or removing some data items from a LCS’s output.

• \( \Delta^{MX}(m, M, op) \): It will perform the operation \( op \), by either adding or removing a data transfer \( m \) to or from a LCS’s schema \( M \). This operator takes \( m, M \), and \( op \) as its input and returns a new schema as its output.
• $\Delta_p^{>>} (s_1, s_2, M)$: It will add a sequential constraint on the invocation order defined in $M$. This operator takes $s_1, s_2, M$ as its input and returns a new schema as its output.

• $\Delta_p^\parallel (s, s_1, s_2, M)$: It will add a parallel constraint on the invocation order defined in $M$. This operator takes $s, s_1, s_2, M$ as its input and returns a new schema as its output.

• $\Delta_p^{-} (s_1, s_2, M)$: It will add an outsourcing constraint on the invocation order defined in $M$. This operator takes $s_1, s_2, M, \text{op}$ as its input and returns a new schema as its output.

• $\Delta_p^? (s, C, S, M)$: It will add a conditional selection constraint on the invocation order defined in $M$. $C$ is a set of condition expression and $S$ is a set of services. $C$ and $S$ have the same cardinality. This operator takes $C, S, M, \text{op}$ as its input and returns a new schema as its output.

5.2.2 Non-functional Change Operators

The non-functional changes include context change ($CM$) and quality change ($QM$). $CM$ consists of a set of context constraints. $QM$ consists of a set of quality constraints. We define two change operators for these two types of changes: $\Delta^{CM} (\lambda, M)$ and $\Delta^{QM} (\delta, M)$.

• $\Delta^{CM} (\lambda, M)$ It will enforce a context constraint $\lambda$ on a LCS with the schema $M$. $\lambda$ is a triplet $\{S, v, e\}$, where the services in $\lambda.S$ should have the value of $\lambda.v$ for the context $\lambda.e$. This operator takes $\lambda$ and $M$ as its input and returns a new LCS instance as its output.

• $\Delta^{QM} (\delta, M)$ It will enforce a quality constraint $\delta$ on a LCS with the schema $M$. $\delta$ is a triplet $\{S, r, i\}$, where the services in $\delta.S$ should have the value of $\delta.r$ for the quality parameter $\delta.i$. This operator takes $\delta$ and $M$ as its input and returns a new LCS instance as its output.

5.3 Change Language

We propose a Web Service Change Management Language (SCML) for the purpose of managing top-down changes. SCML is a SQL-like language. It defines five types of commands: (1) create command for defining a LCS schema; (2) select command for querying both abstract services and concrete Web services; (3) alter command for specifying functional changes; (4)
**update command** for specifying non-functional changes; (5) **drop command** for deleting a LCS schema. The commands are defined and elaborated on in this section.

### 5.3.1 Create Command

The create command is used to specify a new LCS schema. A LCS schema is given a name using two keywords: **CREATE** and **LCS**. For example, by writing

**CREATE LCS** travel-agency...

a LCS named as travel-agency is created. A LCS is associated with a Web service ontology from where it outsources semantics. Therefore, the Web service ontology is specified first. We use a keyword **ONTOLOGY** to specify the ontology provider that offers the ontology. For example, by writing

**ONTOLOGY** o http://wsms-dev.csiro.au:8080/axis2/services/OntologyAccessWithConfig

a LCS is associated with a ontology service which provides ontological semantics for the LCS.

After that, the abstract services in a LCS is specified. Each abstract service corresponds to a service concept in the Web service ontology. It is then described using the name of the service concept. We use the keyword, **SERVICES**, to specify one or more abstract services. For example, by writing

**SERVICES** s\(_a\) airline, s\(_t\) taxi, s\(_h\) hotel

**SERVICES** s\(_p\) payment

we specify four abstract services for the LCS.

We use a keyword, **CONTROL FLOWS**, to specify one or more control flow edges in a LCS schema graph. Each edge is given a name and a description. The description includes the information about the service node that the edge comes from, the service node the edge goes to, and the condition the edge delivers. For example, by writing

**CONTROL FLOWS** c\(_1\) (s\(_a\), s\(_h\), true), c\(_2\) (s\(_h\), s\(_t\), true)

we specify a control flow edge from the airline service to the hotel service.

We use a keyword, **DATA FLOWS**, to specify one or more data flow edges in a LCS schema graph. Each edge is given a name and a description. The description includes the information about the service node that the edge comes from, the service node that the edge goes to,
and the data item the edge delivers. For example, by writing

**DATA FLOWS** d1 \((s_a, s_p, \text{ticket\_price})\),

we specify a data flow edge from the airline service to the payment service with the information of a ticket’s price.

Recall that there are two special service nodes: \(n_\omega\) and \(n_\epsilon\), which refer to the user of a LCS. We use a keyword, **USER**, to specify these two service nodes when defining edges in a LCS schema graph. For example, by writing

**DATA FLOWS** d2 (USER, s1, user\_Id), d3 (s1, USER, flight\_schedule)

we specify two data flow edges. In d1, the information is obtained from a LCS’s users and sent to the airline service. In d2, the data is generated by the airline service and returned to users.

After specifying a LCS schema graph, we use a keyword, **QUALITIES** to specify one or more quality parameters that are used to evaluate a LCS. A quality parameter is given a name and a description. For example, by writing

**QUALITIES** q1 availability, q2 cost

we specify two quality parameters.

We use a keyword, **CONTEXTS** to specify one or more contexts of a LCS. A context is given a name and a description. For example, by writing

**CONTEXTS** c1 location, c2 time, c3 currency

we specify three contexts for the LCS.

Therefore, we can define a LCS schema as follows.

**CREATE LCS** travel-agency (  
  **ONTOLOGY** o http://wsms-dev.csiro.au:8080/axis2/services/OntologyAccessWithConfig  
  **SERVICES** s_a airline, s_t taxi, s_h hotel, s_p payment, s_w weather...
  ...
  **CONTROL FLOWS** c1 (s_a, s_h, true), c2 (s_h, s_t, true )...
  ...
  **DATA FLOWS** d1 (s_a, s_p, ticket\_price),
  ...
  **QUALITIES** q1 availability, q2 cost
  ...
5.3.2 Select Command

The select command is used to specify a query on a Web service ontology. The corresponding change operators include: \( \Pi^F_{op}(op, O) \), \( \Pi^F_d(D_I, D_O, O) \), \( \Pi^F_Q(c, O) \), and \( \Pi^F_C(q, O) \). A query can be performed based on the features of a LCS: functional and non-functional. Similar to a select statement in SQL, a SCML select command is formed of the three clauses, which start with three keywords: SELECT, FROM, and WHERE, respectively.

\[
\text{SELECT } <\text{abstract service list}> \\
\text{FROM } <\text{ontology}> \\
\text{WHERE } <\text{condition}>
\]

where \(<\text{abstract service list}>>\) is a list of abstract services that are intended to be retrieved by the query; \(<\text{ontology}>>\) is the Web service ontology that the query is performed upon; and \(<\text{condition}>>\) is a conditional expression (Boolean) that identifies the services to be retrieved by the query. In SCML, a conditional expression has the following format:

\[
<\text{abstract service}> <\text{operator}> <\text{values}>
\]

The operators include \texttt{hasOperation}, \texttt{hasInput}, \texttt{hasOutput}, \texttt{hasQuality}, and \texttt{hasContext}. They are defined for the four change operators that require a query on a Web service ontology. For each of these change operators, we give an example of a SCML query statement.

- \( \Pi^F_{op}(op, O) \): SELECT s FROM o WHERE s hasOperation (airline_reservation)
- \( \Pi^F_d(D_I, D_O, O) \): SELECT s FROM o WHERE s hasInput (location, date) and s hasOutput (weather_information)
- \( \Pi^F_Q(q, O) \): SELECT s FROM o WHERE s hasQuality (privacy)
- \( \Pi^F_C(c, O) \): SELECT s FROM o WHERE s hasContext (history_data)

5.3.3 Alter Command

The alter command is used to specify functional changes in a LCS. The possible alter LCS schema actions include (1) adding or deleting user input or LCS output, (2) adding, deleting,
or replacing abstract services and/or data flow edges), and (3) adding a process constraint.

For (1) and (2), the alter command is formed as:

\[
\text{ALTER LCS} <LCS\ name> <action> <element\ type> <value>
\]

where an action can be ADD, DELETE, or REPLACE. An element type can be INPUT, OUTPUT, SERVICES, and DATA FLOWS. When the action is REPLACE, the element type has to be SERVICES. The value type for REPLACE action is a pair. For other actions, the value contains a service name and the name of its corresponding service concept in the service ontology. The alter command corresponds to the five functional change operators. We give an example of a SCML alter command for each of them.

- \(\Delta^S\) (s, O, M, op): ALTER LCS travel-agency ADD SERVICES(s_f traffic, s_l local_activity, s_z address_to_zip);

- \(\Delta^{S\rightarrow}\) (s_\text{old}, s_\text{new}, O, M): ALTER LCS travel agency REPLACE SERVICES (s_t, s_c car_rental);

- \(\Delta^I\) (M, D, op): ALTER LCS travel-agency ADD INPUT (car_type)

- \(\Delta^O\) (M, D, op): ALTER LCS travel-agency DELETE OUTPUT (taxi_charge, taxi_schedule)

- \(\Delta^M\) (M, D, op): ALTER LCS travel-agency ADD DATA FLOWS (<USER, s_c, car_type>)

When adding a process constraint, the alter command is formed as:

\[
\text{ALTER LCS} <LCS\ name> \text{ADD PROCESS CONSTRAINT} <constraint\ type> <value>
\]

where <constraint type> can be SEQUENTIAL, PARALLEL, OUTSOURCING, and CONDITIONAL SELECTION. The four constraint types correspond to the four change operators. We give an example of a SCML alter command for each of them.

- \(\Delta^{\gg}_P\) (s_1, s_2, M): ALTER LCS travel-agency ADD PROCESS CONSTRAINT SEQUENTIAL (s_a, s_c);

- \(\Delta^{\|}_P\) (s, s_1, s_2, M): ALTER LCS travel-agency ADD PROCESS CONSTRAINT PARALLEL (s_h, s_c, s_c);
• $\Delta_P^4 (s_1, s_2, M)$: ALTER LCS travel-agency ADD PROCESS CONSTRAINT OUT-SOURCING ($s_w, s_z$);

• $\Delta_P^7 (s, C, S, M)$: ALTER LCS travel-agency ADD PROCESS CONSTRAINT CONDITIONAL SELECTION ($s_h$, $<\text{travel\_type}="\text{international}"$, $s_l>$, $<\text{travel\_type}="\text{domestic}"$, $s_c>$);

### 5.3.4 Update Command

The update command is used to specify non-functional changes. The possible update LCS actions include: (1) changing a LCS quality, and (2) changing a LCS context. When changing a LCS quality, a update command is formed as:

```
UPDATE LCS <LCS name> SET <service list> <quality parameter> <operator> <value>
```

When changing a LCS context, a update command is formed as:

```
UPDATE LCS <LCS name> SET <service list> <context type> <operator> <value>
```

The operators can be “=”, “<”, “<=”, “>”, “>=”, and “<>”.

The command corresponds to the two non-functional change operators. We give an example of a SCML update command for each of them.

• $\Delta^{QM} (\delta, M)$: UPDATE LCS travel-agency SET ($s_a$, $s_h$) $q1=\text{"high"}$

• $\Delta^{CM} (\lambda, M)$: UPDATE LCS travel-agency SET ($s_l$) $c1=\text{"European"}$

### 5.3.5 Drop Command

The drop command is used to drop a named LCS schema. We use two keywords: DROP and LCS to specify a drop command. For example, by writing

```
DROP LCS travel-agency
```

we delete the travel agency LCS schema.
Chapter 6

Change Enactment and Change Optimization

In this chapter, we present a set of algorithms for the processes of enacting and optimizing top down changes. During the process of change enactment, changes are first reacted to at two levels: schema-level and instance-level. Changes are then verified to ensure the correct configuration of a LCS. The result of change enactment may be multiple new LCS instances. The best one will be chosen during the process of change optimization.

