A WEB SERVICE COMPOSITION FRAMEWORK BASED ON INTEGRATED SERVICE SUBSTITUTION AND ADAPTATION

By

LU CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2011

To my Parent, Husband and Daughter

ACKNOWLEDGMENTS

I want to thank my PhD advisers: Dr. Randy Chow. You provided me the chance to fulfill my dream of become a PhD in one of the best universities in the world. You inspired me with your brilliant mind and trained me how to attack those difficult problems in my way. You brought me confidence to solve the problems by myself while patiently support me by pointing out even the smallest flaws. Your demeanors, your kindnesses and your ways of thinking will be the best examples for me through all my life.

I want to thank my husband, Dr. Yan Li. You are the one that share all my troubles, my happiness, my desperation and my hopes. Looking back all these years, I cannot imagine how I can gain what I have today without you. Words are not enough to express my sincerely appreciation to you. However, I still want to say that, thank you, my darling. It is you that bring me the angel: our lovely Sarah.

I want to thank my parents in China. You gave the birth to me. You taught me right and wrong. You showed me what a parent should be. Now, I am a Doctor and a mother. I will follow the examples you set to me. I will try my best to bring happiness for my family members. And for you two, you will always be the ones that I love the most.

Finally, I want to thank all the peoples I meet here. It is you that bring me all the happy time in Gainesville. Without you, I would not accomplish my PhD so smooth and productive. These memories will forever be embedded deeply in my mind and never will fade away.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

A WEB SERVICE COMPOSITION FRAMEWORK BASED ON INTEGRATED SERVICE SUBSTITUTION AND ADAPTATION

By

Lu Chen

May 2011

Chair: Randy Chow
Major: Computer Engineering

Web service composition has attracted much research interests recently. The focuses on this issue evolved from the traditional static service composition without semantics to current semantic-based adaptive composition. In addition to deal with pure service composition, other topics, including web service discovery and composition maintenance, are also integrated to generate a satisfied composite service. To accommodate existing technologies and approaches for web service composition, the main goal of our proposal is to propose an effective web service composition framework. In this framework, the composition process is divided into different stages with supporting modules and approaches. Specifically, OWL-S descriptions for web services are translated to automata representation to refine web service similarity analysis. Thus, it is able to divide a web service registry into fine-grained web service communities, which play a key role in the designing stage for our composition framework. Besides, context information is integrated into the system to facilitate web service discovery and workflow maintenance. To get a clear knowledge of how the system works with proposed approaches, a demonstrating example is shown to exhibit the characteristics of our framework.

CHAPTER 1
INTRODUCTION

With the advances of Internet and web technologies, Service Oriented Computing (SOC) is becoming widely accepted and used in both academic and industry. The goal of SOC is to utilize services as the basic constructs to support the modular development of web-based applications through composing distributed services available in a heterogeneous environment. Under this computing paradigm, questions arise as to which component web services should be chosen and how these chosen services can be combined to form the final working sets of services for a given task plan. These questions constitute the basic research themes for web service composition.

Generally, web service composition considers the issue of precise service composition as well as dynamic composition maintenance. Thus the life cycle of web service composition spans from requirement analysis, service discovery, composition and validation, to actual execution. Requirement analysis generates a formal composition plan from the user requirements. Research on requirement analysis relies heavily on nature language processing and ontology modeling of user expression. Service discovery/selection is a primary research issue in service composition since this technology provides the component service sets for the final composite web service. Abundant researches have been done on web service discovery based on different models and contexts. Composition and its validation have received much interest recently since it is necessary to verify the correctness of a composite service before it is put into real usage. Moreover, an existing composition plan might become invalid due to the changes of run time environment or due to the evolution of component services.

Thus, mechanisms must be integrated into the web service composition to cope with those dynamic changes. Finally, but not the least, context information should be considered thoroughly since a good composition plan might need to take into account the user context (e.g., preferences of user) during composition and the service context (e.g., state of web service) during execution.

Along the line of this description of the life cycle of web service composition, our research focuses on solving several sub-topics, such as service modeling, service comparison, service substitution and adaptation, all in an effort to come up with a complete framework for dynamic web service composition and maintenance.

1.1 Background

The evolution of research on web service composition comes in stages. The earlier research in the business world often used a number of XML-based standards to formalize the specification of web services, their flow composition and execution. This imprecise syntactical description of services inspires the later research by the semantic web community. Web resources with semantics are explicitly declared with preconditions and effects using ontologies. For example, the semantic web service approach aims at automating the composition process; formal logics are adopted to describe the service functionalities precisely; the classical AI planning method is used to select the best solution from a set of candidate composition plans. Though all the above approaches achieve good results in the traditional static web service composition, they are not suitable enough for adaptive web service composition which often requires some degree of dynamic/self reconfiguration.

Recently, more and more requirements are proposed for the self-manageability of web services [1, 2]. Though there is no common definition of self-management in the

domain of web services, a desired self-manageable web service should be scalable, flexible, autonomous, reusable and reliable, and thus, imposes a strict demand on composite web services. Consequently, the traditional static web services are not suitable for the self-manageable requirement, and an adaptive web service composition methodology to cope with the requirements is needed. A broader adaptive composition should be self-adjusting to most general changes during the composition process. These changes include both logic changes of the web service functionality and context changes of the target execution platform.

In most cases, changes in a web service composition process can cause exceptions or errors in the final service workflows. Therefore, new mechanisms are needed for the traditional web service composition approaches to cope with these unexpected changes. By utilizing the classical workflow exception handling approach in the workflow composition process, we aim to efficiently build composite web services and maintain the correctness of those services.

1.2 Aim and Objectives

The main goal of our research is to propose an efficient web service composition framework. This framework should possess three characteristics: 1) it should consider user preferences when generating a composition plan; 2); it should efficiently choose component services for a composite workflow 3) it should be flexible and adaptive to the computing environment. Toward these objectives, three solution approaches are proposed: the development of the notion of web service community which serves as the cornerstone of the entire service composition framework, the enhancement of service registry and discovery with static and dynamic context, and the integration of

synchronous exception handling and asynchronous service interrupt in service composition.

A valid service composition requires a facility that supports static/dynamic substitution and adaptation of services in the planning, binding, and execution of the service plan. The notion of service community facilitates the substitution and adaptation of similar services, and thus is the foundation of a service composition system. It is more than a clustering of services with similar functionalities but also matching contexts. Many service-based systems allow the provisioning of user preferences. Using various preference modeling and ranking approaches, a user is able to participate in the service selection process, and thus the chosen service can better meet the user's satisfaction. A common approach to represent users' preferences adopts context integration. In addition to users' preferences, any other information which is necessary or helpful for the service composition can also be modeled as context information. Four types of context information, including the contexts of service community and workflow, are integrated into our system to further improve the efficiency for our service composition framework.

Another important objective for web service community is to provide a better organization for existing published web services. Currently, UDDI is considered as the main standard in industry. However, traditional UDDI is mainly text-based classification and not sufficient for efficient web service discovery. Thus, web service community aims to provide a fine-grained internal categorization to augment the traditional web service registry. By integrating context information into our service community, the service query process could be made more efficient and precise.

Finally, synchronous exception handling and asynchronous service interrupt are intended to adapt to the changing environment of the system. Instead of building a new composition from scratch whenever the current composite application becomes invalid, it is more desirable to modify the outdated service and make it valid again. By integrating synchronous exception handling and asynchronous service interrupt, our proposed framework aims to efficiently generate a new composition with least efforts and overhead.

1.3 Methodologies and Approaches

Given the desire characteristics of our service composition framework and service registry/discovery schemes, four well-defined categories of contexts are proposed in our research: Service Context, Community Context, User Context and Workflow Context. Sample necessary parameters are also defined for these contexts. With context information, the following advantages are expected: 1) service selection is not only restricted to functionality comparison, but also subjected to context filtering in choosing a list of candidate services; 2) the analysis and processing of user requirements are improved in generating a more reliable formal representation of user goals; 3) the workflow generation process can cooperate with web service community to produce a composition plan efficiently; 4) the current composite service becomes adaptive and sensitive to the changing environment.

The composition of a web service community is dynamic that it should allow insertion of new services and deletion of services when they become invalid. Thus, a membership determination is necessary for operations on service community. We propose a formal approach to compare similarities between web services. In this approach, we assume semantic web services are described in OWL-S ontology, and

adopt an OWL-S API tool [3] for processing OWL-S documents. The comparison of OWL-S descriptions is translated into automaton comparison. By comparing the formal languages generated by the automata and finding their differences, we can infer the relationships between original OWL-S languages. Complementing existing similarity analysis approaches, this methodology results in a new similarity classification scheme with four finer similarity classes: equivalent, strong similar, weak similar and dissimilar. Web service communities are organized with respect to the equivalent and strong similar relationships. In this manner, service substitution can take place within the domain of the same community with greater precision.

Synchronous exception handling and asynchronous service interrupt are proposed to maintain the validation of a composition workflow. The theoretical support for these two concepts depends on current web service substitution and adaptation methods. Traditionally, the idea of web service substitution has been proposed to improve the robustness of service composition. However, current usages of service substitution are restricted to the design time of composite web services, which can't cope with the possible changing environment. Thus, we refine the traditional service substitution concepts into two levels, static and dynamic. Although in general, web service substitution reduces the cost of re-composition, it is not always necessary. Sometimes a minor modification of the service or clarification of user requirements can still maintain the usefulness of the invalidated component. This promotes the adoption of adaptation in web service composition. However, adaptation has different meanings in different stages under this context. At the user level, adaptation can be defined as a refinement of user goals [4]. At the system level, workflow developers may add some

“glue” between communicating web services to maintain the correctness of the overall system. Furthermore, during the execution of web services, adaptation can also refer to “wrapping” of invalid components with predefined modification rules making them valid again [5]. In this research, we assume all three kinds of adaptation and refine them into three groups, planning time, mapping time and binding time adaptation, respectively.

1.4 Proposal Organization

The proposed research addresses three major research topics for efficient and dynamic web service composition: similarity comparison between web services, web service storage and management, and architecture for web service composition from exiting services. Chapter 2 demonstrates the process for comparing web services and refining their similarities. In this section, we introduced a formal approach to translate OWL-S descriptions into automata representations. The automata representations are compared to reflect similarities between the original OWL-S service descriptions. Chapter 3 analyzes the inefficiency of UDDI technology and motivates the necessity to design a new organization for web services. The concept of web service community is proposed and further introduced in this section. To make the service community powerful to support web service substitution, context information is integrated into web service community. In addition, a two stage query process is also introduced for web service community based both on service functionality and service contexts. Finally, a prototype is implemented for web service community and comparisons between traditional UDDI and our service community are provided. With the illustration of approaches in Chapters 2 and 3, a complete web service composition framework is proposed in Chapter 4. Two categorizations are introduced to classify the usage of static/dynamic service substation and service adaptation in different stages. The service

adaptation mechanism is further discussed in Chapter 5. In this chapter, an integration of different adaption modules is introduced to summarize current researching achievements on web service adaptation. In addition, a novel approach, the substitution-based service adaptation, is proposed to avoid the unnecessary complexity of direct service adaptations. Finally, in Chapter 6, two applications are shown to demonstrate how the proposed solutions are integrated into a web service composition process.

CHAPTER 2
WEB SERVICE REPRESENTATION

A composite web service is iteratively composed with atomic/composite web services. Therefore, an automaton for each composite web service can be generated by composing the automata for its component web services.

2.1 Representing OWL-S in Automata

Before introducing how the composition process works, it is necessary to recognize/abstract the control constructs from an OWL-S description, which is realized by using the OWL-S API tools.

2.1.1 OWL-S Parser

To describe a web service's functionality formally and precisely, many languages exist such as WSDL, SWRL, WSMO and OWL-S [6, 7, 8, 9]. This chapter uses the OWL-S specification to describe semantic web service composition. OWL-S describes a semantic web service by providing Service Profile, Service Model and Service Grounding. Current work compared the Input, Output, Precondition and Effect (IOPE) of two web services to decide the behavior similarities among web services. However, our work focuses more on the control constructs of a composite OWL-S Process Model. A composite process specifies how its inputs are accepted by particular sub-processes, and how its various outputs are produced by particular sub-processes. Besides, it indicates its interior control structure by using a ControlConstruct.

OWL-S API is a Java API for reading, writing, and executing OWL-S web service descriptions. It provides an execution engine that can invoke atomic processes and composite processes that uses control constructs such as Sequence, Unordered, and Split. OWL-S API provides four different parsers, OWLSProfileParser,

OWLSProcessParser, OWLSGroundingParser and OWLSServiceParser, which can parse OWL-S Profile files, OWL-S Process Model files, OWL-S Grounding files and OWL-S Service files, respectively.

Currently, the API does not support conditional process execution such as If-Then-Else and Repeat-Until in its default implementation. However, this does not present a problem to our implementation since we only utilize the parser, not the execution engine. The list of control constructs can still be generated from the OWLSProcessParser.

2.1.2 Automata for OWL-S Control Constructs

Given a composite web service w, which is composed of sub-web services w1, w2… wn, to generate the automaton for w, we first define the automata for each control construct in OWL-S. Then the entire automata representation for web service can be generated by integrating sub-web services with control construct. Of all the OWL-S control constructs, the characteristics of Choice, and Split/Split + Join are intrinsically nondeterministic, thus, we choose the nondeterministic finite automaton (NFA) as the basis for modeling in this chapter.

Definition 1 The NFA model for OWL-S construct is a 5-tuple $(Q, \sum\varepsilon, \delta, q0, F)$, where

- Q is a finite set called the states. Each state is a pair (w_i, p_i) of a sub-atomic service w_i with preconditions p_i;
- $\sum\varepsilon$ is a finite set called the alphabet. Each element in the set is a 3-tuple (i_i, o_i, e_i), which means if p_i is true, the sub service w_i will accept i_i and then pass the output o_i to the next service w_{i+1} and generate an effect of e_i to the world;
- δ: $Q \times \sum\varepsilon \rightarrow Q$ is the transition function. This function describes that only if p_i is satisfied under input i_i, w_i is activated to generate information o_i and effect e_i;

- $q_0 \in Q$ is the start state, (w_0, p_0), w_0 is the first atomic process in an OWL-S description;
- $F \subseteq Q$ is the set of accept states, which are also the last/finishing processes in OWL-S description.

The transition diagram for each control construct is given in Figure 2-1. In this figure, the single circle describes the internal service. The previous and following services are omitted in this figure.

Sequence If sub-services w_i and w_j are related via Sequence control construct, the automata is shown in Figure 2-1(a).

Split In OWL-S, the components of a Split process are a set of sub-processes which are executed concurrently. The Split process completes as soon as all of its sub-processes have been scheduled for execution. If a sub-service w_i is split into two branches w_j and w_k, we create two automata, each represents one branch. It means both of the two branches are required for following execution. The automaton for Split construct is shown in Figure 2-1(b).

Split + Join The Split + Join process consists of concurrent execution of a set of sub-processes with barrier synchronization. That is, Split+Join completes when all of its sub-processes have completed. Given a sub-service w_i which satisfies the following two conditions: 1) w_i splits into two branches, which start from web services w_p and w_q respectively; 2) the two execution branches will join at service w_j, then the automata for this branching execution is simulated as sequential execution, such as $w_i w_p w_q w_j$. This is reasonable because the execution order of w_p and w_q is not important as long as both of the two branches will finish. Therefore, we define a specific node **PE** standing for a set of nodes which can be interleaved without fixed execution sequences. For example, if

PE includes w_p and w_q, then the language to describe PE can be either w_p w_q or w_q w_p. An example of such automata is shown in Figure 2-1(c).

Choice Choice calls for the execution of a single branch from a set of following control constructs. Any of the alternative branches may be chosen for execution. After the sub-service w_i, if there are two possible branches w_p and w_q waiting for execution, then the choice between w_p and w_q is nondeterministic. The automaton is shown in Figure 2-1(d).

Any Order Any Order allows a set of sub-processes to be executed without a predefined order. The execution and completion of all sub-processes is required. In addition, each sub-process execution cannot be overlapped or interleaved. Given a sub-web service w_i, sequentially followed by w_p and w_q with Any Order relationship, followed by w_j, the only constraint is that w_p and w_q are executed. Thus, similar to Split + Join, the automaton is shown in Figure 2-1(e) where PE includes w_p and w_q.

If-Then-Else The automaton for If-Then-Else is very similar to Choice, but it is deterministic. As shown in Figure 2-1(f), which one of the continuous branches after w_i is executed is determined by the value of input.

Repeat-While The web service in the Repeat-While construct is iteratively executed as long as the precondition p is true under input i. The automaton is like Figure 2-1(g).

Repeat-Until The Repeat-Until construct does the operation, tests for the condition, exits if it is true, and otherwise loops. Since the process is executed at least once, the automata are shown in Figure 2-1(h).

Since Iterate can be implemented by Repeat-While and Repeat-Until, we do not need to include it. With the above automata definitions for all the control constructs in OWL-S description, we can recursively generate the automata for a composite web service.

2.1.3 Translation of Web Services using Automata

The automata generation for an atomic web service is straightforward, and the automata generation for the composite services is based on it. Given an atomic web service w with input i, precondition p, output o and effect e. An automaton for an atomic web service w is described in Figure 2-2. The double circle describes the starting service while the black circle describes the ending service. Each state is a pair of service and preconditions, each transition is labeled with input, output and effect of the previous node.

In this automaton representation, since w has no sub-web services, it is straightforward that i = i_0, o =o_0, p = p_0, e = e_0 and w = w_0. Thus we have W = {w_0}. In addition, the start state and final state are the same since w_0 is the starting service and the ending service in w, which means F= {w_0} too.

A composite web service is recursively composed by atomic services with nested control constructs. Each control construct can be integrated with each other under the three regular operations defined in the traditional NFA theory: Union, Concatenation and Star. Figure 2-3 describes the automata composition under these three operations. Figure 2-3 (a) depicts the automaton for a single control construct C1 and C2, where (b) ~ (d) shows the resulting automaton after union, concatenation and star, respectively.

Given two automaton of control constructs C1= (Q_1, $\sum\varepsilon_1$, δ_1, q_1, F_1), and C2 = (Q_2, $\sum\varepsilon_2$, δ_2, q_2, F_2), these operations are defined as following.

Definition 2 The *union* of control constructs C1 and C2 forms a new control construct C = (Q, ∑ε, δ, q_0, F), where

- Q = {q_0} ∪ Q_1 ∪ Q_2;
- ∑ε = { (ε, ε, ε) } ∪∑$ε_1$ ∪ ∑$ε_2$;
- q_0 is the start state of C, q_0 = (w_0, true);
- F = F_1 ∪ F_2;
- For any q∈Q and a∈∑ε, $\delta = \begin{cases} \delta_1(q,a), q \in Q_1 \\ \delta_2(q,a), q \in Q_2 \\ (q_1, q_2), q = q_0 \ and \ a = (\varepsilon, \varepsilon, \varepsilon) \\ \phi, q = q_0 \ and \ a \neq (\varepsilon, \varepsilon, \varepsilon) \end{cases}$

Definition 3 The *concatenation* of control constructs C1 and C2 forms a new control construct C = (Q, ∑ε, δ, q_0, F), where

- Q = Q_1 ∪ Q_2;
- ∑ε = ∑$ε_1$ ∪ ∑$ε_2$;
- The start state q_0 = q_1;
- The accept states F of C is the same as F_2;
- For any q∈Q and a ∈∑ε, $\delta = \begin{cases} \delta_1(q,a), q \in Q_1 \ and \ q \neq F_1 \\ \delta_2(q,a), q \in Q_2 \end{cases}$
-

Definition 4 The *star* of control constructs C1 forms a new control construct C = (Q, ∑ε, δ, q_0, F), where

- Q = {q_0} ∪ Q_1, which are the old states of C1 plus a new start state;
- ∑ε = { (ε, ε, ε) } ∪∑$ε_1$;
- The start state q_0 is a new state;
- The accept state F = {q_0} ∪ F_1, which is the old accept states plus the new start state;

For any q∈Q and a∈∑ε, $\delta = \begin{cases} \delta_1(q,a), q \in Q_1 \ and \ q \neq F_1 \\ \delta_1(q,a), q \in F_1 \ and \ a \neq (\varepsilon, \varepsilon, \varepsilon) \\ \delta_1(q,a) \bigcup \{q_1\}, q \in F_1 \ and \ a = (\varepsilon, \varepsilon, \varepsilon) \\ \{q_1\}, q = q_0 \ and \ a = (\varepsilon, \varepsilon, \varepsilon) \\ \phi, q = q_0 \ and \ a \neq (\varepsilon, \varepsilon, \varepsilon) \end{cases}$

2.2 Web Service Similarity Analysis

Many researching work is proposed to measure similarity. Rather than proposing a new measuring method, our work introduces a framework which adopts the behavior similarity approach as the first stage to generate a set of possible similar web services,

then structural matching is operated on this set to improve the precision of similarity measurement.

2.2.1 Equivalence of Web Services and Regular Languages

Theorem The Process Models of two OWL-S descriptions are semantically equal if and only if their automata are represented in the same languages.

Proof: Each automaton, generated from an OWL-S description, is a NFA which can be translated to a DFA. Since EQDFA = {<A, B>| A and B are DFAs and L(A) = L(B)} is a decidable language, it is possible to determine the equality of the two DFAs. Thus, we can compare two OWL-S descriptions by comparing two corresponding DFAs. If the two DFAs are equal, we conclude that the two OWL-S descriptions are the same; otherwise, they are different.

Furthermore, each path in a DFA is a sequence of sub-processes which starts from the beginning web service and ends with the final web service. Since each path can be 1-1 mapped to a string, each DFA can uniquely determine a set of strings, which form the language of DFA. Therefore, it is reasonable to assert that two OWL-S descriptions are equivalent if and only if they share the same automata languages.

2.2.2 Two-Stage Web Service Similarity Analysis Framework

To calculate the similarity of web services with OWL-S specifications, our framework starts from the syntactic similarities for OWL-S Profile descriptions. As shown in Figure 2-5, this process begins with characterizing similar words/concepts in text and operation/process description of two web services. Similarities among text descriptions, input/output, preconditions, effects and operations can be obtained, which are accumulated to sum up the similarity value between two services. Using this process, a set of possible similar web services with mutual similarity values is produced.

However, this process is time-consuming and might not be precise due to two reasons: First, similarity values are not transitive. For example, supposing a, b, c are three web services, a is similar to b with value 0.8, b is similar to c with value 0.7, it is not precise to assert that a is similar to c with value 0.56. It is highly possible that a is similar to c with value 0.9. Second, a similarity analysis of two web services may be wrong as illustrated in Section 2. Thus, internal structural differences need to be considered to eliminate conflict services, which is the main goal for the second stage in our framework.

The input for the second stage is a set of possible similar web services. Then, an automaton is generated for each candidate in this set of services. By comparing the regular language for each automaton, it is able to find conflicted services with the following equivalence and dissimilarity definition.

For any two DFA A and B, a new DFA C can be constructed which satisfies:

$L(A)$ is the language for DFA A, and L(B) is the language for DFA B.

$\overline{L(A)}$ is the complement of L(A).

$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$.

Equivalence Web service A is equivalent to Web Service B if they provide the same functionality. This can be determined by checking whether L(C) is empty, because L(C) = φ if and only if L(A) = L(B). The equivalence relationship is symmetric, thus $Similar_{A \to B} = Similar_{B \to A} = 1$.

Dissimilarity Given two web services A and B, they are dissimilar when the languages which describe the functionality of A and B have no intersection. This is true when

$L(A) \bigcap L(B) = \phi$.

In most cases, the equivalence and dissimilarity are too strong that some services cannot be classified into this two categories, thus, refinements are proposed as following. Similarly, we define another two DFAs, D and E, with following languages:

$$\delta = \begin{cases} \delta_1(q,a), q \in Q_1 \\ \delta_2(q,a), q \in Q_2 \\ (q_1, q_2), q = q_0 \ and \ a = (\varepsilon, \varepsilon, \varepsilon) \\ \phi, q = q_0 \ and \ a \neq (\varepsilon, \varepsilon, \varepsilon) \end{cases}$$

Strong Similar The web service similarity with respect to substitution is described using containment relationship as defined in mathematics. The value for containment with respect to subset/superset is obtained from a Boolean function Contain(x, y) which has only two results, 0 and 1. This is because the containment relationship is not symmetric. Therefore,

1) SimilarA→B = Contain(A, B) = 1 if L(D) =ϕ (or L(A) $\subseteq$ L (B)), which means A is similar to B.
2) SimilarA→B = Contain(A, B) = 0 if L(E) =ϕ (or L (B)$\subseteq$ L (A) if L(E)), which means A is not similar to B.

Weak Similar Though A and B have some languages in common, they also contain conflict languages. Thus, we define their relationship as weak similar with respect to execution contexts. This consideration is practical when the Owls:Choice exist in the process definition.