6.1 Change Enactment

Since the kernel of SCML is a set of change operators, we mainly focus on the enactment of these operators. The enactment process consists of four major steps: selecting service nodes, updating a LCS schema graph, verifying changes, and generating the new LCS instance. Among the change operators, $\Pi_{op}^{F}(op, O)$, $\Pi_{d}^{F}(D_I, D_O, O)$, $\Pi_{c}^{C}(c, s)$, and $\Pi_{q}^{Q}(q, s)$ require the selection of abstract services in terms of service nodes in a service ontology based on a certain requirement, such as functional, context type, and quality model requirement. The other functional change operators are enacted during the process of generating new LCS schema graphs. A LCS schema is expected to be changed from a correct configuration to another correct configuration. Therefore, once there is a change to a LCS schema graph, the change needs to be verified. The non-functional change operators are enacted during the process of generating new LCS instances. This process will also be performed after a new correct schema graph is generated.
6.1.1 Select Service Nodes

Selecting service nodes is demanded by four functional change operators: $\Pi^{F}(op,O)$, $\Pi^{F}_{d}(D_{I},D_{O},O)$, $\Pi^{C}(c,O)$, and $\Pi^{Q}(q,O)$. Implementing these operators is to query the Web service ontology ($O$) and retrieve the desired service nodes. The service ontology query infrastructure proposed in Chapter 3 can then be leveraged. We elaborate on the process of implementing these operators as follows.

- $\Pi^{F}(op,O)$: It requires to find a service node from a service ontology $O$ that fulfills a functional requirement specified by $op$. The desired services are expected to provide the specified operations ($op$). It is implemented by performing an operation-based functionality query using names or path expressions, depending on the format of $op$.

- $\Pi^{F}_{d}(D_{I},D_{O},O)$: It requires to find a service node from a service ontology $O$ that has the ability of generating the specified output using the given input. It is implemented by performing a data-based functionality query on $O$.

- $\Pi^{C}(c,s)$: It requires to select services that fulfill a context type requirement specified by $c$. The desired services are expected to support $c$. It is implemented by performing a quality-based service query on $O$.

- $\Pi^{Q}(q,s)$: It requires to select services that fulfill a quality model requirement specified by $q$. The desired services are expected to support $q$. It is implemented by performing a context-based service query on $O$.

6.1.2 Update LCS schema Graphs

A LCS schema graph will be modified during the process of implementing some functional change operators. These operators include the change to abstract services (i.e., $\Delta^{S}(s,M,op)$ and $\Delta^{s_{old}\rightarrow s_{new}}(s_{old},s_{new},M)$), user input ($\Delta^{I}(M,D,op)$), LCS output ($\Delta^{O}(M,D,op)$), message exchanges (i.e., $\Delta^{MX}(m,M,op)$), and process constraints (i.e., $\Delta^{\gg}(s_{1},s_{2},M)$, $\Delta^{||}(s,s_{1},s_{2},M)$, $\Delta^{\dagger}(s_{1},s_{2},M)$, and $\Delta^{?}(s,C,S,M)$). We elaborate on the process of implementing these operators in this section.

Recall that a LCS schema graph consists of three sets. The set $N$ consists of nodes that represent abstract services. The set $DE$ consists of edges that represent data transfers among the services. The set $CE$ consists of edges that define invocation orders among the services. Adding an abstract service is quite straightforward. It can be implemented
by simply including the corresponding node to $N$. Deleting an abstract service can be implemented by performing a node deletion in a graph. The node will be removed from $N$ first. All of its incoming and outgoing edges will then be removed from $DE$ and $CE$. Adding or deleting a data transfer is straightforward, too. It can be implemented by simply updating $DE$.

Figure 6.1: The potential conflict and redundancy

The change to process constraints is not as straightforward as implementing other functional change operators. This is because that it may cause the potential conflicts or redundancy of a LCS schema graph, which is depicted in Figure 6.1.

- **Conflict**: The changes of the process constraints may cause conflicts on the invocation orders. As showed in Figure 6.1, there are three services: $s_1$, $s_2$, and $s_3$. $s_2$ is invoked before $s_3$ is invoked. $s_3$ is invoked before $s_1$. If a sequential process constraint is added to $s_1$ and $s_2$, where $s_1$ is required to be invoked before $s_2$ is invoked. This implies that $s_1$ will be invoked before $s_3$ is invoked. In this case, it will conflict the previous sequential invocation order between $s_2$ and $s_3$ and cause an invocation deadlock among these three services.

- **Redundancy**: The changes of the process constraints may also cause redundant edges in $CE$. As showed in Figure 6.1, $s_2$ is invoked before $s_3$ is invoked. $s_1$ is invoked before $s_3$ is invoked. If a sequential process constraint is added to $s_1$ and $s_2$, the edge from $s_1$ to $s_3$ is considered as a redundant edge since the invocation order between $s_1$ and $s_3$ can be derived from the other two edges.

Considering the above two situations, adding a process constraint needs to follow four steps to update a LCS schema graph: (1) Map a process constraint to a $CE$ edge (2) Check and break the potential circles (3) Check and remove the redundant edges (4) Add the new $CE$
edges to the schema graph  We will follow these steps when adding the four types of process constraints to a LCS. We elaborate on them as follows.

As showed in Algorithm 5, the first step of implementing $\Delta^>_P (s_1, s_2, \mathcal{M})$ is to map the change operator to the change of $CE$. That is, a new edge pointing from $s_1$ to $s_2$ will be first created (Line 2). After that, it needs to check and fix the potential conflicts made by the change (Line 4 to 6). These edges include the ones pointing from $s_2$ or $s_2$’s descendent nodes to $s_1$ or $s_1$’s ancestor nodes. They will be deleted from the graph to avoid the potential deadlock. The next step is to check and remove the redundant edges. These edges include the ones pointing from $s_1$ or $s_1$’s ancestor nodes to $s_2$ or $s_2$’s descendent nodes. They will be deleted, too (Line 7-9). Finally, the new edge will be added to the graph (Line 11).

The algorithm traverses through the edges in $DE$. Therefore, its computational complexity is $O(|\mathcal{M}.CE|)$.

Algorithm 5 Processing Sequential Process Constraints Operators

1: Function change_Parellel_Process_Constraints($s_1, s_2, \mathcal{M}$)
2: Input: abstract service $s_1, s_2$, a LCS schema $\mathcal{M}$
3: Output: an updated $\mathcal{M}$
4: $e = (s_1, s_2, true)$; // Step 1: map the process constraint to a CE edge
5: for all $e' \in \mathcal{M}.DG.CE$ do
6: if $e'.start == \{s_2 \cup s_2.descendent\}$ and $e'.end == \{s_1 \cup s_1.ancestor\}$ then
7: $\mathcal{M}.DG.CE = \mathcal{M}.DG.CE - e$; // Step 2: Check and break the potential circles
8: end if
9: if $e'.start \in \{s_1 \cup s_1.ancestor\}$ and $e'.end \in \{s_2 \cup s_2.descendent\}$ then
10: $\mathcal{M}.DG.CE = \mathcal{M}.DG.CE - e$; // Step 3: Check and remove the redundant edges
11: end if
12: end for
13: $\mathcal{M}.DG.CE \leftarrow e$ // Step 4: Add the new CE edges to the schema graph

As showed in Algorithm 6, the first step of implementing $\Delta^||_P (s, s_1, s_2, \mathcal{M})$ is to map it to the change of $CE$. As the definition of parallel constraint, the invocation of $s$ should be in parallel with the invocation block starting from $s_1$ and $s_2$. Therefore, $s$ should be invoked after the invocation of the parent nodes of $s_1$ and before the invocation of the child nodes of $s_2$. Therefore, the edges pointing from the parent nodes of $s_1$ to $s$ and the edges pointing from $s$ to the child nodes of $s_2$ are created. (Line 3 to 10). After that, it needs to check and fix the conflicts made by the change (Line 12 to 17). The conflicting edges include the ones pointing from $s$ or $s$’s descendent nodes to $s_1$ or $s_1$’s ancestor nodes as well as the ones pointing from $s_2$ or $s_2$’s decedent nodes to $s$ or $s$’s ancestor nodes. These edges will be deleted from the schema graph. The next step is to check and remove the redundant edges. These edges include the ones pointing from $s$ or $s$’s ancestor nodes to $s_2$ or $s_2$’s descendent nodes as well as the ones pointing from $s_1$ or $s_1$’s ancestor nodes to $s$ or $s$’s decedent nodes.
They will be deleted, too (Line 18-22). Finally, the new edges will be added to the graph (Line 25-27).

The algorithm traverses through the edges in $DE$. Therefore, its computational complexity is $O(|M.CE|)$.

**Algorithm 6** Processing Parallel Process Constraints Operators

1: Function change\_Parallel\_Constraints($s, s_1, s_2, M$)  
Input: abstract service $s, s_1, s_2$, a LCS schema $M$  
Output: an updated $M$

2: $E=\emptyset$;  
3: for all $(e \in M.DG.DE)$ and $(e.end == s_1$ or $e.start == s_2)$ do  
4: if $e.end == s_1$ then  
5: $E \leftarrow (e.start, s, true)$; // Step 1: map the process constraint to CE edges  
6: end if  
7: if $e.start == s_2$ then  
8: $E \leftarrow (s, e.end, true)$;  
9: end if  
10: end for  
11: for all $e' \in M.DG.DE$ do  
12: if $(e'.start \in \{s$ or $s'$s descendent} and $e'.end \in \{s_1$ or $s_1'$s ancestor}) then  
13: $M.DG.DE = M.DG.DE - e'$; // Step 2: Check and break the potential circles  
14: end if  
15: if $(e'.start \in \{s_2$ or $s_2'$s descendent} and $e'.end \in \{s$ or $s'$s ancestor}) then  
16: $M.DG.DE = M.DG.DE - e'$;  
17: end if  
18: if $(e'.start \in \{s$ or $s'$s ancestor} and $e'.end \in \{s_2$ or $s_2'$s descendent}) then  
19: $M.DG.DE = M.DG.DE - e'$; // Step 3: Check and remove the redundant edges  
20: end if  
21: if $(e'.start \in \{s_1$ or $s_1'$s ancestor} and $e'.end \in \{s$ or $s'$s descendent}) then  
22: $M.DG.DE = M.DG.DE - e'$;  
23: end if  
24: end for  
25: for all $e \in E$ do  
26: $M.DG.DE \leftarrow (e)$; // Step 4: Add the new CE edges to the schema graph  
27: end for

As showed in Algorithm 7, the first step of implementing $\Delta_\triangleleft^P (s_1, s_2, M)$ is to map it to the change of $CE$. In case of adding an outsourcing constraint, $s_1$ should be invoked first and then trigger the invocation of $s_2$. $s_1$’s invocation will be pending when $s_2$ is invoked since it may need the input from $s_2$. After $s_2$ is invoked, $s_1$’s invocation will be continued and finished. Therefore, the two corresponding edges will be created and included in $CE$. One is pointed from $s_1$ to $s_2$ and the other is pointed from $s_2$ to $s_1$ (Line 3 to 4). These two edges generate a circle in the graph, although they won’t cause an invocation order deadlock. Since we assume that $s_2$ does not have any interactions with other services in LCS except for $s_1$, adding the outsourcing constraint will not cause any potential conflicts and redundancy. Therefore, we can ignore the step 2 and step 3. The two new edges will be added to $DE$. 
The algorithm generates and adds two edges to $CE$. Therefore, its computational complexity is $O(1)$.

**Algorithm 7** Processing Outsourcing Process Constraints Operators

1: Functional change-Outsourcing_Constraints($s_1$, $s_2$, $M$, $op$)  
**Input:** abstract service $s_1$, $s_2$, a LCS schema $M$ and an add/remove operation $op$  
**Output:** an updated $M$

2: $e_1 = (s_1, s_2, true)$; $e_2 = (s_2, s_1, true)$; // Step 1: map the process constraint to CE edges
3: $M.DG.CE ← e_1, e_2$; // Step 4: Add the new CE edges to the schema graph

As showed in Algorithm 8, the first step of implementing $\Delta_P^C(s,C,S)$ is to map it to the change of $CE$. As defined by a conditional selection constraint, the services in $S$ should be invoked after the invocation of $s$. Each service’s invocation will be associated with a condition in $C$. Therefore, the related edges will be created (Line 3 to 6). The next step is to check and fix the conflicts introduced by the change. The conflicting edges include the ones pointing from the node in $S$ or its descendent nodes to $s$ or $s$’s ancestor nodes. These edges will be deleted from the schema graph (Line 9 to 11). The next step is to check and remove the redundant edges. These edges include the ones pointing from $s$ or $s$’s ancestor nodes to the node in $S$ or its descendent nodes. They will be deleted, too (Line 12-14). Finally, the new edge will be added to the graph (Line 17-19).

The algorithm traverses through the edges in $CE$ and use these edges to check each node in $S$. Therefore, its computational complexity is $O(|M.CE| * |S|)$.

**Algorithm 8** Processing Conditional Selection Process Constraints Operators

1: Functional change-ConditionalSelection_Process_Constraints($s$, $C$, $S$, $M$)  
**Input:** abstract service $s$, an abstract service set $M$, a condition set $C$, a LCS schema $M$  
**Output:** an updated $M$

2: $E = \phi$
3: for all $c_i \in C$ do
4: $e = (s, s_i, c_i)$; // Step 1: map the process constraint to CE edges
5: $E ← e$;
6: end for
7: for all $(e \in M.DG.CE$ do
8: for all $s' \in S$ do
9: if $e.start \in \{s' or s'.descendent\}$ AND $e.end \in \{s or s's ancestor\}$ then
10: $M.DG.CE = M.DG.CE - e$; // Step 2: Check and break the potential circles
11: end if
12: if $e.start \in \{s or s'.ancestor\}$ AND $e.end \in \{s' or s's decedent\}$ then
13: $M.DG.CE = M.DG.CE - e$; // Step 3: Check and remove the redundant edges
14: end if
15: end for
16: end for
17: for all $e \in E$ do
18: $M.DG.DE ← (e)$; // Step 4: Add the new CE edges to the schema graph
19: end for
6.1.3 Verify Changes

After a LCS schema graph is changed, it needs to be verified. Put differently, the schema graph should maintain its correct configuration after a change has been implemented. We define a set of correctness criteria in Chapter 4. The new schema graph needs to conform to these criteria. If not, it needs to be corrected. A schema graph is supposed to be correct before it has been changed. Therefore, we only need to check the nodes that are involved in the change. These nodes include those that are newly added or have edges that have been added or removed. We use $N_c$ to denote these nodes. In this section, we first analyze the impact of verification order on the process of verification. We then elaborate on the whole process of verification.

6.1.3.1 Verification Order

The proposed criteria checks the correctness of a schema graph from two aspects: structural and semantic. The structural correctness criteria check the incoming and outgoing edges of each nodes. The schema graph should not have any isolated node (criterion 1). It should not have any node that does not have any incoming edge in $CE$, which ensures that its invocation will be triggered. (criterion 2). The graph also should not have any node that does not have any outgoing edge in $DE$, which ensures that it will contribute to the output of the LCS directly or indirectly (criterion 3). The semantic correctness criteria check a schema graph from two aspects. First, it needs to check whether the edges in $CE$ ensure that each service gets its input (criterion 4). Second, it needs to compare edges between $DE$ and $CE$ to ensure that the data transfers among the services are consistent with their invocation order (criterion 5). To sum up, the semantic correctness criteria are used to ensure that services can be orchestrated properly. The structural correctness criteria are used to ensure that the nodes in a schema graph are all required.

During the process of change verification, a detected error needs to be fixed by modifying the schema graph, such as adding or removing nodes as well as adding or removing edges. Since all of the modification on the schema graph should to be verified, change verification may be trapped for repeatedly detecting and correcting. For example, a schema graph is checked to be correct with respect to criterion 5 and incorrect with respect to criterion 4. In this case, the graph will be modified to conform to criterion 4, which may result in violating other criteria. To avoid potential error, the graph needs to be re-examined with respect to other criteria. Moreover, verifying a change in an ad hoc way may cause the wrong deletion of a node. For example, a newly added node may be detected to be isolated. This will happen if
its composition with other nodes does not be specified by the owner. When this structural incorrectness is detected, the node will be removed, which is semantically wrong. Therefore, the process of verification needs to be carefully designed. First, it should ensure that the each criterion should be checked only one time. Put differently, once an error is detected, the process of fixing this error should not violate the criteria that have been checked before. Second, it should ensure that the modification on the schema will not delete a node by error.

From the above analysis, the process of verifying changes will start at examining the semantic correctness. Once an error is detected, the schema graph will be modified to fix the error. This will ensure that the modification on a LCS schema graph at this step is for the semantic purpose. After that, it will examine the structural correctness. We will elaborate the process of change verification as below, where the details of change verification order is given.

**Algorithm 9 Checking Data Transfers in a Schema Graph**

1: Functional check Data Transfer($M$, $N_c$)

**Input:** a LCS schema $M$ and a set of nodes that are affected by the change $N_c$

**Output:** an updated $M$

2: for all $n \in M.DG.N$ do
3: generating $n.mi$ and $n.mo$;
4: end for
5: $N_E = \phi$
6: for all $n \in N_c$ do
7: $I_G = \bigcup_{i=1}^{n} n.mi_i \cup D$;
8: if ($I_G \supseteq n.I$) == false then
9:   $N_E \leftarrow n$;
10:  $n.L = n.I \setminus I_G$;
11: end if
12: end for
13: for all $n \in N_E$ do
14:   for all $n' \in M.DG.N$ and $n \neq n'$ do
15:     if $n$ depends on $n'$ with data $D$ and $D \supseteq n.L$ then
16:       $M.DG.DE \leftarrow (n', n, D)$; update $n.mi$ and $n'.mo$;
17:       $n.L = n.L - D$;
18:     end if
19:     if $n.L == \phi$ then
20:       $N_E = N_E - n$; break;
21:   end if
22: end for
23: end for
24: for all $n \in N_E$ do
25:   if $n' \in M.DG.N$ and $n'.O \cap n.L$ then
26:     $M.DG.DE \leftarrow (n', n, n'.O \cap n.L)$; update $n.mi$ and $n'.mo$;
27:     $n.L = n.L - n'.O \cap n.L$
28:   end if
29:   if $n.L \neq \phi$ then
30:     $M.DG.DE \leftarrow (n_c, n, n.L)$; updating $n.mi$ and $n_c.mo$;
31: end if
32: end for
6.1.3.2 Semantic Verification

Semantic verification is to check whether a schema graph is correct using the two semantic correctness criteria. The criteria measure the correctness of both the data transfers and invocation order among services, which are embodied by the two edge sets in the schema graph: $DE$ and $CE$. It is worth to note that the invocation order is determined by the data transfer. Therefore, we will first check the correctness of data transfer among services. Once an error is detected, it will be fixed by adding or removing edges in $DE$. After examining the data transfer, we will use the edges in $DE$ to check the correctness of invocation order among services.

The first step of checking data transfers in a schema graph is to ensure that each service can get all the input it requires. It is to enforce the correctness criterion 4. This process is described in Algorithm 9. Our algorithm only checks the nodes that have been involved in the change. These nodes can be the newly added one or the nodes whose incoming and outgoing messages in $DE$ have been modified, either by removing a data transfer, or removing a node. We use $N_c$ to denote a set of such nodes. There are two user nodes in the graph: $n_\epsilon$ and $n_\omega$. $n_\epsilon$ only has outgoing messages containing the data provided by a LCS’s users. $n_\omega$ only has incoming messages containing the data required by a LCS’s users. Therefore, the user node $n_\epsilon$ may also be included in $N_c$ if the user input or LCS output is required to be changed. For example, a travel agency LCS may want to provide the local map information to its users. In this case, $n_\omega$ will be included in $N_c$ since its output changes.

We first check whether the nodes in $N_c$ can get all the required input. If not, these nodes will be added to a set $N_E$. The input that has not been covered by the node’s incoming $DE$ edges will be stored in $n.L$ (Line 6 to Line 12). The next step is to fix the errors by finding an alternative resource to supply the required input. The algorithm will follow three steps, which correspond to the three types of resources that a service node $n$ can count on for their input. The resources include the service nodes that $n$ has dependency relationship with, the users, and the other service nodes in the schema graph. The dependency relationship between different service nodes is defined in the proposed service ontology. It is described as a triplet, $s_1$, $s_2$, and $D$, meaning that $s_1$ depends on $s_2$ to provide $D$. Therefore, the algorithm will first check whether there are such nodes in the schema graph. If yes, the data transfer will be created and included in $DE$. $n.L$ will be updated, too. If $n.L$ is still not empty, the algorithm will find the service nodes in the schema graph that can provide the data left in $n.L$. It will compare between the output of other nodes in the graph and the data in $n.L$. If there is a match, the data transfer will be created and included in $DE$. If there are still some data items left in $n.L$, the algorithm will ask the user to provide the
data. The reason that we put the user node as the last one to provide the data is based on an assumption. The assumption is that a LCS’s owner will explicitly add a data transfer between the user node $n_e$ and the service node if he wants to let users provide the required information.

For each node in $N_E$, the algorithm needs to compare it’s required input with the output of other nodes in the schema graph. Therefore, the maximum time complexity is $O(|N_E| * |M.DG.N|)$.

Once the data transfer is verified, the consistency between the invocation order and the data transfer needs to be checked. This is for the purpose of ensuring that services can be orchestrated properly. As showed in Algorithm 10, for each edge $m$ in $DE$, the algorithm will check whether there is a path in $CE$ between $m$'s start node and $m$'s end node. If not, an edge pointing from the start node to the end node will be created and included in $CE$. This is to ensure that the invocation order between these two nodes is consistent with their data transfer. The path information can be stored and maintained by using a matrix.

**Algorithm 10 Checking the Consistency between $DE$ and $CE$ in a Schema Graph**

1: Function check_Message_Exchange($M$)
2: Input: a LCS schema $M$
3: Output: an updated $M$
4: for all $m \in M.DG.DE$ do
5:   if Is_a_path($M.DG.CE$, $m$.start, $m$.end) == false then
6:     $M.DG.CE$ ← $(m$.start, $m$.end, true);
7: end if
8: end for

Since Algorithm 10 will check each edge in $M.DG.DE$, the time complexity is $O(|M.DG.DE|)$.