For example, given the two following two automata in Figure 2-6, due to the existence of the Owls: Choice construct, the language for Automata 1 is {abc, ade}, and the language for Automata 2 is {acb, ade}. It is clear that the two languages have

conflict process execution orders, which are abc and acb. In this case, the two OWL-S descriptions are not similar to each other.

2.3 Related Work

There is an abundance of research work on deciding the similarity between two semantic web services. The Woogle search engine [10] computed the overall service similarities based on the comparison of the text descriptions of names, operations, and input/output of web services. Eleni Stroulia and Yiqiao Wang [11] used a comparison matrix to integrate similarities of data types, messages, operations and web services of WSDL descriptions to reflect similarities between two web services. Hau, Lee and Darlington [12] defined the intrinsic information value of a service description based on the “inferencibility” of each of OWL Lite constructs. Since OWL-S is based on OWL, the similarity between service profile, service model and service grounding could be computed to generate the overall similarities between two semantic web services. Wang, Vitar and Hauswirth et al [13] first built a new application ontology which was a superset of all the heterogeneous ontology. Then within this global application ontology, many existing methods, like semantic distances, can be reused to generate similarities between web services. All the above solutions only compared the functional semantics of two web services. The internal behavioral sequences of web services are not considered completely.

Recently, some researchers considered the structural sequences among web services in their web service discovery solutions. For example, to retrieve the exact/partial matched services for users’ requirements, Grigori, Corrales and Bouzeghoub [14] proposed a behavior model for web services and reduced the behavior matching to a graph matching problem. Wombacher, Fankhauser and Neuhold

[15, 16] translated the BPEL descriptions of web services to annotated deterministic finite state automata (aDFA) since BPEL itself cannot provide the state info of services. In this approach, the stateful matchmaking of web service is defined as an intersection of aDFA automata. Shen and Su [17] utilized automata and logic formalisms to build a behavior model, which integrates messages and IOPR model in OWL-S for service descriptions. They also designed a query language to integrate structural and semantic properties of services behaviors. Lei and Duan [18] proposed an extended deterministic finite state automaton (EDFA) to describe service, which also aimed to describe the temporal sequences of communication activities. Our work differed from the above ones in two aspects. First, since our goal focuses on building a web service community, it is more important to compare the descriptions between web services rather than to compare the specifications of user requirements against service description which is a key issue for service discovery. The advantage is that imprecise users' description will not affect the similarity results. Second, our work models the node of automata as a pair of sub-service and its precondition, and labels the automata transition with both input/output and effect. Thus, our work complements Lei and Duan's work since they only used the nodes to describe the states of services and state transition only considered the input/output of activities. Precondition and effects are not included in the automata model. Thus, the IOPE model of OWL-S is completely compatible with our model.

2.4 Implementation

Ideally, four components should be accomplished to demonstrate the efficiency of our similarity comparison framework: 1) OWL-S descriptions for web services; 2) translation engine to generate automata from OWL-S; 3) automata analyzer to produce

regular languages of automata; 4) language comparison. For simulation purpose, we simplify the simulation system by a) assuming all the parameters, operations, messages in OWL-S descriptions are single characters; b) assuming the text-based similarity comparison is solved, and so the single characters in a) are from the same symbol set; c) manually produce the regular languages by the users. With the above three simplifications, languages comparison are limited to String comparison and we only focus on comparing the structure of OWL-S descriptions rather on considering on text-based similarities via WordNet.

Currently, we partially finish the task to generate automata from OWL-S description. However, our work is limited to Sequence, Any Order. Due to the lack of OWLS API documentation, we are unable to support control constructs including branches, for example, Choice, If-Then-Else, Spit, and Spit-Join. Further work includes proposed a possible solution to support nondeterministic structures. In addition, due to the lack of real web services with OWL-S specification, we don't have an existing data set. Thus, we only design the OWL-S descriptions rather than implement the whole services, since it is enough to demonstrate our idea. In the future, more OWL-S specification will be designed and collected to test our solution.

2.5 Summary

This chapter proposes a formal approach based on automata to model the process behavior of web services. The primary objective is to enhance the similarity analysis of a set of substitutable services by taking the differences of the process model into consideration when comparing web services. In addition, we show that OWL-S similarity is equivalent to automata similarity. Compared with existing researches, the

proposed modeling of web service functionalities is more comprehensive and the introduction of automata facilitates the refinement of similarity greatly.

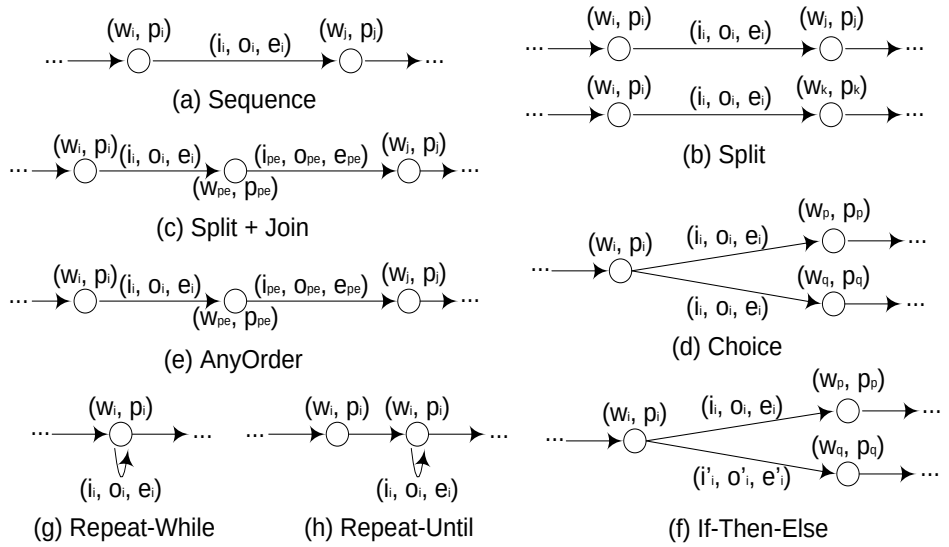

Figure 2-1. Automata representation for OWL-S control constructs

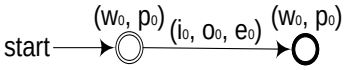

Figure 2-2. Automaton Representation for an atomic web service

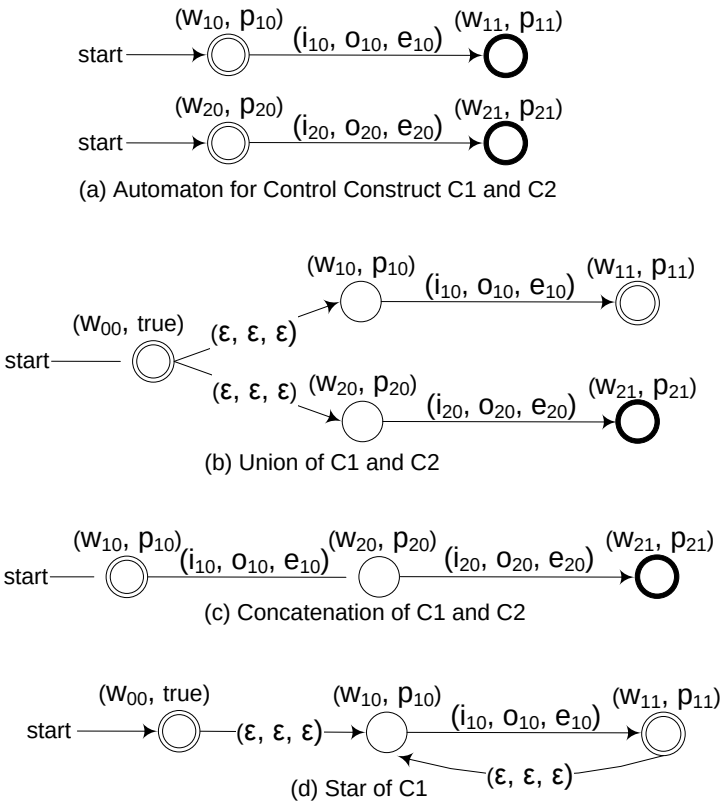

Figure 2-3. Automaton composition of OWL-S constructs

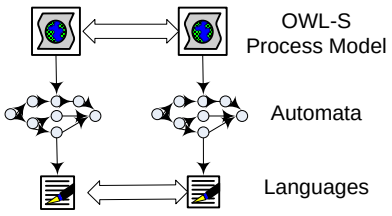

Figure 2-4. Equivalence of OWL-S Process Model and Regular Language

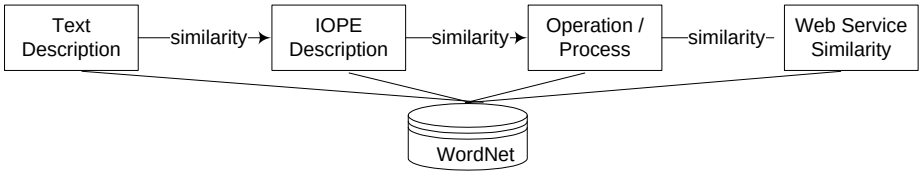

Figure 2-5. Traditional WordNet-based Similarity Analysis for OWL-S Profile

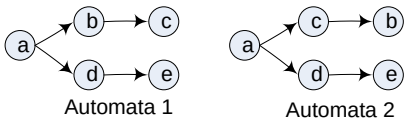

Figure 2-6. Example Automata

CHAPTER 3
WEB SERVICE COMMUNITY

The emerging service-oriented computing is becoming the main programming paradigm for the development of modern web-based applications. These applications can range from simple invocations of some web services to complex workflow systems composed of many interrelated web services. In all cases, they require a fundamental system support for identifying and locating the appropriate services they need. Traditionally, this function of service discovery is implemented through service registries. Web service providers advertise their services at public registries for service consumers. Thus, service registries are the cornerstones of a service-oriented web system analogous to the critical role of the domain name servers in the Internet. The current industry standard for web service registries is UDDI. UDDI allows simple web service look-up operations based on static and syntactic web service descriptions (e.g., service profile and interface). As we move into the next generation semantic web, it is necessary and beneficial to augment the web service discovery systems with semantic descriptions (e.g., service functionality and behavior). A semantics-based service registry allows for more precise matching between service requirement and description as well as facilitating the service discovery process.

In another software development trend, context-aware computing has gained much attention due to the desire for developing closer integration of computer systems and the physical world. Context-aware applications are software systems designed to be dynamically adaptive to the changing context of the executing environment. The notion of context-awareness has been exploited in application domains that have a strong emphasis on the physical context such as pervasive and ubiquitous computing.

Context-aware applications based on logical context (e.g., context-aware searching or information retrieval) are also becoming widespread. In the “context” of web services, the notion of context will eventually be extended from users, systems, and the execution environment to include the context of web services (e.g., the instantiations of a web service at a certain situation).

Coming from the above motivations, the primary goal of this chapter is to investigate the feasibility and open issues relating to the incorporation of context into the design of the next generation service registries and service discovery systems, where the best services can be determined based on not only the service description and functionality but also the context where they are being invoked.

The research is further driven by another critical requirement from web service composition. Compound web services can be composed from atomic or other composite web services. During the composition or execution of such composite web services, it is often necessary to find substitutes when some services are failing or not meeting the expectation. Substitutable services are similar and can be clustered into a community. The concept of service community is fundamental in systems that support dynamic web service composition. Clustering (or categorization) is also a fundamental concept in service registries (or any directory structures), except that our clustering of web services into service communities is dynamic. In other words, service communities are dynamic and have their own context.

The previous paragraphs outline the need for three different context types: User Context (U-Context), Service Context (S-Context), and Community Context (C-Context). This chapter tackles this research of integrating contexts into service registries by taking

a high-level view of the overall system, i.e., from how web services are composed, bounded, and executed, we address how service registries are to be deployed and how contexts are distributed and managed. In this sense, we add another context type, that is, the context of the composite web service in execution; we call Workflow Context (W-Context). The details of these contexts are explained later. We also discuss several design issues of the service community with respect of the above contexts. We interpret the web services' processing model as a finite state automaton and use it as the semantic-level taxonomy for the services in the community. We also discuss our community management policy using context information, including service validation, ranking and log info mining. For web service retrieval, we present the techniques on how context facilitates the precise service discovery and substitution.