### 6.1.3.3 Structural Verification

Structural verification is to check whether a schema graph is correct using the three structural correctness criteria. According to the criteria, the following types of nodes need to be removed from the schema graph: isolate nodes, the nodes that do not have any incoming $CE$ edges, and the nodes that do not have any outgoing $DE$ edges. Once a node is removed, all of its edges in $DE$ and $CE$ will be removed, too. This will change the incoming and outgoing edges of the removed node’s neighbors. Therefore, removing a node may trigger the removing of other nodes. To avoid repeatedly checking the incoming and outgoing nodes in the schema graph, we will examine the path instead. This process is described in Algorithm 11.
Algorithm 11 Checking structural correctness of a Schema Graph

1: Functional check_Structure_Correctness($M$)
Input: a LCS schema $M$
Output: an updated $M$
2: for all $n \in M.DG.N$ do
3: if Is_a_path($M.DG.DE$, $n$, $n_\omega$) == false then
4: delete_nodes($M.DG$, $n$);
5: end if
6: if Is_a_path($M.DG.DE$, $n_\epsilon$, $n$) == false then
7: delete_nodes($M.DG$, $n$);
8: end if
9: end for

Algorithm 11 examines each node in the schema graph for one time. Therefore, the time complexity is $O(|M.DG.N|)$.

Figure 6.2 shows the schema graph of the travel agency LCS after implementing the change in our running example. The Taxi service has been removed. All of its incoming and outgoing edges have been removed, too. The red nodes represent the newly added services, including Car Rental, Local Activities, and Traffic services. The red edges represent the newly added data transfers and invocation orders. The red data flow edges will be generated first. For a Local
Activities service, it requires three inputs: location information, user preferences and user information. The three data transfer edges need to be automatically generated so that the Local Activities service can get its required input. According to our algorithm, the dependency relationship will be first examined. We assume that a dependency relationship between the Local Activities and Hotel services is defined in the travel ontology. When these two services are combined together, the Hotel service will provide the location information. In this case, an edge pointing from the Hotel service to the Local Activities service will be generated. For the user information required by the Local Activities, our algorithm will check the output of other member services. Since the Airline has an output of flight information, which contains the user information that can be used by the Local Activities. Therefore, an edge pointing from the Airline service to the Local Activities service will be generated. The user preference cannot be provided by other services. In this case, the algorithm will explicitly require users to provide the information. An edge pointing from the user to the Local Activities will be generated. The generated edges will be included in the DE. After updating the data transferring, the invocation orders will be specified. For example, since there is a data transfer from the Airline service to the Local Activities service, a CE edge connecting these two services will be generated.

6.1.4 Compile LCS Instances

A LCS schema graph defines the participant services in the LCS and their collaboration. Therefore, once there is any update on a LCS schema graph, the LCS’s instance needs to be recompiled. It may take two steps which are determined by the change requirement. The first step is to locate Web services that fulfill both the functional and non-functional requirement of changes. The second step is to orchestrate these services following their collaboration defined in the schema graph. We elaborate on these two steps as follows.

6.1.4.1 Locate Web Services

The first step of compiling a LCS instance is to locate Web services that fulfill both the functional and non-functional requirement of a change. It will be taken under two situations. First, if a change requires the addition of a functionality, the Web service that offers the functionality needs to be located. This could be the result of implementing the following change operators: $\Delta^S (s, \mathcal{M}, op)$ and $\Delta^{S\leftarrow} (s_{old}, s_{new})$. Second, if a change is associated with a non-functional requirement, such as a context constraint or a quality requirement, the Web service that can fulfill the requirement needs to be located. This could be the result of
implementing the following change operators: $\Delta^{CM}(\lambda, \mathcal{M})$ and $\Delta^{QM}(\delta, \mathcal{M})$.

Our proposed service ontology classifies Web services into different categories based on their functionality. Each service node in a service ontology defines a type of functionality. A Web service subscribes to a service node if it provides the functionality. Therefore, the subscription relationship can be used to locate a Web service that fulfills a specified functional requirement.

The change operator $\Delta^{CM}(\lambda, \mathcal{M})$ enforces a context constraint $\lambda$ to a LCS defined by $\mathcal{M}$. Recall that $\lambda$ is a triplet that consists of $S$, $e$, and $v$, where $S$ is a set of outsourced abstract service of a LCS, $e$ specifies the context type, and $v$ specifies the new value of the context. Implementing this change operator follows two steps. First, it needs to perform a functionality-based Web service search. We specify an abstract service using the service node in a service ontology. Therefore, an abstract service is associated with a set of concrete Web services, which will be targeted during this step. Second, the services that fulfill the context constraint will be selected. This same steps are followed to implement the change operator $\Delta^{QM}(\delta, \mathcal{M})$.

### 6.1.4.2 Orchestrate Web Services

After locating services, the services that participate in a LCS can be determined. We use $WS$ to denote these services. The services are expected to fulfill both the functional and non-functional requirements introduced by a change. The next step is to orchestrate them together. The orchestration of Web services follows the combination pattern defined in the LCS schema graph. An executable BPEL description of the service orchestration will be generated.

The two major components of a BPEL description are partners and activities. We will specify each service in $WS$ as a partner link, using tags of `<portType>`, `<operation>`, and `<input>`. The directed edges in $DE$ in a LCS schema graph correspond to the two types of BPEL activities: receive and reply. The directed edges of a LCS schema graph can be mapped to the different types of BPEL activities, including receive, reply, sequence, and switch, invoke, while, etc. For example, the preparation of the required input messages for participant services can be specified using `<assign>` and `<copy>` tags. The parallelization of services can be specified using `<invoke>`. The conditional selection of services can be specified using `<switch>`, `<case>`, and `<otherwise>`.
6.2 Change Optimization

Due to the competition of service providers with similar functionalities, the updated LCS schema may result in multiple instantiations (we call them as LCS instances). Each of these instances satisfies the functional requirement specified by the change specification. However, they are differentiated from each other with respect to their non-functional properties, i.e., the quality of the provided service. We propose to optimize the LCS instance selection through quality assessment of all the LCS instances. Since providers may not always deliver according to their “promised” quality, relying only on provider promises may produce sub-optimal results. We use the reputation of concrete services in each LCS instance as an indicator of the services’ past performances, i.e., their ability to “deliver,” in accordance with the “promises” made about the Quality of Web service (QoWS). As depicted in Figure 6.3, we propose a two-phase optimization approach relying on the QoWS and reputation as the two criteria for providing the best LCS instance resulted from the change management process. We first use service reputation to filter out the services that have the low reputations. We then use QoWS as the second filter to choose the best one from the remaining trustworthy services. In what follows, we present the details of the two-phase change management optimization process.

![Two-phase optimization process](image)

Figure 6.3: Two-phase optimization process

6.2.1 Phase I: Reputation Based Optimization

The first optimization phase is to assess the reputation of the candidate Web services in a LCS instance. The result of this process is the list of Web services that have the high reputation. Since Web services are independent and autonomous, it is hard to have a central monitoring mechanism to assess their reputations. To address this issue, we use a Reputation Manager to assess the reputation of a concrete service involved in the LCS instance being evaluated. The reputation manager may inquire several peer Reputation Managers, and aggregate the respective personal evaluations for a service $s_j$. We assume a reputation collection model presented in [61], and extend it to the Web services domain using methods
presented in [43]. The single derived value (obtained as a result of the aggregation of personal evaluations collected) is defined as the service provider’s aggregated reputation in that Reputation Manager’s view. The LCS optimizer uses these aggregated reputation values for all the candidate services and filter out the ones that have the low reputation. Different Reputation Managers may employ different aggregation techniques. Therefore, the aggregated reputation value for the same provider may be different for each consumer, i.e., it may not be consistent across all consumers. Formally, the reputation of $s_j$, as viewed by a consumer $i$ is defined as:

$$Reputation(s_j, i) = \bigwedge_{x \in L}(PerEval^x_j)$$  \hspace{1cm} (6.1)$$

where $L$ denotes the set of service raters, $k$ represents the different QoWS attributes (as reliability, availability, etc.), and $\bigwedge$ represents the aggregation function. It can be as simple as representing the union of personal evaluations where the output is a real number, or an elaborate process that considers a number of factors to assess a fairly accurate reputation value. Equation 6.1 provides an approximation of how the service reputation may be calculated.

In the following, we build upon this equation to define key metrics for accurate reputation assessment. We aim to counter attacks related to deception in reputation management, i.e., identifying, preventing, and detecting malicious behavior of peers or a set of colluding peers acting as either service providers or raters. Specifically, we focus on the problems of unfair ratings through collusion (ballot stuffing or bad-mouthing). Problems as free riding, fake identities, ratings incentives, etc. are outside the scope of this paper.

**Credibility of Raters:** A major drawback of feedback-based systems is that all ratings are assumed to be honest and unbiased. However, in the real world we clearly distinguish between the testimonies of our sources and weigh the “trusted” ones more than others [32]. A Web service that provides satisfactory service, may get incorrect or false ratings from different raters due to several malicious motives. In order to cater for such “bad-mouthing” or collusion possibilities, a reputation management system should weigh the ratings of highly credible raters more than consumers with low credibilities [32, 79]. In our model, reputation is calculated as a weighted average according to the credibilities of the raters. Thus, Equation 6.1 becomes:

$$Reputation(s_j, i) = \frac{\sum_{t_x=1}^{L}(PerEval^x_j \ast C_r(t_x, i))}{\sum_{t_x=1}^{L}C_r(t_x, i)}$$  \hspace{1cm} (6.2)$$

where $Reputation(s_j, i)$ is the aggregate reputation of $s_j$ as calculated by the service consumer $i$ and $C_r(t_x, i)$ is the credibility of the service rater $t_x$ as viewed by the service consumer $i$. 
Personalized Preferences: Service consumers may vary in their reputation evaluations due to their differences in QoWS attribute preferences over which a Web service is evaluated. For instance, some service consumers may label Web services with high reliability as more reputable while others may consider low-priced services as more reputable. We allow the service consumers to calculate the reputation scores of the Web services according to their own personal preferences. Each service consumer stores its QoWS attribute preferences in a reputation significance vector (RSV). This allows the consumers the ability to weigh the different attributes according to their own preferences. Let $\phi^x_k(s_j, u)$ denote the rating assigned to attribute $k$ by the service rater $x$ for service provider $s_j$ in transaction $u$, $m$ denote the total number of attributes and $RSV_{ik}$ denote the preference of service consumer $i$ for attribute $k$. Then, the local reputation for $s_j$ as reported by service rater $x$ (using $i$’s attribute preferences) is defined as:

$$PerEval^{xj} = \frac{\sum^m_{k=1}(\phi^x_k(s_j, u) \times RSV_{ik})}{\sum^m_{k=1} RSV_{ik}}$$ (6.3)

Reputation Fading: There are situations where all the past reputation data is of little or no importance. For instance, a Web service performing inconsistently in the past may ameliorate its behavior. Alternatively, a service’s performance may degrade over time. It may be the case that considering all historical data may provide incorrect reputation scores. In order to counter such discrepancies, we incorporate temporal sensitivity in our proposed model. The rating submissions are “time-stamped” to assign more weight to recent observations and less to older ones. This is termed as reputation fading where older perceptions gradually fade and fresh ones take their place. We adjust the value of the ratings as:

$$PerEval^{xj} = PerEval^{xj} \times f_d \quad f_d \in [0, 1]$$ (6.4)

where $PerEval^{xj}$ is as defined above and $f_d$ is the reputation fader. In our model, the recent most rating has the fader value 1 while older observations are decremented at equal intervals for each time instance passed. When $f_d = 0$, the consumer’s rating is not considered as it is outdated. The “instance of time” is an assigned factor, which could be anywhere from a single transaction, ten transactions or even more than that. All transactions that are grouped in one instance of time are assigned the same fader value. In this way, the service consumer can define its own ‘temporal sensitivity degree.’ For example, a service can omit the fader value’s effect altogether by assigning it a null value. We propose to use a fader value that can then be calculated as: $f_d = \frac{1}{P_u}$, where $P_u$ is the total number of past transactions over which the reputation is to be evaluated. This allows the fader value to include all prior transaction history. However, as mentioned earlier, other calculated values for the fader
are also acceptable. Incorporating the defined metrics together, the equation for overall reputation calculation becomes:

\[
Reputation(s_j, i) = \frac{\sum_{t_x=1}^{L} \left( \frac{\sum_{k=1}^{m_i} (\phi_x(s_j, u) \cdot RSV_{ik})}{\sum_{k=1}^{m_i} RSV_{ik}} \right) \cdot f_d \cdot C_r(t_x, i) \right) \sum_{t_x=1}^{L} C_r(t_x, i) \]  

(6.5)

Through experimental evidence we have found that the above equation provides a comprehensive assessment of the reputation of a given service provider.