3.1 Motivating Requirements

3.1.1 UDDI Extensions

Generally, to locate a specific web service, most of the solutions for web service discovery relied on a good structure of web services. However, the existing UDDI standard failed to receive wide acceptance due to many reasons. First, although the concept of UDDI was originally proposed for general web service discovery, its capability is fairly limited mainly because it supports only a single keyword-based search criterion, and thus becomes inadequate when new standards and complex query requirements emerge. Second, UDDI is a business-centric registry. It uses certain taxonomy to classify web services into different categorizes, and few hierarchies are available to manage the metadata or models of those web services. For example, existing industrial UDDI, like Enterprise UDDI Services in Microsoft Windows 2003 Server [19], jUDDI [20], IBM WebSphere Application Server [21] and Systinet Business

Service Registry [22], are proprietary service registries without a shared ontology. Third, the traditional UDDI has no autonomous self-control mechanism. It provides only minimum control on service providers when publishing web service descriptions under different categories. Thus, it is likely to cause imprecision to a registry when the provider has little knowledge of the structure or classification taxonomy about the registry. Furthermore, if a provider neglects to notify UDDI when its service is modified or no longer provided, inconsistency can occur which could be very harmful for web service binding. Fourth, but not the least, traditional UDDI has little considerations of the internal structure of web services that it cannot accommodate future composite web services with complex structures.

Since the traditional UDDI is not adequate for the state-of-art web service technologies, we propose the concept of web service community. A community of web services gathers web services that address the same users' needs and support their binding through a common interface. This service community is organized under the web service registry. Similar to the classification in traditional UDDI, web services are first organized into different registries using certain taxonomy. Then, web services within a same service registry are further clustered into different service communities based on their similarity comparisons with other services. Details about this classification and organization of web service community along with the management of community are discussed in later sections.

3.1.2 Precise Binding

Service binding is the process of instantiating logical components in composite workflow into concrete web services. It is more than the traditional service discovery of finding an existing web service which meets a user's requirements. In our UDDI

refinement, the service binding process is separated into two stages, community mapping and service binding, respectively. After a logical composite workflow is generated, components of this workflow are first mapped to respective web service communities containing a set of services with similar functionalities. This process is the main goal for the community mapping process. Once a certain community is chosen, service binding aims at finding the most suitable service in this community to instantiate the logical components in the composite workflow.

3.1.3 Service Community in Web Service Composition

With the incorporation of web service community and supporting contexts modeling, we propose the architecture of web service composition as shown in Figure 1. In this flow diagram, a user's requirement is passed into a composition engine to generate a set of logical composite workflows {LWi}. Meanwhile, U-Context is abstracted from the user's requirement to support the ranking and selection processes for a top-ranked LWj to participate in the Community Mapping process. In this process, each component in the chosen workflow LWj is mapped to a web service community. To achieve efficient and precise community mapping, W-Context and C-Context are used to reduce the search space for improving search performance. A physical composite workflow PWj is generated when all components in LWj have been mapped. In the next step of service binding, concrete web services within a service community are chosen to produce an executable composite workflow. In this step, C-Context and S-Context are used to help improve the service query process. Details on community mapping and service binding are again illustrated in Sections 3.5.

3.2 Context Modeling

The contexts associated with the service communities can be classified into 4 categories: S-Context, C-Context, W-Context and U-Context. S-Context describes information about a single web service and is relatively static. C-Context describes the contexts associated with each individual web service community. They are modified dynamically whenever 1) the community management task registers or de-register a service; 2) the autonomous historical log mining process starts. U-Context describes the service user and the user's requirement. It remains static unless a user explicitly changes it because of workflow feedback. W-Context describes the contexts of the intermediate logical and physical workflows. These four contexts support functionalities provided by the service communities, including service classification, management and query. The details of how these contexts are modeled and presented are introduced as the following.

S-Context describes the contexts of a single web service. Whenever a web service is bounded to a workflow execution, its instantiation runs in the remote server. To facilitate the correct binding and execution, a set of plausible parameters that S-Context may consist of are described in Table 3-1. Thus, the Service Profile in OWL-S description in the proposed web service composition architecture is extended to include these parameters for generating the S-Context. Note that the S-Context information is less likely to be changed unless the service is updated to a new version. When a service provider registers a service into a service community, S-Context is published along with the functional description of the service. With the S-Context in local storage, the service binding process can be done within the service community.

C-Context describes the contexts of a web service community with associated web services. Unlike the static information recorded in S-Context, these contexts focus on the relationships among different services. They are available only when services are registered in the community. Table 3-2 gives some examples of C-Context. C-Contexts are obtained from statistical analysis of the history logs on the community management and query operations. They are changing from time to time with more data collected during the routine operation of the registry. Since the C-Contexts are only valid within the scope of service community, they are stored as auxiliary information in the corresponding service registry or communities.

Use of U-context and W-context helps to facilitate query processing and result refinement in requests for web services or service communities. Tables 3-3 and 3-4 provide a list of useful U-Context and W-Context, respectively. U-Context specifies the user's context such as user's expected cost, expected performance, location and other constrains.

With U-Context, we can distinguish users from each other and use their preferences to refine the service selection process. W-Context mainly describes the constraints and execution environment for a composite workflow. The workflow queries the service registry to find a web service component that can fulfill its functionality requirements. Besides the functional specification, the query request also specifies the context such as its neighborhood services, security environments, and whatever the expected service is going to fit into. Moreover, to cope with the emerging dynamic web service composition [23, 24, 25], modify W-context in the service substitution process would further reflect the dynamic nature of the composition.

3.3 Community Organization

Since the traditional UDDI is not adequate for the state-of-art web service technologies, we propose the concept of web service community in this chapter. A community of web services gathers web services that address the same users' needs and support their binding through a common interface. This service community is organized under the web service registry. Similar to the classification in traditional UDDI, web services are first organized into different registries using certain taxonomy. Then, web services within a same service registry are further clustered into different service communities based on their similarity comparisons with other services.

Traditional UDDI is a centralized service directory for publishing web service descriptions. The green pages in UDDI help businesses to catalogue published web services according to their tModels, which describe the various business, service, and template structures stored within the UDDI registry. However, service classification using tModel only is insufficient to support today's automatic discovery where there exists a huge amount of web services. A fine-granularity internal structure is needed to better organize the old UDDI Business Registry (UBR). In this section, we assume an initial UBR is already generated for web service descriptions represented in OWL-S.

The key idea behind dividing an UBR into sub directories is that, tModel-based classification is rough; services under a same UBR may have conflict functionalities. Thus, this chapter proposed the concept of web service community under an existing UBR. The classification of web services is mainly based on the similarity comparisons between web services. In our previous work on web service similarity comparison [26], OWL-S descriptions of web services are transformed into automata representation. By comparing the languages of the generated automata, service similarities can be

calculated. In this scheme, we defined four similarity relationships: equivalent, strong-similar, weak-similar and conflict. This chapter on community-based categorization focuses on the equivalent and strong similar relationships.

With these two definitions of the similarity relationships among different web services, an existing UBR is re-organized in Figure 3-2. In this structure, each web service in a service registry is classified into only one web service community based on its functionality. The internal organization for a service community is analogous to a max-heap. In this tree-like data structure, the root node for a max-heap is the starting node for searching the community when no context information is available. Each community is associated with some relationships, except communities containing one web service. When more services are inserted into a community, its associated relationships will be updated.

Figure 3-2 shows three service communities, A, B, C. Community A is organized based on both relationships: equivalent and strong similar. The equivalent services are linked into a list as shown in solid arrows, while the strong similar services are linked to its parent nodes represented in dashed arrows. For instance, service a is the root service for community A, and it is linked to two services b and c, that are functional equivalent to a. The functionalities of service d and f are subsets of service a, thus, their relationships to a are strong similar. For community B, there is only one relationship, which is equivalent. For community C, it has only one service. Thus, no relationship is available for community C.

3.4 Community Management

The composition of a service community changes dynamically. A new service can be added and registered into an existing community or form a new community. When

the web service validation mechanism detects that a service is outdated, the service should be removed both from the community and outer registry. When S-Context of web services changes, the C-Context of community containing those services should also be changed correspondingly. Insertion and deletion of web services are two primary operations for community structure management.

3.4.1 Insert a Web Service into a Community

Given a web service registry R with existing communities c1, … , cn and assuming a new service s is added into R, the first step is to choose a community to register this service. This procedure is depicted in Figure 3-3. It first compares the root service in each community, once a root service is strong similar with s, (which implicitly imply that the community contains also the equivalent relationship), the selection process stops and the community containing this root service is returned. If no root service is available, an empty community is returned.

When a community is chosen to insert the new service, the service registration process in Figure 3-4 is invoked. This insertion algorithm simulates the process to insert a new node to a max-heap. Since service s can be added into a community only when it is equivalent to or strong-similar with the root service of the current community, it is guaranteed that s can either becomes a new root (lines 4-6) or be inserted into the equivalent list of certain service (lines 7-10) or the sub-tree of current service (lines 11-16). In particular, lines 11-13 describe the scenario that s is only strong-similar to service i, but implicitly weak-similar to any child service of i. Lines 14-15 show that the functionality of s is between i and j, while line 16 means s is also strong similar to children service of i.

3.4.2 Delete a Web Service from a Community

Once a service is removed from a community, it might trigger a community split process. This is because only equivalent and strong-similar web services can be linked together to form a community. The siblings are weak-similar. Thus, it is necessary to split and create new communities when deleting the root service, which has no equivalent service. Figure 3-5 describes the community split process. When a community is ready to be split, the root of the tree representation of this community has no equivalent services. The number of children of the root is calculated to decide how many new communities are to be created (lines 6-7). Then, each sub-tree of the old root service is assigned to a new community (lines 8-10).

The service deregistration process is divided into three cases, based on the position of node n in the tree as shown in Figure 3-6. Lines 4-8 describe the operation when n is a leaf node. Lines 9-14 illustrate the process when n is the root service in the community. Specifically, a service deregistration can cause a community split, as shown in lines 10-11. Finally, lines 15-21 shows the operations when n is an intermediate node in the tree.

3.5 Community Query

To instantiate a logical workflow into an executable workflow, each component in the logical workflow must be bounded to a web service. The binding process consists of two steps, community mapping and service binding.

When processing a service query request, a community level search is first conducted to identify a community of services that fulfill the required functionality. In this search process, a workflow component description is first sent to a service registry. Based on the functionality-based comparison, a set of candidate service communities

with the required functionality are identified in this registry. It is followed by a context-based filtering of the candidate communities. The registry first collects the C-Contexts of the candidate communities. Second, it retrieves the U-Context and W-Context from the composition engine. These contexts are evaluated together to select the most suitable community. This step is illustrated in the upper part of Figure 3-7.

After a community has been chosen in a registry, this community can conduct the traditional service discovery of its containing services. Many existing approaches exist for service discovery [27, 28, 29]. Our framework has no specific preference on any one of them. Based on the functionality-based service selection, a set of potentially feasible services is identified. The system selects an appropriate service based on all four different contexts in the system. The lower part in Figure 3-7 shows this process. First, it eliminates the services that are not available for current query. For example, the number of current running instances of the service has reached the limit specified by the service vendor, or the security requirement given by the service could not be satisfied by the workflow. Then, a scoring function is called to determine the best candidate based on the context information, e.g., choosing services with the nearest location or the lowest price or some combined criteria of both.

A set of evaluation criteria can be specified by the user, and weight for each evaluation criteria can be set to customize the evaluation policy. Scores of each available candidate service computed for all criteria are summed as its total score according to their weights. The service with the highest score is selected and will be instantiated by the workflow as its components. In this manner, users can assign the scores according to their own contexts and current mapping status of the workflow by

customizing the detail of the evaluation policy at run time. Some default evaluation policy can also be provided by the service registry in case the user does not want to set any policy.

One noteworthy observation is that, in the view of service registry, the mapping operations for service composition and service substitution are of no difference as they can all be accomplished by query requests. However, the query results can be cached either in the workflow or at the registries, the substitution query can skip the community level search, i.e., the resulting community of the previous query can be applied directly.

3.6 Related Work

Web Service registry has long been an important topic in web service research. UDDI is the current industry standard for web service discovery. However, UDDI is pure syntax based and does not support any ontology and semantic information. Hence UDDI is not suited for categorizing semantic web services and incorporating context information. Recently, several researchers have proposed solutions to improve the traditional UDDI. Examples including adding semantics to UDDI [30, 31], merging decentralized UDDIs into a federated one [32, 33], associating UDDI with service QoS attributes [33, 34], etc. However, none of them incorporate ontology and context information into the web service registries as we do in our system.

Yamato [35] describes an abstract design of a semantic web service registry (meta-data DB), but the registry's taxonomy is not specified in detail. Incorporating context information for the enhancing web service performance is discussed by the authors in [36]. They use independent context plug-in to refine the output of web services before they are returned to the users. The authors also propose a very important concept that context processing should be loosely coupled with the web

service itself for better flexibility and preciseness. Appling context information in the service composition process is widely discussed by researchers. In [35, 37], the authors present a framework for web service evaluation using the context of users and web services. However, only static contexts such as like location and similarity are considered. Unlike our system, dynamic contexts that related to service runtime environment, such as service association and priorities, are not addressed. In [37, 38], they employ software agents to store and process context information during composition. Three types of context are defined for the service composition process from the viewpoint of service composition engine. In this chapter, the community context is added so as to reflect the view of the service community. Furthermore, we apply some data mining concepts for discovering the association among service categories and specific services. This concept of web service mining has generated much interest recently. For example, a web service mining framework is proposed in [39] that automatically discover interesting web service compositions. In [40], data mining technique is applied to web service composition logs to improve the quality of work flow design.

3.7 Discussion and Implementation

3.7.1 Problem Discussing

When query a web service against a given service registry, the first step is to choose a proper web service community. This step, which is called community mapping, is accomplished by comparing the automata of the required service against the functionality of each root service in the web service registry. This mapping process can improve the query efficiency since it decreases the search space of the service registry. Thus, the efficiency of the community query process depends on two factors:

1) the representation of the community root service; 2) the number of web services in each web service community. There are a couple of fundamental design questions:

Are the web service communities balanced, i.e., uniformly distributed?

One of the primary goals of web service community is to enhance the web service discovery process in UDDI. This is achieved by refining the traditional service registry into fine-grained web service communities. The classification criterion depends on the functional similarities among web services. Thus, it is possible that different service communities contain quite different number of web services. If one service community contains most of web services while others contain only a few, the search space is still large and the query process is not optimized. Therefore, balance among web service communities is important in the system. However, the balance property is only partially maintained via the split operations. No explicit operations are proposed to guarantee the fairness for two reasons: 1) no merge operation is provided in the system, which is explained later; 2) the insertion order of web services may affect the membership of web service communities because different inserted services may cause different split operations and then result in various service communities.

Why there is no merge operation proposed for the web service community?

It is reasonable to assume that a merge operation is needed for the construction of web service communities. However, this operation is not provided in the system for the following considerations. First, the cause for the merge operation is not unique. For example, an inserted new web service could become a new root service for more than one web service community. In this case, those communities should be merged to form a new one. The problem is that, it will generate a large community which breaks the

balance among other communities. Another example is that, a split operation can generate small communities, which may be subsets of other communities in reality. However, determination of this subset relationship is time-consuming. Second, the structure of the generated community depends on the order of merge operations of sub-communities. This is because the membership determination only relies on equivalent or strong similar relationships. For example, given four web service communities with root service a, b, c and d. If a and b are strong similar to both c and d ($a \subseteq c$, $a \subseteq d$, $b \subseteq c$, $b \subseteq d$), and c and d are only weak similar to each other ($c \cap d \neq \phi$, $c \not\subset d$, $d \not\subset c$), there exists four alternative merge operations: 1) a is merged to c, and b is merged to d; 2) b is merged to c, and c is merged to d; 3) a and b are merged to c; 4) a and b are merged to d. Thus, the result is not unique. Since the complexity of the merge operation outweighs the advantage, it is excluded in the design of our service community system.

Is the root node the most representative service in the community?

In the community organization structure, it is guaranteed that the root service of each community is the most representative one. This property is not affected by any of the community operations even when the structure of service community is altered. 1) In the service registration process, the new service is always compared to the root service at first. If the functionalities of the two services are equivalent, the new service is inserted to the equivalent list. If it is strong similar to the root service, the new service will either become a new root service or recursively compared to the children services to find its place in the heap-like structure. Thus, the root service is always the most representative one. 2) In the service deletion process, if the root service is deleted, either a service from the equivalent list is chosen to be the new root service or the

community split process is triggered. In the first case, the new root service is obviously the more representative than any other services. In the second case, since each child node of the deleted root service is weak similar to each other, none of them is qualified as a new root service. Thus, the original community is split into several new communities, each of which still maintains the heap-like internal structure. Thus, each root service in these new formed communities still remains its representativeness.

3.7.2 Community Functional Blocks

To show the feasibility of the concept of web service community we demonstrate a prototype implementation. The functional blocks for the web service community system are demonstrated in Figure 3-8. For simplicity, a workflow specification is represented as a composition of regular languages of required web services. The web services contained in a service pool are data records that are described in automata. All the functional blocks inside the service registry module are implemented in Java on a Windows 7 PC.

When a workflow is sent to the execution engine, its required concrete web services are discovered and bound to the workflow. This task is accomplished via following blocks.

Workflow API: This block receives the binding request from the workflow execution engine, and extracts the functionality of the required service in an automata format. It also cooperates with the Context Manager to obtain the W-Context. These two results are passed to the Query Processor.

Context Manager: Four types of context information exist in the system and are used in the entire service query process. The Context Manager is the module that extracts context information and provides it to other modules.

Query Processor: When the functionality of a required service is obtained, Query Process matches it against the root services of all service communities. This step is called community mapping. It selects a matched community for further discovery process.

Community Management: The proposed service community operations are built in this module, including community selection, service insertion, service deletion and community split.

3.7.3 Community Implementation

3.7.3.1 Complexity Analysis

To evaluate the improvement of our web service community over the traditional UDDI, we analyze the complexity of different operations over the two structures. In addition, a simulation is implemented to demonstrate the performance difference. UDDI is a standard and has many different implementations in industry. For simplicity, in our simulation we assume a linear list implementation of it. Table 3-5 shows the operational complexity analysis between the two structures based on the above assumption. In this table, big N is the number of web services inside a UDDI directory, c is the average number of communities inside a UDDI directory, and small n is the average number of web services inside a web service community.

Due to the linear list structure of UDDI implementation in our work, all the operations except the insertion on UDDI have to traverse the entire list to locate the target web service in the worst case. Thus, each operation has O(N) time complexity. For web service communities with the max-heap like organization, the first step is to choose a specific service community prior to locate the target web service. For

example, to insert a new web service into a registry, the system has to choose 1 out of c communities. This step costs O(c) time. Then, this new service is inserted into this chosen community with O(*lg* n) time complexity. Thus, the final time is O(c+*lg* n). For community operations, it is necessary to separate Delete-Max from random Delete operation. This is because delete a root service without equivalent backups can cause a community split process. Since the root node of a heap is fixed, the complexity is O(1).

3.7.3.2 Performance Evaluation

From the table, it is observed that most operations have linear complexity for both UDDI and community. However, since N = Σ(i<=c)ni, generally, n+c is considered less than N. Thus, we can expect better performance of community operations over UDDI operations. To verify this idea, a simulation is conducted. The details are shown below.

Web Service Data Set Generation

For each web service, we provide three descriptions: key words, regular languages and context information. Thus, the data ranges for the three aspects are designed. For example, for key word specification, we provide a range of 10 characters, from A to J. To specify the range for regular languages, we assume each character is associated with one interval, which is called minimum identifier. For example, A is associated with {1, 3}. Thus, there are totally 10 intervals provided for the 10 key words, which are IA = {A1, A2}, IB = {B1, B2} … IJ = {J1, J2}. The integers for the start and end values of intervals are randomly distributed and selected. The only requirement is that each interval can be mapped to a unique key word. Thus, the entire 10 intervals are disjoint with each other, which means Ii∩Ij=Φ, A≤ i, j ≤ J. For context information, we provide two parameters: cost and number of available instances. The values for the two

parameters are randomly generated integers. With the above data ranges for each aspect, a web service is generated in the following steps:

1) Each web service is randomly assigned with only one key word to simplify the simulation;
2) A function is designed to generate the regular languages for a web service. This function takes the minimum identifier of the service key word as input. It extends the intervals of the key word by randomly decreasing the starting value and randomly increasing the ending value. For example, key word A is assigned to a web service, then the regular language for this service is f (A1, A2) = [A1 - m, A2 + n], 1≤ m, n ≤ 10. This function is necessary to make weak similar relationship exists among generated web services.
3) For each web service, we require that it either has both context parameters or has no context information. The cost is a random integer from 1 to 30, and the number of available instances is from 1 to 5.

Therefore, an example description of web service without context information is {{A}, {3, 8}}, while one with context information is like {{C}, {13, 22}, {30}, {5}}. These generated web services are inserted into the end of a linear list for the UDDI. With service community, we follow the insert algorithm to generate a set of service communities.

Query Generation

A query request describes the functionality of a desired web service. Thus, it possesses the same descriptions as the web services: key words, regular languages

and context information. The data range is the same in each aspect. However, there are three types of query provided.

For UDDI, it uses the key word query. Thus, UDDI query requests have only one aspect. An example query request is {C}.

For a web service community, it considers also the functionality of web services. Therefore, its query request has two aspects: key words and regular languages. However, rather than designing a function to generate the regular languages as adopted in generating a web service, the minimum identifier is sufficient. This is because we can increase the number of retrieved web services to be filtered by context constraints. Thus, when a key word is randomly assigned to a query, its interval is adopted directly as the regular languages. In this case, the query has the format {{M}, {M1, M2}}, $A \leq M \leq J$. An example query requirement is {{C}, {2, 4}}.

For service communities with context information, the query request contains the entire three aspects. Thus, the query request has the same pattern as a web service description. One example is like {{A}, {3, 8}, {30}, {9}}.

Query Process

For each type of query request, we provide a corresponding query processor. A UDDI query processor traverses the linear list of web services and returns the first service which has the same key word as the query request.

A service community query processor compares the regular languages of a query against the languages of root services for communities. Since the community is a max-heap like structure, if the language for a query request is a subset of the language for a root service, the community is chosen and the comparison is performed against its

member services. This comparison inside a community is implemented via doing a BFS search over the member services to find the one whose language is a direct superset of the required service. If the language for a query request is a superset of the language for a root service, then the root service is returned directly and it implies that no equivalent service is available inside the community. Note that, each interval is mapped to a unique key word when designing the data set. Thus, queries on regular languages implicitly contain the key word query. Therefore, we can expect better performance for community query.

In addition to the functionality level comparison, a context-aware community query processor integrates a context-level comparison. As introduced above, a web service either has none context information or includes both context parameters. Thus, when a query request with context requirement is received, only a web service which satisfies both the number of instances and cost is returned. This is determined by :1) the number of available instances of the retrieved web service is greater than one; 2) the maximum cost in the query request is greater than or equal to the cost of the retrieved web service.

Experimental Results

To verify the refinement of web service communities over the traditional UDDI, we compare the average accepting ratio and query time of the two structures. Figure 3-9 and 3-10 shows the results.

Figure 3-9 describes how the accepting ratio is affected by the average number of web services which share the same key word. A retrieved web service is accepted when it can satisfy all three aspects, which means: 1) the key words for the query request and the retrieved web service are exactly the same; 2) the regular languages

for the query request and the retrieved web service are equivalent or strong similar; 3) the context information of the retrieved web service can meet the context constraints of the query request. In Figure 3-9, the average number of services that share the same key word can affect the disambiguity among web services. Generally, the larger the number is, the harder to find the accepted service. Figure 3-10 describes how the average query time for 1000 query requests changes when the number of web services increases.

From Figure 3-9, it is observed that the context-aware community has the best accepting ratio, UDDI has the worst result, and community query is between the two. This is because context-aware community query considers all three aspects; in particular, it increases the precision significantly by filtering out weak similar services. Since it does more comparison to improve the result, we expect it to take longer time than community query without context information as shown in Figure 3-10. Another observation in Figure 3-10 is that, UDDI query costs less time than context-aware community query when the query size is small. When the number of web services grows, context-aware community query exhibits better performance. This is because UDDI query compares the requirement against the entire service set, while community query reduces the number of service by refining the query process inside a specific community.

3.8 Summary

The traditional UDDIs are becoming inadequate to support the growing web services, which exhibit increasing complexities in their internal structures. This chapter proposes a framework for implementing future web service registries which incorporates

both semantic and context information to enhance the preciseness and efficiency of service registration and discovery. The key concept in the framework is the use of service community as an internal sub-structure in the registries. Integrating service communities into a workflow of web leads us to consider a comprehensive context model that includes four types of useful contexts: Service Context, Community Context, User Context and Workflow Context. The first two contexts help the hierarchical organization of web services while the later two contexts optimize the searching process in service community. The prototype implementation demonstrates the effectiveness of using service community in service registry and service composition.