Till now, the reputation manager comprehensively, efficiently assesses the service reputations, which enables the service optimizer to choose the Web services that have the high reputation. Considering the expected high volume of the available Web services on the Web, the result of composing these Web service would still be multiple LCS instances. Therefore, we will use the next phase to further optimize the result of the change management.

### 6.2.2 Phase II: QoWS Based Optimization

The reputation-based optimization helps select the concrete services with desired trustworthiness. This ensures the confidence on the consistency between the promised quality and the delivered quality of the selected services. In this section, we continue to present the QoWS based optimization that selects the best LCS instance.

We define a score function to compare the quality of LCS instances [82]. Since a LCS instance contains multiple operations, we need to aggregate the QoWS parameters from different operations. Table 4.1 lists the aggregate function for each of these QoWS parameters. Since users may have preferences over how their queries are answered, they may specify the relative importance of QoWS parameters. We assign weights, ranging from 0 to 1, to each QoWS parameter to reflect the level of importance. Default values are otherwise used.

We use the following score function \( F \) to evaluate the quality of the LCS instances. By using the score function, the optimization process is to find the LCS instance with the maximum score.

\[
F = \left( \sum_{Q_i \in \text{neg}} W_i \times \frac{Q_i^\text{max} - Q_i}{Q_i^\text{max} - Q_i^\text{min}} \right) + \sum_{Q_i \in \text{pos}} W_i \times \frac{Q_i - Q_i^\text{min}}{Q_i^\text{max} - Q_i^\text{min}}
\]

where \( \text{neg} \) and \( \text{pos} \) are the sets of negative and positive QoWS respectively. In negative (resp. positive) parameters, the higher (resp. lower) the value, the worse is the quality. \( W_i \) are weights assigned by users to each parameter. \( Q_i \) is the value of the \( i^{th} \) QoWS of the LCS instance obtained through the aggregate functions from Table 4.1. \( Q_i^\text{max} \) is the maximum value for the \( i^{th} \) QoWS parameter for all potential LCS instances and \( Q_i^\text{min} \) is the minimum.
These two values can be computed by considering the operations from service instances with the highest and lowest values for the $i^{th}$ QoWS.

By using the score function, the optimization process is to find the LCS instance that maximizes the value of $F$. We present two approaches for finding the best LCS instance: exhaustive search and greedy search.

The exhaustive search enumerates the entire space of LCS instances. Suppose that a LCS instance needs to access $M$ Web services. We also assume that there are $N$ competing service providers in each service. Therefore, the complexity of the exhaustive search is $N^M$, which is exponential.

The greedy search achieves the polynomial complexity by using a divide-and-conquer strategy. It generates an optimal sub-LCS-instance from each service through local search. It then combines these sub-LCS-instances to form the final LCS instance. In order to apply the divide-and-conquer strategy, we take logarithms on the aggregation functions for reliability and availability. Specifically,

$$\text{Reliability} = \sum_{i=1}^{n} \log(\text{rel}(op_i)) \quad (6.6)$$

$$\text{Availability} = \sum_{i=1}^{n} \log(\text{avail}(op_i)) \quad (6.7)$$

These enable to express the score of the final LCS instance as a linear combination of the scores from the sub-LCS-instances. Thus, the greedy search has a complexity of $N \times M$. 
We implement a change management system in a Web service environment. The system is designed based on the following three assumptions. First, changes occur in a sequential way. We will deal with concurrent changes in our future work. Within a change specification, there is no requirement on the implementation order between different change operators. Second, we assume that the change requirement is feasible. Put differently, we can always find the Web services that fulfill the functional and non-functional requirements of a change. Third, we assume that a change is initiated from a single user. There should not be any conflict in a change specification. An example of such a conflict is that a change may require to add and remove a Taxi service at the same time. As depicted in Figure 7.1, the system consists of several components, including User Interface, Change Management, Ontology Support, Web Service Providers, Web Service Registry, and Visualization. We will elaborate on each component in this chapter.

### 7.1 User Interface

The proposed change management framework uses a graphic user interface to get three types of information as its input: a LCS schema graph, a change specification, and a Web service ontology. The information is stored as configuration files.

Users need to edit a configuration file graph.dat which contains the definition of a LCS schema graph. The information includes the nodes of the graph and the two sets of edges (i.e., data flow edges and the control flow edges). Each node represents an abstract Web service and is assigned to an id. An example of such a node id is Airline. Each node corresponds a concept
in a Web Service Modeling Language (WSML) file, which contains the semantic definition of the abstract service. A data flow edge is represented as a triplet: the node that the edge comes from, the node that the edge goes to, and a data set pair delivered by the edge. An example of such a triplet is \{Airline, Hotel, \{arrival_date, check_in_date\}\}, which means that an airline service sends a message to a hotel service containing the information of the arrival_date, which can be used as the check_in_date for the hotel service. A control flow edge is represented as a triplet: the node invoked first, the node invoked afterwards, and the condition on the invocation of the second node. An example of such a triplet is \{Airline, Hotel, “true”\}, which means that a hotel service will be invoked after an airline service is invoked. The system will first read the graph.dat and initialize the LCS schema graph.

The change.dat contains the specification of top-down changes. We use different lines to store different types of change operators. That is, each line is started with a notation which represents one type of change operators. For example, “op_01” represents adding a service node. Therefore, there will be a set of ids of service nodes following “op_01” in this line, meaning that these service nodes will be added to the LCS. If the line is started with “op_02”, it means that the service nodes at this line will be removed from the LCS.

The configuration of the associated Web service ontology is specified in the sub-menu of preference. We leverage an ontology service to provide the definition of the Web service ontology.
Considering that the process of change management may require to retrieve semantics from the ontology service frequently, it is not efficient to invoke the ontology service every time when it is needed. To improve the efficiency, we use a database as a cache between the ontology service and the system. The system accesses the ontology service only once when it is started. It will retrieve all the information from the ontology service and stored it in a database. By doing this, the system localizes the access to the semantics in a Web service ontology and improves this efficiency.

7.2 Ontology Support

In our proposed change management framework, we use ontologies to support the automation of the process of change management. The ontology support includes both semantic support and query support. The semantic support is to provide machine-understandable description of Web services such that they can be automatically located and orchestrated. The query support is to provide an efficient way to retrieve the semantics from the proposed ontology. In our prototype, we focus on the semantic support. The performance evaluation of the query support will be covered in Chapter 8. We design three subcomponents to achieve semantic support, including Ontology Providers, Web Service Registry, and WSMO API.

7.2.1 Ontology Providers

A Web service ontology definition is offered by ontology providers, which can be some knowledge experts in a certain domain. The definition will be wrapped in a WSML file. A WSML file consists of a set of service concepts. Each concept corresponds to a type of functionality offered by Web services in a domain. A service concept defines service properties, such as the service data (i.e., input and output), service operations, quality parameters, and context model. It adds machine-understandable semantics to the description of the properties. For example, for each data item of the input of a service, WSML specifies its definition in a namespace. By doing this, the meaning of the data items can be understandable by machines. More specifically, a WSML file corresponds to a service ontology of a domain. It contains a set of concepts. Each concept defines a type of functionality. A WSML file also contains the definition of the dependency relationships between different services. A dependency relationship is captured as a triplet \( \{n_1, n_2, d\} \), meaning that the data item set \( d \) is included in the input of the service node \( n_1 \) and the output of \( n_2 \). \( n_1 \) depends on \( n_2 \) to provide \( d \).
7.2.2 Web Service Registry

A service registry serves as a broker between service providers and service users. More specifically, the service providers publish their services to a service registry and service users query and locate their desired service from the registry.

The Web service registry maintains the information of a list of Web services, including their functionality (i.e., the operations they offer) and their invocation (i.e., the endpoints). In the proposed system, the Web service registry also maintains the semantic description of the Web services. When a Web service is published to the service registry, it also registers its mapping to a Web service ontology to the registry. For example, a service may offer two operations: $op_1$ and $op_2$. These two operations are mapped to two abstract services in the travel service ontology airline_quotation and airline_reservation. By using this mapping, the semantic description of the operations can be retrieved and understood by machines. The mapping between Web services and a service ontology also improve the service independency. More specifically, service providers are independent with respect to their naming mechanisms for the services and operations and the way they implement the functionality.

7.2.3 WSMO APIs

We use WSMO APIs to maintain and retrieve the semantics from a set of WSML files. The WSMO APIs provide a Java-based programming interface to build semantic Web service applications. By using WSMO APIs, we can define a set of concepts in term of classes and their instances, such as the data type of the input and output of the a service. We can also define the dependency relationships between different services. These semantic information is defined in WSML files. The WSMO APIs allow the retrieval of the semantics from WSML files. We also use WSMO APIs to implement the Web service registry. More specifically, we use the concept-instance relationship to build up the mapping between a Web service and its subscribing abstract services in the service ontology.

The application modules for this part include the follows.

- **Get_Inputs:** This module is designed to get a list of input data items of an abstract service. It takes a WSML file as input and generates the input data items as well as their formal definitions. The WSML contains the semantic description of a certain type of Web services. For example, airline.wsml describes the properties of Web services that offer the airline reservation functionality.
• **Get Outputs**: This module is designed to get a list of output data items of an abstract service. It takes a WSML file as input and generate the output data items as well as their formal definitions.

• **Get Operations**: This module is designed to get a list of operations provided by an abstract service. It takes a WSML file as input.

• **Get Concepts**: This module is designed to get the formal definition of a term. It takes a term and a WSML file as an input and generates the name of the related concept as output. The term can be an input or output of a service. It should be an instance of a concept that is defined in a certain namespace.

• **Get Instances**: This module is designed to get the instances of a concept. It takes a concept and a WSML file as an input and generates a set of instances of the concept as output. An instance can be either a data item or a service. For the formal case, the output of this module will be a set of data items that have the same meaning, such as zipcode and postcode. For the latter case, the output of this module will be a set of wsdl URIs that refer to a set of Web services that provide the similar functionality.

• **Check Dependency**: This module is designed to check the dependency between two services. It takes two service Ids as input and generate a set of data items. If there is a dependency relationship between these two services, the output of this module will be the data items that are associated with the relationship. If not, the module returns an empty set.

### 7.3 Change Management

The system focuses on managing changes at the schema-level. It is mainly performed for the purpose of fulfilling the functional requirement of changes. Since we use a graph to define the functionality of a LCS, the graph will be modified once there is a requirement to change the functionality of LCSs. The two types of correctness of a LCS schema graph need to be maintained: semantic and structural correctness. Therefore, there are two subcomponents implemented for this phase: schema graph modifier and change verifier.

The schema graph modifier maintains and manages the schema graph of a LCS. A graph contains a set of service nodes. Each node is associated with a WSML file, where the semantic service definition is stored. The data flow and control flow among different services are stored in terms of two edge sets in the graph. The schema graph modifier takes a change
operator as input, such as adding a service node, removing a service node, etc. It first checks whether the change is feasible. If yes, it then implements the algorithms proposed to modify the graph to implement the change operators. We use Java to implement this part. The application modules for this part include the follows.