Table 3-1. Parameters of S-Context

Name	name of the service
Location	location of the published service
Cost	cost/charge to use the service
Number of Instances Allowed	ability of the service to process requests in parallel
Constraints	precondition, assumption, post-condition and effect of using the service
Date	time of last update of the above parameters

Table 3-2. Parameters of C-Context

Identifier	name of the service
Registry	ID of the registry inside which the community resides
Num of Services	number of web services within the community
Similarity	indicates whether the element services are equivalent or strong similar
Community Query History	historical information of the next queried community in the same registry
Community Association	historical information of the next queried community in different registries
Running Instances	number of instances that a service has been invoked
Next Instance Availability	indicates whether a service is ready for instantiation
Service Query History	historical information of the next queried service within the same community
Service Association	historical information of the next queried service in different communities
Date	time of the last update of the above parameters

Table 3-3. Parameters of U-Context

Name	identity of a user
Cost	maximum cost, if any, of the user
Performance	expected response time of a web service
Constraints	the IOPE requirement of the user
Security	expected security level of the user
Location	user's location
Preferences	user's preferences over a list of services
Date	time of last update of the above parameters

Table 3-4. Parameters of W-Context

Num of Services	number of services participating in the composition plan
Current Mapping Community	It describes the current mapping community so that: 1) it restrains the community to select binding service; 2) it specifies the location for rebinding substitution services when the binding service failed.
Next Mapping Community	the next mapped web service community from which a binding service can be chosen
Current Running Service	It describes the current running component service. The workflow might not have finished binding all component services before execution. It reflects the idea of building on-the-fly service composition.
Next Binding Service	next component service to bind
Computing Resources	current available computing resources to facilitate mapping and binding selection
Date	time of last update of the above parameters

Table 3-5. Complexity Analysis of Management Operations

	Insert	Delete	Update	Delete-Max (Split)	Query
UDDI	O(1)	O(N)	O(N)	N/A	O(N)
Community	O(c+lg n)	O(c+n)	O(c+n)	O(1)	O(c+n)

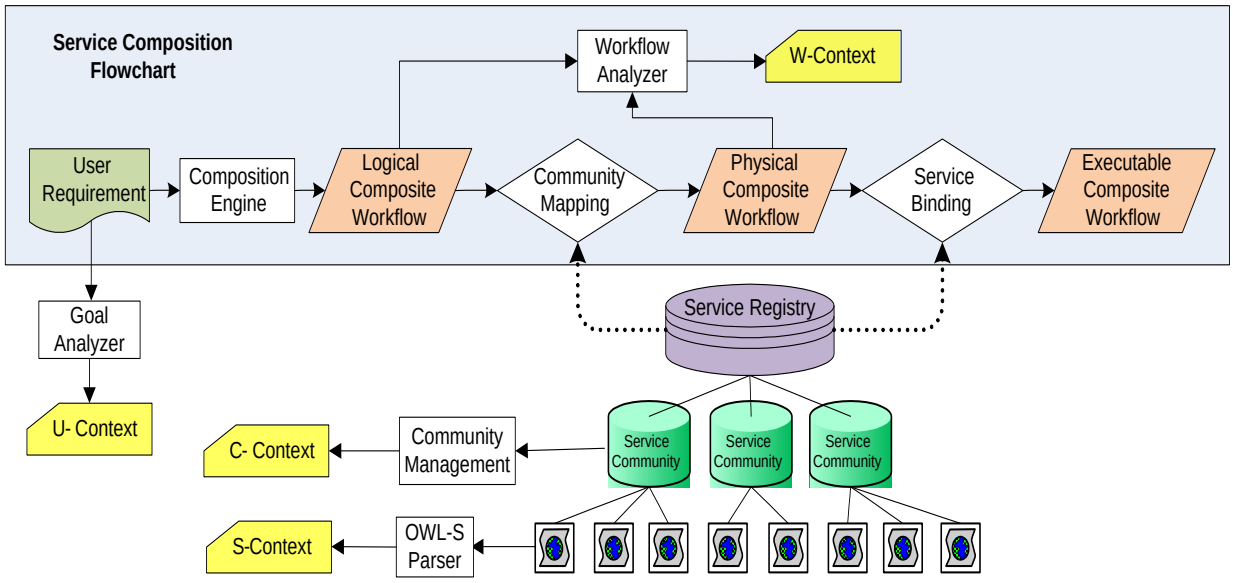

Figure 3-1. Web Service Composition with Context and Community Support

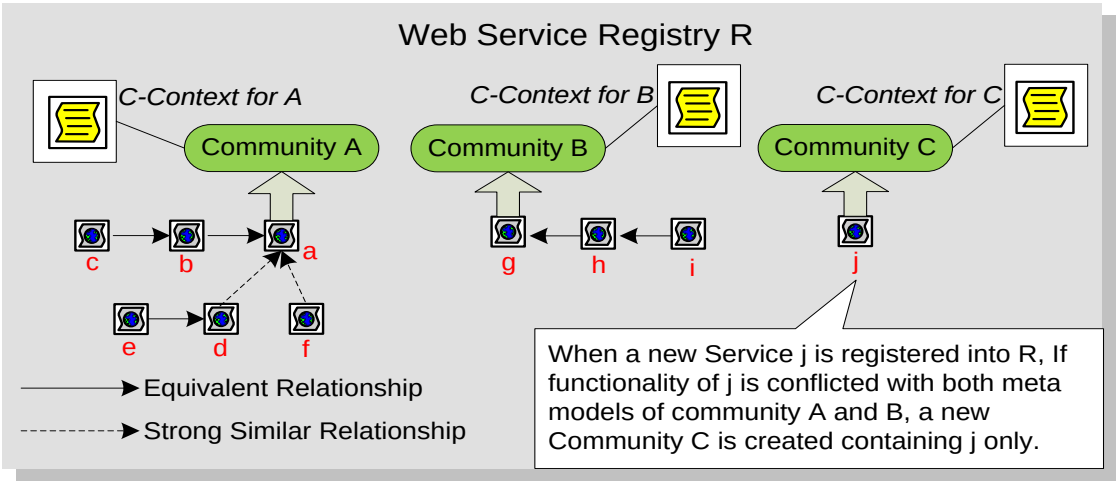

Figure 3-2. Re-organize a web service registry R into service communities

```
1  Input: community C1, C2, …,Cn, service s
2  Output: community Ci
3  Procedure: community_selection (List C )
4  For each root service sr_i in community Ci
5    Compare the automata of sr_i with s
6    If strong similar
7    Then return Ci
8  End For
9  If no Ci
10 Then building an empty Ci and return
```

Figure 3-3. Community Selection in a Registry

```
1  Input: tree t, node i, service s
2  Output: tree t
3  Procedure: service_registration(t, i, s)
4    If s is a superset for t
5    Then s becomes a new root for t, and link node i to s
6           Return t
7  Traverse the tree t from node i
8    Compare the similarity of s and current node i in t
9    If s is equivalent to t
10   Then add s into the linked list of i and return t
11   Else compare s with child nodes of i
12     If no strong similar with any child nodes
13     Then add s as a new child node of i and return t
14     Else if s is superset of child node j
15           Then insert s between i and j, and return t
16          Else service_registration (t, j, s)
```

Figure 3-4. Insert a service into a service community

```
1  Input: community Ci
2  Output: community C1, C2, …,Cn
3  Procedure: community_split(C)
4  IF root r has no children
5  THEN delete Ci and return null
6  ELSE calculate the number of children stored in N;
7          make N new communities C1...CN;
8          Assign Ci with the subtree ti of r;
9          delete C;
10         return C1...CN;
```

Figure 3-5. Split a Service Community

```
1  Input: tree t, service s
2  Output: tree t
3  Procedure: service_deregistration()
4  IF node n is a leaf node in tree t
5  THEN IF n has no equivalent services
6           THEN delete n and return C
7         ELSE link the first service in the equivalent  service list to the  parent node of n;
8                 delete n and return C;
9  ELSE IF n is a root node
10                THEN IF n has no equivalent services
11                        THEN community_split()
12                ELSE choose the first service in the service list as the new root node n';
13                        re-link the child nodes of n to root n';
14                        delete n and return C
15        ELSE IF n is an intermediate node
16                THEN IF n has no equivalent services
17                        THEN re-link child services of n to n's parent service;
18                                delete n and return C;
19                        ELSE link the first equivalent service in n's list both to parent
20                                node and child  nodes of n;
21                                delete n and return C;
```