- **Schema_Graph_Initialization:** This module is designed to initialize a LCS schema graph. It first reads the configuration file graph.dat which contains the description of the schema graph. It then uses the information to assign values to the graph’s elements, including the nodes and two sets of edges.

- **Change_Analysis:** This module is designed to generate a set of steps of change reaction. It first reads the configuration file change.dat which contains the description of the changes. It then calls other modules to implement the changes.

- **Update_Inputs:** This module is designed to change the user input of a LCS. The input data items that are intended to be added are stored in an arraylist AddInputs. The input data items that are intended to be deleted are stored in an arraylist RemoveInputs.

- **Update_Outputs:** This module is designed to change the abstract services that a LCS outsources. It will first add service nodes to the LCS schema graph. The information about these service nodes is stored in an arraylist AddServices. It will then remove services from the LCS schema graph. The information about these service nodes is stored in an arraylist RemoveServices.

- **Update_AbstractServices:** This module is designed to change the abstract services that a LCS outsources. It will first add service nodes to the LCS schema graph. The information about these service nodes is stored in an arraylist AddServices. It will then remove services from the LCS schema graph. The information about these service nodes is stored in an arraylist RemoveServices.

- **Update_DataEdges:** This module is designed to update the data flow edge set of a LCS schema graph. A data flow edge is represented as a triplet \( \{n_1, n_2, d\} \), meaning that service node \( n_1 \) sends a message to \( n_2 \) containing the data set of \( d \). This module will first map the data transfers specified in the change file to data flow edges and add them to the LCS schema graph. The information about these data transfers are stored in an arraylist AddDataTransfer. This module will first map the data transfers specified in the change file to data flow edges and add them to the LCS schema graph. The information about these data transfers are stored in an arraylist AddDataTransfer. It will then
delete the data transfers specified in the change file from the LCS schema graph. The information about these data transfers are stored in an arraylist RemoveDataTransfer.

The change verifier checks and ensures that the correctness has been maintained during the process of implementing a change operators. When checking the semantic correctness, it takes input from a set of WSML files to get the semantic description of a service node. The information required by the change verifier includes service input, output, and the dependency relationship between services. By using the information, the change verifier first detects and fixes the incorrectness of data transfer among services. It then detects and fixes the inconsistency between the data transfer and the invocation order among services. We use Java and the WSMO API to implement this part. The application modules for this part include the follows.

- **Check_Data_Transfers**: This module is designed to check and ensure that all the service nodes can get their required input. It takes a LCS schema graph as input and updates the graph if necessary. This module takes a two-step process. During the first step, it will call the module of Get_Inputs to get the input information of each service. It then compares the input and the related data flow edges to check whether there is any input data item that can not be covered by the data flow. If yes, the service will be added to an arraylist IncompleteInputNodes. During the second step, the module will generate new data transfers for the services in IncompleteInputNodes so that they can get their required input. It will first call the module of Check_Dependency to check whether the service depends on other services to provide the input. If yes, a new data flow edge between the two services will be created and added to the schema graph. If there is still any input data item that can not be covered by the data flow, the module will then check whether the user input contains the required information. If not, the module will check whether the other services in the LCS that can provide the information. It will call the module of Get_Outputs to get the output of other services and compare them with the required input data items.

- **Check_Consistency**: This module is designed to check the consistency between the data flow edges and control flow edges. It takes a LCS schema graph and update the graph if necessary. For each data flow edge, the module will check whether there is a control flow path between the two service nodes. If not, a new control flow edge between these two nodes will be created and added to the graph.

- **Check_Graph Structure**: This module is designed to check the structural correctness of a LCS schema graph. It will first check whether there is any isolated node in the
graph. If so, the node will be removed. For the remaining service nodes, the module will check whether there is a data flow path between the node to the user node. If not, the node will be removed. The module will also check whether there is a control flow path between the user node to the service node. If not, the service node will be removed, too.

Figure 7.2: Importing a Web service ontology

7.4 Visualization

The system provides a graphic user interface for visualizing the process of change management. There are two types of information that are visualized. First, the interface will show the schema graph before and after implementing a change. Second, the interface will show
every step of modifying a LCS schema graph, such as adding a node, removing a node, adding an edge, removing an edge, etc.

For the purpose of clarity, the schema graph is decomposed into two sub-graphs. One sub-graph depicts the control flow among the outsourced services. The other sub-graph depicts the data flow among the outsourced services. A log file is used to keep track of the process of change management. The visualization module reads the management steps from the log file and visualizes them. We leverage the JGraph and JGraph Layout Pro library to visualize the graphs [1]. The graph can be zoomed in and zoomed out for a better view. It can also change the orientation of the graph as required.
Figure 7.4: The west-oriented view control flow of a LCS after change management
7.5 System Usage

In this section, we introduce how to use the implemented change management system. We use a travel agency LCS in our running example as the scenario. For the sake of space, we only introduce some representative steps of using the system.

Figure 7.5: The north-oriented view of control flow of a LCS after change management

Figure 7.2 shows the first step of using our system. It is to load a Web service ontology to the system. The system invokes the ontology service and retrieves the related semantics from the ontology service. By configuring the access to an ontology service, users can import the information from the Web service ontology, such as the types of Web services within a domain and their features (i.e., input, output, etc.) Users can also import the dependent relationship among services from the ontology to compose services together.

Figure 7.3 shows how to import the information about a LCS and a change specification.
Figure 7.6: The data flow of a LCS after change management
A LCS is specified in a configuration file. Once the file is loaded, the user interface will display the list of the services that are included in the LCS. There are two ways of specifying a change. First, it can be done by loading a change configuration file. Second, it can be done by using the left-arrow and right-arrow buttons in the functional change panel. The latter way can be used if the changes only limit to the changes of the outsourced services, such as adding or removing a service. In this case, the system will automatically generate a change configuration file. It will read the service definition from the database and update the user input and LCS output. Once a LCS schema graph and a change are specified, users can click the Apply Change to implement the change. The user interface also gives an option of tracking the process of managing changes. If users choose this option, the steps of change management will be traced and stored in a file, which can be used for the further visualization.

Figure 7.4, Figure 7.5, and Figure 7.6 show a LCS schema graph before and after implementing changes. Users can choose between control flow and data flow to display the corresponding information. Figure 7.4 shows the control flow of the LCS before and after the change management. In the graph, there are three new services added (i.e., CarRental, Traffic, and LocalActivities). The control flow can also be showed in a different orientation, as depicted in 7.5. The invocation orders among these services and other services in the LCS are automatically generated. Figure 7.6 shows the data flow of the LCS before and after change management.

Figure 7.7 and 7.8 shows the change of the LCS schema graph for different steps of change management. Users can use “next” button to get the difference of the schema graph step by step. Users can also specify a step number and directly jump to the step.
Figure 7.7: The first step of change management
Figure 7.8: The 31st step of change management
Chapter 8

Performance Study

In this chapter, we conduct a set of experiments and simulations to evaluate the performance of the three approaches proposed in this dissertation: change enactment process, ontology query optimization, and change optimization. We will elaborate on the performance study on the three approaches in the chapter.

8.1 Change Enactment Performance

We conducted a set of experiments to assess the performance of the change management algorithms. We use the travel agency example as our testing environment to setup the experiment parameters. We include typical Web services in the travel domain: Airline, Hotel, CarRental, Taxi, LocalActivities, Weather, and Map. We run our experiments on a MacPro with a Quad-Core processor, 4-GB of RAM, and under Mac OS X 10.5. We focus on three sets of parameters and investigate how these parameters affect the efficiency of the proposed algorithms. In particular, LCS size refers to the number of the services in the original LCS. In the first set of experiments, we study how the LCS size affects the performance. Change size refers to the number of services involved in a change. In the second set of experiments, we study how the change size affects the performance. We focus on two different types of changes: adding services and removing services. Table 8.1 summarizes the parameter settings of the experiments.

---

1Replacing a service can be achieved by first removing the service and then adding a new service.
Table 8.1: Experiment parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS Size</td>
<td>[1, 7]</td>
<td>ls</td>
</tr>
<tr>
<td>Change Size</td>
<td>[1, 6]</td>
<td>cs</td>
</tr>
<tr>
<td>Change Type</td>
<td>add, remove</td>
<td>ct</td>
</tr>
</tbody>
</table>

### 8.1.1 Effect of the LCS Size

We evaluate how the size of a LCS affect the performance of the algorithm. More specifically, we fix the change size as one and vary the LCS size from 1 to 5. We assess the performance against two types of changes: adding services and removing services. Since different services may cause different cost for a change, we conduct the experiment on two different selected services for comparison.

![Figure 8.1: The impact of the LCS size (cs=1, ct='add')](image)

Figure 8.1 shows the processing time spent by adding a new service to a set of LCSs with different sizes. More specifically, it presents the time used to react to and verify the change. From the figure, we have the following observations.

- First, the processing time of adding a service does not necessarily increase with the size of LCS. A LCS is assumed to have a correct configuration before a change. Therefore, each service in the LCS can get its input from either the user or the other services. During the change enactment process for adding a service, we only examine the “affected” nodes for the verification purpose. Since adding a service won’t cause the deletion of any existing data flow edge, the only “affected” node is the new service. New data flow edges will then be generated to supply the input for the new service. Therefore, the
processing time of change enactment does not necessarily depend on the size of the LCS.

- Second, adding a different service will use different processing time for change enactment. Web services are different in terms of service data (i.e., input and output) and their dependent relationship with other services. All of these features have a direct impact on the process of change enactment. For example, suppose service A has more data items in its input than service B. It means that more data items need to be checked when adding service A than when adding service B. Therefore, it is likely that adding service A will cost more time than adding service B. For another example, suppose that service A has dependent relationship with service C in the LCS and service B does not have. Recall that when finding the input supplier for an “affected” service, the service dependent relationship will be first checked. After that, user input will be checked. Finally, if the service still has some input not covered, other services in the LCS will be checked. In this case, C can be first located to provide service A’s input when adding service A. For B, in contrast, it may need to go through the entire process to find its input supplier. Therefore, it is likely that adding service B may cost more time than adding service A. As Figure 8.1 shows, adding a Weather service always costs more time than adding a Traffic service since the Weather service has more service data items than Traffic service.

Figure 8.2: The impact of the LCS size (cs=1, ct='remove')

Figure 8.2 shows the processing time spent by removing a service from a set of LCSs with different sizes. From the figure, we find that the performance of change enactment varies when removing different services. When removing an Airline service, the processing time increases with the size of LCS. On the other hand, when removing a Payment service, the
processing time varies very slightly with the LCS size. Since removing different services may result in generating different “affected” services, which determines different processing time. An Airline provides input for several services, such as Hotel, CarRental, etc. When removing the Airline service, these services are all “affected”. During the change verification process, alternative input suppliers need to be found for each “affected” services. The number of “affected” services is likely to increase with the LCS size, which leads to the increase of the processing time. A Payment service, on the other hand, does not provide input for other services. Therefore, the processing time of removing the Payment service does not increase with the LCS size.

8.1.2 Effect of Change Size

We evaluate the performance in terms of the change size against the LCS size and change type in this section. We set the LCS size as three for the adding services and seven for the removing services. We conduct two sets of experiments which correspond to the two type of changes: adding services and removing services. We apply two method for adding and removing services. For the first method, the change is applied in a “batch mode”. More specifically, when multiple services need to be added to or removed from a LCS, they will be dealt with simultaneously. This method is actually adopted in the proposed change enactment process. For the second method, the change is applied in a “single mode”. Specifically, when multiple services are involved in the change, they will be dealt with one after another.

![Change size vs. Processing time (ls=3, ct=’add’)](image)

Figure 8.3: The impact of the change size (lcs=3, ct=’add’)

Figure 8.3 shows the processing time spent by adding different number of services to a LCS. The LCS consists of an Airline, a Hotel and a Taxi service. The number of the added services varies from 1 to 4. From the figure, we can have the following observations.
• First, the processing time of adding services to a LCS increases with the change size. Adding services will not cause the deletion of the existing data flow edges. Thus, the only “affected services” will be the ones that are newly added. Therefore, the more services that are added to a LCS, the more “affected services” the LCS has. As a result, the more processing time will be used to find input suppliers to these services.

• Second, adding multiple services together achieves better performance than adding multiple services separately. During the process of change verification, once there is a change on the data flow edges, the consistency between the data flow and control flow will be examined. After that, the structural correctness of a LCS schema will be checked too. When adding services together, the consistency and structure check only needs to be performed one time. When adding services separately, on the other hand, the consistency and structure checks need to be performed each time when adding a new service to the LCS. Therefore, it will consume more time.

![Change size vs. Processing time (ls=6, ct=’remove’)](image)

Figure 8.4: The impact of the change size (lcs=6, ct=’remove’)

Figure 8.4 shows the processing time spent by deleting different number of services from a LCS. The LCS consists of an Airline, a Hotel, a CarRental, a Traffic, a Weather, a Payment, and a LocalActivities service. The number of the deleted services varies from 1 to 6. From the figure, we can have the following observations.

• First, the processing time of deleting services from a LCS does not obviously increase with the change size. Removing services will cause the deletion of the existing data flow edges. This may potentially generate the “affected” services. Therefore, it will cost more time on finding the input supplier for the affected services. However, the number of “affected service” does not necessarily increase with the change size. We
can use an artificial example to illustrate this. Consider two changes $c_1$ and $c_2$. In $c_1$, service $A$ is removed. In $c_2$, both service $A$ and service $B$ are removed. Service $B$ is the only “affected” service when removing service $A$ from the LCS. In this case, there are more “affected” service generated in $c_1$ (i.e., service $B$) than the one in $c_2$ (i.e., none). Therefore, it will cost more time to react to $c_1$ than to react to $c_2$.

- Second, removing multiple services together achieves better performance than removing multiple services separately. The reason is the same as discussed in the case of adding services.

8.2 Ontology Query Optimization

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.

![Figure 8.5: NA Vs. $d$](image)
8.2.1 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 8.5 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 8.6 shows the actual CPU time, which demonstrates a very similar trends as the number of node accesses.

![Figure 8.6: Time Vs. d](image)

8.2.2 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 8.7 and 8.8 show the number of node accesses and the CPU time, respectively. The results are fairly
consistent with those from Section 8.2.1. The results also further confirm the efficiency of the proposed algorithm.

![Node accesses vs. f](image1)

**Figure 8.7: NA Vs. f**

![CPU time (ms) vs. f](image2)

**Figure 8.8: Time Vs. f**

### 8.3 Change Optimization

We conducted a set of experiments to assess the performance of the proposed change optimization approaches. We assume that the concrete services available to the QoWS optimization process are selected by the reputation manager. Thus the promised quality of these services can be guaranteed with enough confidence. We use the travel agency example
as our testing environment to setup the experiment parameters. The purpose is to demonstrate how our approach can help efficiently manage top-down changes in LCSs. We run our experiments on a cluster of Sun Enterprise Ultra 10 workstations under Solaris operating system.

We create a service ontology containing four types of services, including an airline service, a hotel service, a car rental service, and a taxi service. For simplicity, we omit the unnecessary service operations and consider the runtime and business QoWS parameters. Each service contains two operations: $S_{\text{airline}}$: (flightSearch, flightReservation), $S_{\text{hotel}}$: (hotelSearch, hotelReservation), $S_{\text{car}}$: (carSearch, carReservation), $S_{\text{taxi}}$: (taxiSearch, taxiReservation). The number of service providers in each service ontology varies from 10 to 60. The values of these QoWS parameters are generated within a range based on uniform distributions. We consider three types of changes, removing a service from a LCS, replacing a service in a LCS with another one, and adding a new service to a LCS. Specifically,

- $C_1$: Remove the hotel service $S_{\text{hotel}}$ from the LCS($S_{\text{airline}}, S_{\text{hotel}}$).
- $C_2$: Replace the car rental service $S_{\text{car}}$ with a cruise service $S_{\text{taxi}}$ from the LCS($S_{\text{airline}}, S_{\text{car}}$).
- $C_3$: Add a hotel service $S_{\text{hotel}}$ to the LCS($S_{\text{airline}}, S_{\text{taxi}}$).

We measure the performance of the optimization approaches when applying these three types of changes. We use computational time and score function value as our evaluation criteria. We conduct a set of experiments for each of the three types of changes.

![Figure 8.9: Optimization time Vs. number of providers](image)

(1) **Optimization time**: Figure 8.9 shows the total optimization time spent by exhaustive search and greedy search. Specifically, it presents the time used to select the best LCS instance when dealing with changes $C_1$, $C_2$, and $C_3$. For change $C_1$ where there is one service
remained in the LCS after the change, these two optimization approaches have a similar performance. This is because the divide-and-conquer strategy makes no difference when only one service is considered. For changes $C_2$ and $C_3$, greedy search is much more efficient than exhaustive search due to the divide-and-conquer strategy.

Figure 8.10: Scores of the best LCS instances

(2) **Score of the best LCS instances**: In addition to the improvement on the optimization time, greedy search is able to maintain the quality of the best LCS instance. Figure 8.10 shows the scores of the best LCS instances generated by both optimization approaches. In dealing with all the three changes, greedy search generates the best LCS instances with scores almost the same as those from the best LCS instances generated by exhaustive search. The slight difference comes from the two approximation functions used by greedy search to aggregate QoWS parameters.

Figure 8.11: Optimization time vs. number of services

(3) **Effect of the number of services**: In the above experiments, we focus on examining how the number of concrete services affects the optimization time. In this set of experiments, we investigate the relationship between the number of services in a LCS instance and the optimization time. As shown in Figure 8.11, the optimization time of greedy search has a much slower increasing rate than that of exhaustive search as the number of services increases.
Chapter 9

Related Work

Change management is an active research area. Although it is relatively new in the Web service community, some frameworks have been proposed for managing changes for process-oriented systems, such as workflows [23, 13, 57, 36, 62, 45]. In this chapter, we overview some related work in this area.

9.1 Change Management in Web Service Community

In [5], a framework is proposed for detecting, propagating, and reacting bottom-up changes in an SOE. A bottom-up change starts at the implementation level of an SOE. Change may occur to an individual service without consent of the enterprises that utilize the service. The changes may affect the invocation of other services in the SOE, which affects the entire performance of the SOE. To manage these bottom-up changes, 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.

In [11], a framework is presented to detecting and reacting to the exceptional changes that can be raised inside workflow-driven Web application 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. For example, the free user navigation through Web pages may result in the wrong invocation of the expired link, or double-click the link when only one click is respected. The semantic exceptions are driven by unsuccessful logical outcome of activities execution. For example, a user does not keep paying his periodic installments. 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 modeling framework that describes the structure of activities inside hypertexts of a Web application. The hypertext belonging to an activity is broken down into pages, where are univocally identified within an activity. It presents a framework to handle these changes. The framework consists of three major components: capturing model, notifying model, and handling model. The capturing model capture events and store the exceptions data in the workflow model. The notifying model propagate the occurred exceptions to the users. The handling model defines a set of recovery policy to resolve the exception. For different types of exceptions, different recovery policies will be used.

In [55], a framework is presented to manage the business protocol evolution in service-oriented architecture. It uses several features to handling the running instances under the old protocol. These features include impact analysis and data mining based migration analysis. The impact analysis is to analyze how protocol change impacts on the running instances. It will be used to determine whether ongoing conversations are migrateable to the new protocol or not. The data mining based migration analysis is used for cases where the regular impact analysis cannot be performed. Service interaction logs are analyzed using data mining techniques. It then uses the result of the analysis to determine whether a conversion is migrateable or not. In [55], the work mainly deals with dynamic protocol evolution. We focus on automatically modifying the composition of Web services once there is a new requirement introduced by a change.

### 9.2 Adaptive Workflow System

Workflows are the popular means of composing enterprises. They provide the ability to execute business processes that span multiple organizations [59]. Traditional workflows do not provide methods for dynamic change management. Workflows are geared towards static integration of components. This characteristic inhibits the profitability, adaptability, utility, and creativity in an enterprise. Furthermore, workflows do not cater for the behavioral
aspects of Web services. For example, they do not distinguish between the internal and external processes of a Web service [12].

Workflow management systems (WfMS) aim at coordinating activities between different business processes (systems) so they can run in an efficient manner. This is done by automating the business processes and invoking appropriate resources in a sequence. According to the WfMC [18], “workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve or contribute to an overall business goal.”

[23] focuses on modeling dynamic changes within workflow system. It first identifies a set of modalities of changes, including change duration, change lifetime, change medium, change time-frame, change continuity, change agents, change rules, and change migration. The change duration specifies whether the change is to happen quickly (instantaneous) or over a noticeable long (but finite and well specified) time period or an unspecified amount of time (indefinite). The change lifetime specifies the amount of time that the change is in effect. The change medium specifies the medium to make change, such as human (manually), software agents (automatic), or mixture. The change time-frame specifies whether the change should enforce constraints on the work-cases that are currently in progress. The change continuity specifies the migration strategy, such as preemptive or integrative. The change agents specifies which participants play which organizational roles within the change process. The change rules guide a change process in its pursuit of meeting the goals that changes come to life to achieve. Change migration refers to the ability to bring the filtered-in cases into compliance with the new procedure in accordance with the migration policies agreed upon by the change designers. This work then 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.

[54] presents a Petri net based approach to maintaining correctness between process type and instances. A process type represents a particular business process described by a schema. Process instance is a real time execution of the process type. Changes to process type occur when the process schema is modified in response to the environment. For example, a business process may adapt to comply with new legislation, or it may be optimized for performance reasons. The respective process type changes must be propagated to the process instances.
Inversely, process instances may be changed to accommodate for changes in the execution environment. For example, an exception may cause the process to skip a task. A mapping of this instance to the process type results in a schema that is different from the original process schema. When process type and instance changes are executed independently, they are no longer in harmony with each other.

9.3 Change Management in Traditional Databases

Several change detection algorithms have been proposed to measure changes in Web content and also XML documents [15, 19]. Most of the algorithms that deal with changes in Web content aim to maintain the consistency of data on the Web site. They do provide mechanisms to “understand” what the data represents and react appropriately. Research on detecting changes to XML documents has been mainly focused on the syntactic changes to a document. For example, [19] provides an efficient algorithm that detects structural change to an XML document. It does not efficiently detect semantic changes to XML documents such as WSDL descriptions. Furthermore, [19] deals with detecting changes after a move operation.

Similarly, [15] proposes the detection of changes in hierarchical data. This data is primarily nested and is not represented by a unique identifier. It proposes techniques for comparison between two structured data versions. However, these techniques do not deal with the Web service environment. Therefore, change detection is only at the abstract layer and cannot be readily applied to Web services.

9.4 Ontologies for Web Services

There are two major purposes to markup semantic Web service description. First, adding semantics to service description is to enable automatic service composition. It means that software agents can automatically select, compose, and interoperate Web services given a high-level description of a requirement. To reach this goal, a service description needs to provide declarative specification of the IOPE (Input, Output, Precondition, Effect) of the service. Second, adding semantics to service description is to facilitate efficient access to Web services. It means that software agents can optimize an execution plan from the multiple competitive services for a given requirement. To reach this goal, service description must provide declarative specification of the quality of Web services. A research trend for
semantically describing Web services is to build ontologies for Web services.