Figure 3-6. Delete a service from service community

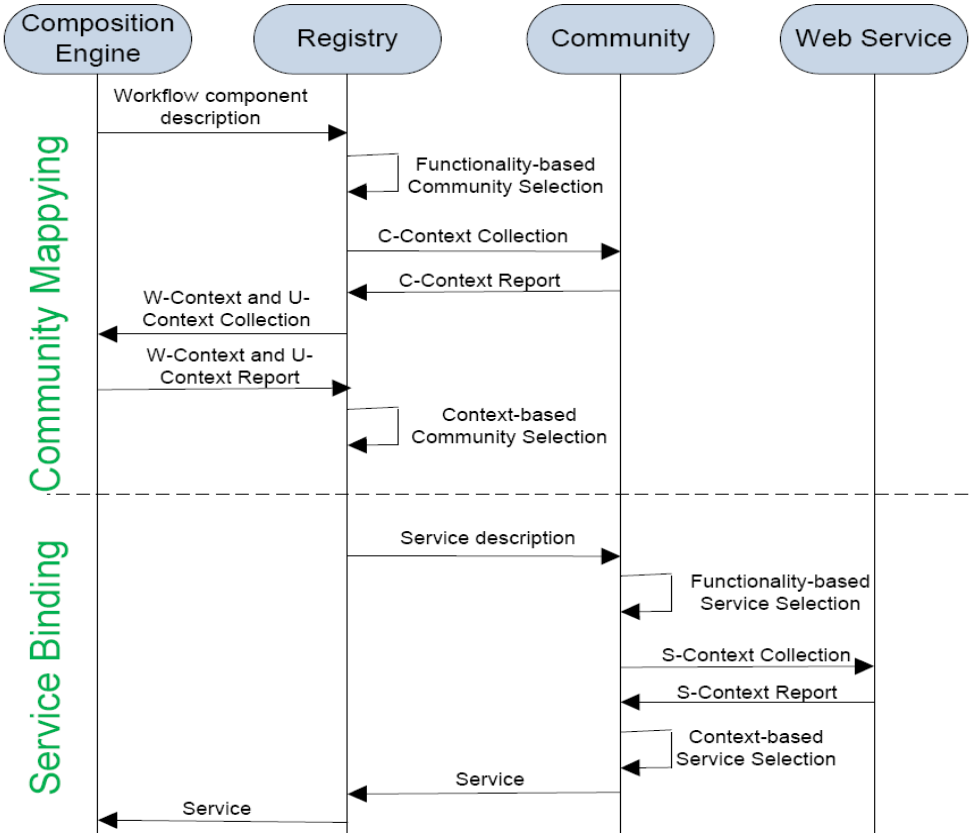

Figure 3-7. Community Query Conversations

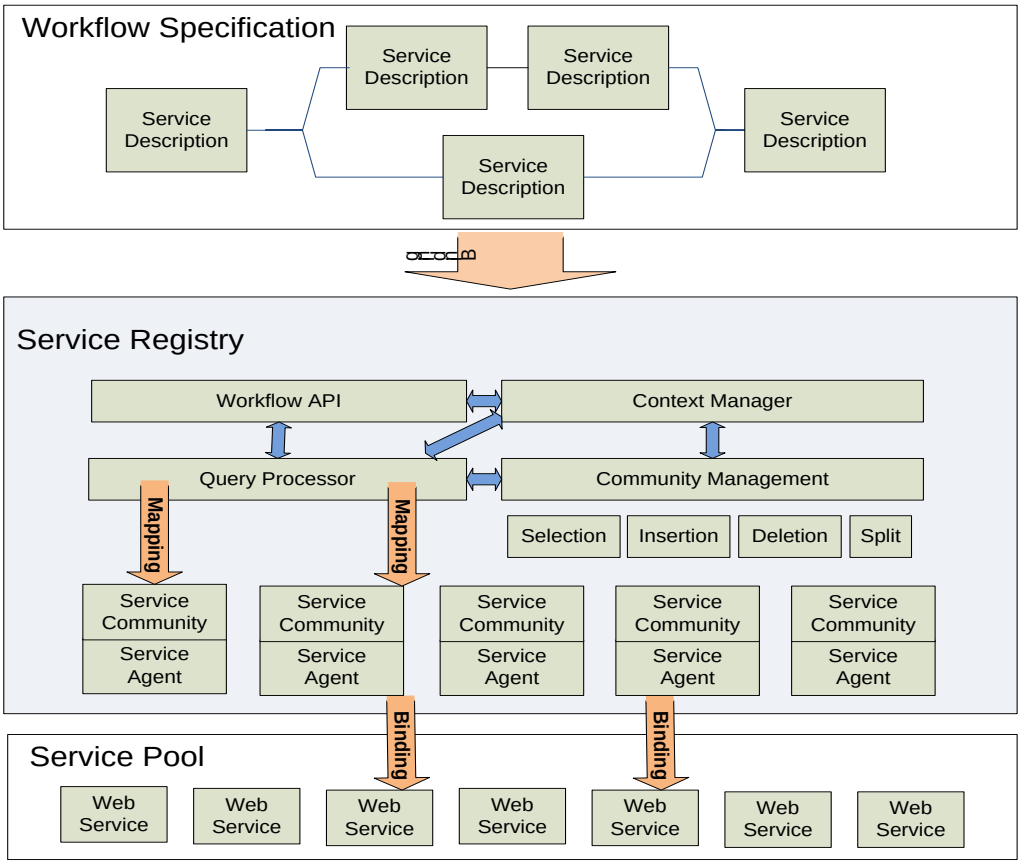

Figure 3-8. Web Service Community Functional Blocks

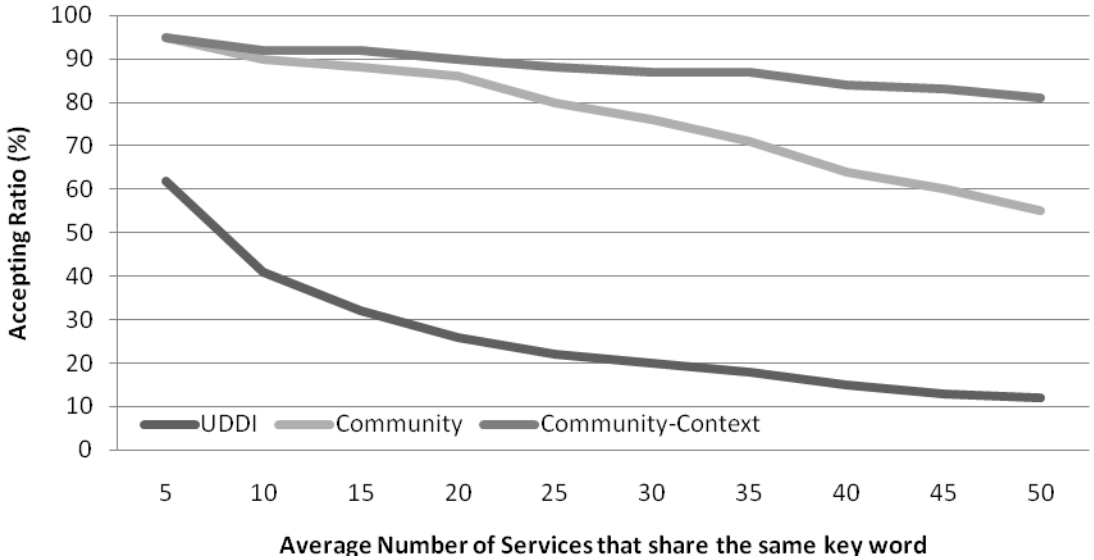

Figure 3-9. Accepting Ratio vs. Number of Services

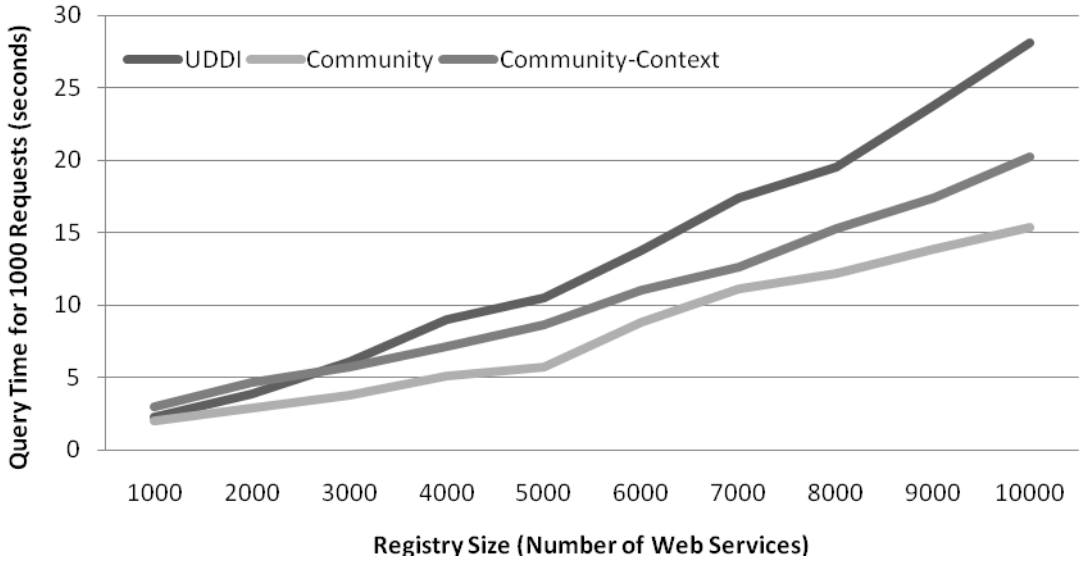

Figure 3-10. Query Time vs. Registry Size

CHAPTER 4
WEB SERVICE COMPOSITION FRAMEWORK

Recently, more and more requirements are proposed for the self-manageability of web services [1, 2]. Though there is no common definition of self-management in the domain of web services, a desired self-manageable web service should be scalable, flexible, autonomous, reusable and reliable, and thus, imposes a strict demand on composite web services. Consequently, the traditional static web services are not suitable for the self-manageable requirement, and an adaptive web service composition method to cope with the requirement is needed. A broader adaptive composition should be self-adjusting to most general changes during the composition process. These changes include both logic changes of the web service functionality and context changes of the target execution platform.

In most cases, changes in a web service composition process can cause exceptions or errors in the final service workflows. Therefore, new mechanisms are needed for the traditional web service composition approaches to cope with these unexpected changes. By utilizing the classical workflow exception handling approach in the workflow composition process, we aim to efficiently build composite web services and maintain the correctness of those services.

Our solution contains two part, synchronous exception handling and asynchronous service interrupt, respectively. The theoretical support for these two concepts depends on the traditional web service substitution and adaptation methods. Traditionally, the idea of web service substitution has been proposed to improve the robustness of service composition. However, current usages of service substitution are restricted to the design time of composite web services, which can’t cope with the possible changing

environment. Thus, we refine the traditional service substitution concepts into two levels, static and dynamic, respectively. Although in general, web service substitution reduces the cost of re-composition, it is not always necessary. Sometimes a minor modification of the service or clarification of user requirements can still maintain the usefulness of the invalidated component. It promotes the adoption of adaptation in web service composition. However, adaptation has different meanings in different stages under this context. At the user's level, adaptation can be defined as refinement of user goals [13]. At the system's level, workflow developers may add some "glue" between communicating web services to maintain the correctness of the overall system. Furthermore, during the execution of web services, adaptation can also refer to "wrapping" of invalid components with predefined modification rules making them valid again [9]. In our work, we assume all three kinds of adaptation and refine them into three groups, planning time, mapping time and binding time adaptation, respectively.

Various solutions have been proposed for the web service composition problem. In general, most of those solutions used staged processes to generate the final composite services as shown in Figure 4-1.

In this staged web service composition framework, a user's specification of a required service is abstracted and decomposed to machine understandable sub goals. Then these sub goals are passed to the composition generation stage, where an executable web service composition plan is produced and put into the composition execution stage. Currently, many approaches are used in the composition generation stage, as introduced in the previous related work. However, several problems exist for the generated composition. First, most composition generation approaches use the

static knowledge of existing services to generate a composite service, e.g., the classical AI planning method is based on complete static service knowledge. Second, little attention has been paid to web service composition with timing constraints. For example, some states of web services are held only for a period of time (e.g., the sales price is kept for a very short duration.) Third, the composition generation system may be unaware of failures of already finished partial workflow caused by unanticipated changes, and thus it is incorrect to continue the generation stage. Lastly, it is possible that the participating services in a composite service workflow were changed or failed during the composition execution. Instead of re-composing the required service from the scratch, it is desirable to avoid failures and enhance the correctness of the resulting composite workflow at various stages of the entire composition generation and execution process.

To solve the above problems, the following Figure 4-2 shows our ideas. It extends the staged web service composition in Figure 4-1 by including adaptive reconfiguration of composition in the generation and execution stages. During a composition generation process, anticipation of internal events can be embedded into the process, thus, the handling of these events are termed as Synchronous Exception Handling (SEH). Similarly, during a composition execution process, external events that could affect the execution can be treated like interrupt, and the mechanism is termed as Asynchronous Service Interrupt (ASI).

4.1 Synchronous Exception Handling

In the composition generation stage, failures can occur at any time. Wrong users' specifications, incompatible service interfaces, unmatched service functionalities, etc., can all cause failures. These failures should be divided into two groups: real failures and

workflow exceptions. Real failures refer to the absence of a valid composition plan with existing services, no matter what technologies are adopted. On the other hand, exceptions mean that an invalidation of the current workflow is temporary and this problem can be fixed by the system automatically. For example, a shipping service is composed of three component services: car, train and airplane. If all the component services are not available at the users' specified time, real failures happen. Otherwise, there is only an exception since one of the three services is available and can substitute the other two services. In order to solve failures caused by workflow exceptions, we refer to the traditional workflow exception handling. Particularly, this exception handling is refined into two sub processes. First, for a failed service in the partial composition, a substitution process replaces it with the most suitable candidate from the same web service community automatically. Second, an adaptation process uses "glue" rules to bridge the gap between sequential WSDLs in the composition plans so that we can still maintain a combination of communicating web services to satisfy users' requirements. The following sections introduces the above two methods in details.

4.1.1 Static Web Service Substitution

In general, substitution means replacing one service component with another, as long as the replacing component produces the same output and satisfies the same requirements as the one being replaced [41]. Web service substitution could be due to non-responsiveness to client requests or better arrangement with another competitor Web service.

In our work, we call the traditional service substitution as static substitution since it happens in the composition generation stage. Many web service composition studies considered static substitution and its correctness. Static service substitution can be

implemented by first verifying the equivalence between two services, and then substituting one with the other. Li and Jagadish represent BPI (Business Process Interaction) model of web services using graphs [42]. Their goal is to determine the compatibility and difference of two web services represented as graphs. In this sense, service substitution is treated as a graph homomorphism problem. We can also consider substitution as a new task of composition, whose requirement has the same functionalities of the old invalidated web services. This type of substitution was incorporated in the work by Hamadi and Benatallah [43]. They used Petri-net models to analyze the reachability of the web service workflows and to deduce the satisfiability of the composed web services. Both substitution approaches improve the efficiency of web service composition. Our proposed approach differs from others in refining the web service substitution into two types, static and dynamic.

Pathak, Basu, etc. [44] propose a specification-driven approach for the web service composition. The framework allows users to start with an incomplete specification of a goal service. If the required service cannot be satisfied by existing services, the system identifies the cause for failures which can then be fed back to the developer to reformulate the goal specification. In this way, the notion of adaptation is considered. They extend their service compositions by focusing on the problem of context-specific service substitution which requires that some desired properties of the component being replaced are maintained [45]. Two variants of the context-specific service substitution problem are introduced, namely, the environment-dependent and environment-independent substitutability, which relax the requirements for substitutability relative to simulation or observational equivalence between services.

Their work gives a formal foundation of substitution and achieves a kind of correctness. However, communications between services are not solved in their work. Moreover, their work didn't consider the dynamic changeover of web services. Taher, Fauvet, etc. [41] adopt the idea of deploying communities of web services. One major advantage of the concept of web service community is that it is easier to find a candidate of peer web service for substitution when needed.

In our proposed approach, static substitution refers to the replacement of similar web services at design time. It is performed during the generation of a web service composition plan. When the web service discovery mechanism is requested, it replies with a list of candidate abstract web service descriptions (AWSs). The AWSs are ranked by their closeness in meeting the user's requirements. The system chooses the highest ranked AWS. From its definition, an AWS can be mapped to several WSDL descriptions, from which the most suitable WSDL description is chosen to be included in the composite plan. If the generation fails, the system picks the next ranked WSDL from the same community to regenerate the plan. If none of the WSDL descriptions are suitable, the process will return to the previous AWS selection stage. Finally, each WSDL is bound to a concrete server in the final composite plan.