The term *ontology* originates from philosophy. In that context, an ontology is used to describe existing objects by grouping them into abstract classes based on shared properties. Since the beginning of the 1990s, ontologies have been used in artificial intelligence areas to facilitate knowledge sharing and reuse. In a nutshell, ontologies improve communications among humans and software entities by providing “*the common understanding of a domain of interest* [6].” Many definitions of ontologies have been offered. The most cited one, which best characterizes the essence of an ontology, is “*an explicit and formal specification of a conceptualization* [29].” *Conceptualization* refers to an abstract model that encapsulates the properties of objects and activities of the real world. An ontology should describe knowledge in a way that is same as what people take to conceive the world. *Formal specification* refers to the fact that knowledge should be specified in a machine-processable language. *Explicit specification* refers to the fact that concepts and their relationships should be defined explicitly. An ontology should avoid of containing implicit information to decrease confusion in knowledge specification. An ontology contains a vocabulary of terms (e.g. people, stuff, courses, etc) in a given domain. These terms can be presented in an ontology in form of classes (concepts). The relationships between these terms can be shaped as a lattice or taxonomy of classes and subclasses. Ontologies are different in their degrees of formality. At one end, ontologies like Dublin Core [77] and WordNet [25] provide a thesaurus for large number of terms to explain online information. At other end of the spectrum, ontologies like CYC [38] and KIF [27] provide formal axiomating theories based on the first-order predicate calculus. A tradeoff exists between the two contributions of an ontology: knowledge expressiveness and reasoning supports.

### 9.4.1 OWL

DAML+OIL was adopted by W3C as a starting point of OWL, an ontology language for the Semantic Web [76]. OWL builds on the syntax of RDF and RDF Schema. Moreover, it tackles the expressive limitations of RDF. RDF defines the online resources and their relationships by modeling them as classes. It also describe their relationships. However, it fails to provide the other important features for reasoning. For example, if x is a subclass of y and y is a subclass of z, we can know that x is a subclass of z. However, RDF cannot express such transitive property of subclass relationships. OWL adopts the RDF meaning of classes and properties and add more language primitives for online resource inference. These primitives define transitive, symmetric, and inverse properties of classes. They also define boolean combinations (union, intersection, and complement) to support flexibly reasoning
online resources.

9.4.2 OWL-S

As a key evolution of distributed applications, Web services are experiencing a fast growth. Ontologies help enrich the semantic descriptions of Web services to facilitate automation of service discovery, service invocation, service composition, service interoperation, and service execution monitoring. OWL-S [22] is a representative ontology language built on OWL for Web services. It provides a set of markup language constructs for describing properties and functionalities of Web services in a machine-understandable manner.

As showed in Figure 9.1, OWL-S describes a Web service from four aspects, including service provider, service capabilities, service execution process, and service accesses. These four aspects are respectively represented by four classes, including resource, serviceProfile, serviceModel, and serviceGrounding. The serviceProfile provides a high-level view on Web services. It describes information about a service function, including the service provider, functional description, and the service characteristics. The functional description specifies the required input and the generated output of the service. It also specifies the preconditions and effects. The serviceProfile plays the similar roles on Web service representation as what UDDI plays. The difference between the capabilities of serviceProfile and UDDI is that serviceProfile enriches more semantics. The serviceProfile also supports automatic Web service discovery. The serviceModel provides a detailed view on how a service operates as a process. A process consists of a set of tasks and their execution order. The specified information of a process includes its input, output, preconditions, and effects. The
input/output relates to the processes that transform data. The precondition/effect relates to the processes that change the state of the world. The serviceModel also specifies data flows and parameter bindings. It enables agents to derive the services choreography. It thus supports automatic Web service composition. The serviceGrounding specifies the detail of how to access Web services. It mainly describes the binding information of services, including protocols, message formats, serialization, transport, and addressing. It thus can be used to support automatic Web service invocations.

9.4.3 WSMO

Web Service Modeling Ontology (WSMO) is an ontology for describing several aspects of semantic Web services. It takes Web Service Modeling Framework (WSMF) [26] as a starting point. The WSMF consists of four main parts, including goals, ontologies, mediators, and Web services. The goal defines the problems that a Web service is expected to address. The ontology defines the formal semantics for the terms used in other elements of the WSMF. The mediator is used to address interoperability problems, such as data mismatches and process sequence mismatches. The Web service part defines several elements to describe a Web service, including the precondition, post-condition, data flow, control flow.

WSMO refines the WSMF and defines these elements in a formal fashion. The WSMO definition of a Web service consists of four parts, including nonFunctionalProperties, usedMediators, capability, and interface. The nonFunctionalProperties describe the properties that are not directly related to a Web service’s functionality, such as service providers, cost, performance, reliability, security, etc. These properties are mainly used to help software agents discover and select Web services. They can also be used for negotiation. The usedMediators define the mediators used by a Web service. For example, a Web service can use the concepts and relationships elsewhere by importing ontologies through ontology mediators (ooMediators). It can also use wgMediator to address interoperability problems between a Web service and goals. The capability defines the functionalities of a Web service. It helps software agents locate a desirable Web service. A capability can be modeled by using preconditions, postconditions, and effects. The interface describes how a Web service functionalities can be fulfilled. It can be used to help software agents invoke and combine Web services. The WSMO definition describes the interface of a Web service from a twofold view, including choreography (from user’s prospective) and orchestration (form other service provider’s perspective). The choreography describes the information about how to interact with a service to make use of its functionalities, such as Message Exchange Pattern (MEP). The orchestration describes the information about how a Web service is outsourced.
to provide a value-added service. It has a tight relationship with the Problem Solving Pattern (PSP), which specifies a sequence of activities to achieve a given requirement. *Web Service Modeling Language* (WSML) [78] provides a formal syntax and semantic for WSMO. It is based on well-known logical formalisms, including first-order logic, description logic, and logic programming.
Chapter 10

Conclusion

In this chapter, we summarize the major contributions of this research. We then discuss several directions for the future research.

10.1 Research Summary

Web services are gaining momentum as a new computing paradigm for delivering business functionalities on the Web. They are increasingly regarded as the most promising backbone technology that enables the modeling and deployment of the Service-Oriented Architecture. Many service providers expose to move their business functionalities on the Web using Web services. This, in turn, has opened the opportunities for composing autonomous services on demand. Meanwhile, it has raised the research issues of managing changes during the lifetime of composed services.

In this dissertation, we propose an end-to-end framework to model, implement, verify, and optimize top-down changes in a LCS. We summarize below the major contributions of this dissertation.

- **Web service ontology support** - We formally define a Web service ontology to capture the common features of Web services in a certain domain. The semantics defined can be used for automatic service selection and composition. Web services are classified into different categories based on their functionality. Based on the classification, we define a tree-like structure for a Web service ontology. Each service category is defined as a service concept in a Web service ontology, a.k.a., abstract service.
also propose a set of query mechanisms for efficiently retrieving semantics from the hierarchical structure of a Web service ontology.

- **LCS schema** - We formally define a LCS schema that gives a high-level description of a LCS. A LCS schema provides the semantics for specifying, reacting to, and verifying changes. The LCS schema is expected to help software agent understand what a LCS offers and how it works. We use a direct graph to define the composition among services. The graph includes two sets of edges, which correspond to the execution order and message exchange among services, respectively. We then define the correctness of a LCS schema based on the definition.

- **Change management language** - We propose a change management language to specify top-down changes in a LCS. Each change is associated with a requirement on modifying a LCS’s feature, both functional and non-functional. We first define a change taxonomy that classifies changes into different categories. The change classification is based on their associated requirement. We define a change operator for each type of changes. Based on the change taxonomy, we propose a change language. The change management language, SCML, consists of three types of commands: definitive commands, query commands, and change commands. The definitive commands can be used to define a LCS schema. The query commands can be used to query semantics from a Web service ontology. The change commands can be used to specify changes.

- **Change reaction and change verification** - We present a process of change enactment to implement the changes specified in terms of the proposed SCML language. Changes are reacted to at two levels: schema-level and instance level. During the schema-level change reaction, a LCS schema is modified to fulfill the functional requirement associated with a change. During the instance-level change reaction, a new LCS instance is generated by selecting and orchestrating Web services that both fulfill functional and non-functional requirement associated with a change. We also propose a process of verifying changes. We first check the correctness of a LCS schema from the semantic perspective. It is for the purpose of ensuring that the services in a LCS can be invoked gracefully. We then check the correctness of a LCS schema from the structural perspective. It is for the purpose of ensuring that there is no isolated services in a LCS.

- **Two-phase change optimization** - We propose a two-phase process of optimizing the result of change management. There are multiple service providers competing to provide a similar functionality. This results in multiple candidates when generating a LCS instance. During the first phase, We use service reputation as a criterion for
optimization. The service reputation reflects the degree of users’ confidence on whether a service delivers the functionality and quality as promised. During this phase, the services that have low reputation will be filtered out. During the second phase, we use QoWS as a criterion to choose the best service.

10.2 Directions for Future Research

We identify the three directions for the future research: relaxation on change specification, concurrent change management, and bottom-up change management.

- **Relaxation on change specification** - In this dissertation, the change specification is required to contain unambiguous and sufficient information about a change. An insufficient description of a change may result in a failed change enactment. For example, a LCS owner may want to add a car rental service to the LCS. The choice between invoking a car rental service and a taxi service will depend on customer’s travel type, which can be either international travel or domestic travel. In this case, a conditional selection process constraint between the car rental service and the taxi service needs to be included in a change specification. This requirement is imposed on the LCS owner, which requires human involvement and is error-prone. Relaxing this requirement could be an interesting direction for our future work.

- **Concurrent change management** - In this dissertation, we assume that changes occur sequentially. Specifically, there is only one change that needs to be managed at one time. In the real world, changes may occur concurrently and they need to be managed at the same time. A top-down change is always the result of a new business strategy, policy, regulation, or law. It reflects a new requirement on a LCS’s functional and/or non-functional features. Therefore, concurrent change management is needed when there are multiple requirements on a LCS at the same time. Concurrent change management raises several new issues that need to be addressed. The issues include: (1) detecting and reconciling the conflicts between different changes; (2) incorporating and dealing with different priorities on different changes (3) investigating the proper management order among multiple changes.

- **Bottom-up change management** - In this dissertation, we mainly focus on managing top-down changes. Top-down changes refer to those that are initiated by LCS owners. Bottom-up changes, on the other hand, refer to those that are initiated by service providers. Due to the dynamic environment of Web services, bottom-up changes
also occur commonly and frequently. Therefore, incorporating bottom-up change management to the current framework could be another interesting direction for our future work.
Bibliography


# Appendix A

## Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>LCS</td>
<td>Long-term Composed Service</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language for Services</td>
</tr>
<tr>
<td>OWL-S</td>
<td>OWL-based Web Service Ontology</td>
</tr>
<tr>
<td>QoSWS</td>
<td>Quality of Web Service</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SCML</td>
<td>Web Service Change Management Language</td>
</tr>
<tr>
<td>SOC</td>
<td>Service Oriented Computing</td>
</tr>
<tr>
<td>SOE</td>
<td>Service Oriented Enterprise</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Definition Language</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery and Integration</td>
</tr>
<tr>
<td>WFMS</td>
<td>Workflow Management System</td>
</tr>
<tr>
<td>WSMF</td>
<td>Web Service Modeling Framework</td>
</tr>
<tr>
<td>WSMO</td>
<td>Web Service Modeling Ontology</td>
</tr>
<tr>
<td>WSMS</td>
<td>Web Service Management System</td>
</tr>
<tr>
<td>WSN</td>
<td>Web Service Notification</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>