4.1.2 Planning Time Web Service Adaptation

There is no commonly agreed definition of adaptation with respect to web service composition, although adaptation is a very generic concept used in many different contexts. Our proposed adaptation method is motivated by the work by Chafle, Dasgupta et al. They proposed a staged method for web service composition based on adaptation [5]. In our approach, we also adopt the staged approach, but with significant differences. When failures happen in each composition stage, the control flow first

transfers to the substitution process instead of to the adaptation process in their system. More specifically, we define our adaptation in three levels: planning time, mapping time and binding time.

The planning time web service adaptation is needed when the service composition fails at the composition planning time. Such failures could be due to inability of finding the desired web services or imprecise information for composition. The paper addresses the latter case. When using traditional AI planning techniques for web service composition, the general assumption is that the system must have a complete knowledge of existing web services before generating a plan [46]. Naturally, this assumption is not realistic for real-world applications since a web service might change and its functionalities might not be completely specified. This is an example of imprecise information caused by either poor service provider's specification or the system's inability to extract precise information from the specification. Similarly, imprecise information could be due to incomplete or wrong description of the user goals. A solution to the imprecise user goal specification is to allow a feedback process for the refinement of user goals such as the framework MoSCoE [47]. Our planning time adaptation approach for service composition aims to deal with the difficulties resulted from both kinds of imprecise information.

For the implementation of our proposed web service composition system, we complement the AI planning approach with planning time adaptation mechanisms. First, instead of accepting imprecise goal descriptions from a user as inputs to the planning machine directly, the system uses heuristic methods to refine those imprecise specifications iteratively, similar to the work in [4]. Second, the system is extended to

periodically update the current information in the AI planner's knowledge base, which maximizes the completeness of the knowledge of existing web services. Finally, a concept similar to check-pointing and recovery is added. When some web services possess values which are conflicted with their default values, the system logs the event and lets the planning processes continue. These logged records can be used to check the system's knowledge of web services when failures occur.

4.1.3 Mapping Time Web Service Adaptation

Comparing with the planning time adaptation process which helps users to refine their goal descriptions about web services, mapping time web service adaptation is transparent to the user. Mapping time adaptation is complementary to static substitution and it happens if static substitution fails. As discussed in the previous sections, when mapping AWS to the highest ranked WSDL fails in the workflow generation process, static substitution is activated to choose another similar WSDL. If failures remain, the system will return to the previous AWS selection stage. At this time, the control flow will transfer to two concurrent branches. The system can drop the current AWS and choose the next ranked AWS to resume the mapping again. Simultaneously, mapping time adaptation is triggered to adapt the current AWS. By doing so, the choice of candidate web services for the system is maximized. This adaptation is regulated by certain adaptation rules. At this time of research, these adaptation rules are mainly "glue" rules which bridge the gap between WSDL descriptions and allow the component web services to communicate effectively with each other.

4.2 Asynchronous Service Interrupt

In the composition execution stage, the availability and context constraints of the concrete (actual) services in a composition workflow are checked. The conflicts could

impede the binding of a physical workflow to the concrete services to be executed. The problem can be resolved by either 1) dynamic substitution which substitutes an invalidated service with its backups or 2) binding time adaptation which works like a “wrapper” that encapsulates an invalid web service making it a valid candidate again. In order to maximize the effectiveness of the two corrections, the system divides the context changes into two groups: unrecoverable changes and recoverable changes.

Unrecoverable context changes invalidate the current workflow permanently. It forces the composition system to go back to the previous stage for a new set of abstract/physical composite workflows. Therefore, nothing can be done in the runtime stage. In contrast, recoverable context changes just delay the composition execution; the system can return to a right state via certain modifications. Such “interruptions” to the composition execution caused by the recoverable context changes are generally related to the web services providers or the service execution environment.

The occurrence of ASIs can be effectively detected by the system. Since the execution of an instantiated composition is monitored internally in the composition execution stage, any execution exceptions and sharp performance degradations are fed back to the execution module periodically as service interrupts. In addition, most service interrupts are detected when there are explicit external events from the web services. These triggering events may include: 1) the unavailability of concrete services, 2) the context conflicts between participating services and execution environment, 3) the functionality updates of participating services by their providers. Respective solutions are provided for each situation. In case 1, the system restarts execution after the invalid services are replaced by their back-ups. In case 2, a resumption response is generated

after rebinding WSDL to similar services, which satisfy current contexts. These are the work of the dynamic substitution. In case 3, the updates include increasing, changing or removing functionalities by the providers. Currently, solutions are provided only for functionality incensement by offering required services to users while unnecessary behaviors are hidden from the current workflow. And that is responsibility of the binding time adaptation process.

4.2.1 Dynamic Web Service Substitution

Dynamic substitution can be triggered if one concrete web service, which participates in at least one of the running composite plans, becomes invalid or conflicts with a user's QoS requirements during execution. This problem can be addressed by rebinding WSDL to concrete web services. Though rebinding of services to servers may seem difficult and time-consuming since it requires the correctness verification of the choreography and orchestration constraints in most cases, it is valuable under two considerations. First, comparing with the compulsory overall workflow verification when re-compose from the scratch, the verification of workflow after dynamic substitution is reduced only to the changed part of the workflow. Second, substituting a web service with its replicas can maintain the choreography and orchestration constraints in general. Replication of web services can be done at different nodes over the Internet, which is similar to data replication in databases. Those replicas possess the same WSDL descriptions, state transition diagrams, and QoS constraints. Any single copy of those replicas has the same behaviour in one executable workflow. Meanwhile, its interactions with other web services will remain the same even if it is replaced by its replica. Therefore, substitution of web services with replicas does not contradict with the choreography and orchestration constraints. This dynamic substitution is highly efficient

since the only work is to substitute the invalidated one with its corresponding backup and redo part of the computation before the execution is resumed. Figure 4-3 demonstrates the idea. The horizontal bars represent the constructs in WS-BPEL, and the vertical bars indicate the instantiation of the corresponding web services (solid lines in the Figure 4-3).

4.2.2 Binding Time Service Adaptation

Binding time adaptation happens in case of dynamic substitution failures. Generally, a simple implementation of dynamic substitution is to find a back-up component web service to replace the invalid one during runtime. However, not all web services have back-up replicas. In addition, invalidation of web services caused by functional conflicts cannot be solved by replicas. Therefore, functionality failures of dynamic substitution demonstrate the need for binding time adaptation. In this case, binding time adaptation works like a “wrapper” to encapsulate an invalid web service making it a valid candidate again. This candidate will then be compared with other candidates and the best ranked one will be put into runtime execution. We adopt traditional software wrapping techniques for binding time adaptation as the work in [48]. Another potential need for binding time adaptation could be due to non-functional requirements such as violation of QoS or time constraints. This is an open issue for web service composition to be considered in our next phase of research.

4.3 System Architecture

4.3.1 Overall Framework

As motivated and analyzed in the previous sections, we combine several key concepts for web service composition including service discovery, substitution, adaptation and AI planning. Figure 4-4 depicts the overall architecture of the system.

The flow of the dynamic composition process is organized in three stages. The Logical Composition stage contains two constructs. The knowledge base (KB) stores the descriptions of Abstract Web Services (AWS) extracted from the extended UDDI. The AI plan Generator generates a set of logic workflows from KB denoted by {Lw}, which is sorted by the rank information. Meanwhile, the knowledge base refreshes itself periodically for the most current knowledge. The second stage, Physical Composition, also contains two constructs. The functionality of mapping is to map the AWS in {Lw} to WSDL descriptions, which results in a set of executable workflows {Ew} sorted by the rank information. Static substitution facilitates the mapping process by finding the equivalent web services in the same web service community for candidate executable workflow generation. The last Runtime stage chooses the most suitable workflow from the previous stage and binds this workflow to concrete web services. The criteria for filtering are decided by the rank mechanisms and QoS requirements. The result is a particular executable workflow Ew, which will be put in the Execution Environment.

The ranking rules for ranking web services or workflows are extracted from users' preferences or non-functional requirements. At each stage of the above framework, the feedback information is collected and sent to the previous stage for better arrangement. For example, in the Execution Environment, the running workflow is monitored and any degradation in performance or conflicts in requirements will be returned to the Runtime Stage. Therefore, dynamic substitution or binding time adaptation will be triggered to produce a new executable workflow. If this process fails, feedback will be returned to the previous stage to fix the problem. This process continues iteratively until the system re-enter a correct state.

4.3.2 Extension to Service Oriented Architecture

The system architecture is complementary to the traditional Service Oriented Architecture (SOA). SOA provides methods for system development and integration. It allows users to combine and reuse distributed online services for developing new web applications. Traditionally, SOA organized a large application as a collection of component services. These sub services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. However, possible exceptions or errors exist in the communication processes between those sub services. Our proposed synchronous workflow exception handling and asynchronous service interrupt extend the traditional SOA by providing a mechanism to facilitate the communication and maintain the validity of the final composition plan. That is to say, the traditional SOA produces an initial plan for service composition, and then SEH and ASI are used to refine and maintain this plan.

4.4 Related Work

Currently, abundant researches are focusing on web service adaptation and related topics. For example, much work discusses the cause of adaptation, such as incompatibility between service signatures and behaviors. Lucas, Gwen and Daniela, etc, defined and analyzed two concepts: compatibility and substitutability in the view of static properties and dynamic behavior [49]. They thought that a simple description of the service behavior based on process-algebraic or automata-based formalisms can help detecting many subtle incompatibilities in their interaction. Benatallah, Casati and Toumani discussed the different ways in which the middleware can leverage protocol descriptions and focuses in particular on the notions of protocol compatibility, equivalence and replaceability [50]. They characterized whether two services can

interact based on their protocol definition, whether a service can replace another in general or when interacting with specific clients, and which are the set of possible interactions among two services. To formally detect incompatible services, an algorithm was proposed to determine the compatibility between web services in [42]. The algorithm takes two graphs as inputs. Each of the graphs represented the external interface of a web service. Then the system can tell whether the two services are compatible or not and also the differences between two graphs. Thus, the result can be used by a service to enable dynamic collaboration between each other. With the existing achievements on web service compatibility analysis, many researchers proposed different solutions for web service adaptation. For example, Benatallah, Casati and Grigori, etc. characterized the problem of adaptation of web services by identifying and classifying different kinds of adaptation requirements, such as the heterogeneity and the higher levels of the interoperability stack and the high number of clients supporting different interfaces and protocols [51]. Thus, the authors proposed a methodology for developing adapters in web services based on the use of mismatch patterns and service composition technologies. In their further researches, the authors proposed an Aspect oriented framework a solution to provide support for service adaptation [52]. Recently, they proposed an approach to identify mismatches between service interfaces [53]. In addition, they also identified all ordering mismatches between service protocols and generate a mismatch tree. Then, semi-automated support in analyzing the mismatch tree is provided to help in resolving such mismatches. Brogi and Popescu presented a methodology for the automated generation of service

adapters capable of solving behavioral mismatches among BPEL processes [54]. A key ingredient of their work is the transformation of BPEL processes into YAWL workflows.

Research on web service composition has drawn much attention in the web service community. Skogan, Gronmo and Solheim [55] model the composition of web services using a UML Activity Diagram. This diagram is translated to a preliminary interface model and a composition model of the new web service. These models are then translated to executable BPEL (Business Process Execution Language). However, their selection of candidate services is based on textual description of web services which are not precise enough. Moreover they do not consider potential future changes of services. Chafle et al. allowed a staged approach for adaptive Web Service Composition and Execution that clearly separated the functional and non-functional requirements of a new service, and enabled different environmental changes to be absorbed at different stages of composition and execution [5]. Yoji and Hiroshi established a flexible staged service composition framework, where a semantic-level service scenario was translated and its components can be dynamically found, selected and bound [56].

4.5 Summary

The novelty and significance of this chapter is the integration of the concepts of synchronous workflow exception handling and asynchronous web service interrupt to facilitate a valid web service composition. A valid web service composition means that the integrated composite service remains executable even if the environment changes after the plan is already generated. To maintain the validity of composite service, we incorporate the static/dynamic substitution and adaptation methods into the composite workflow generation process. The static substitution, the planning time adaptation and

the mapping time adaptation are categorized as synchronous exception handling, while the dynamic substitution and binding time adaptation are treated as asynchronous web service interrupt.

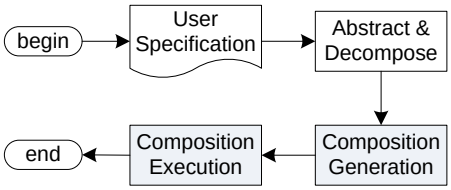

Figure 4-1. Web Service Composition Framework

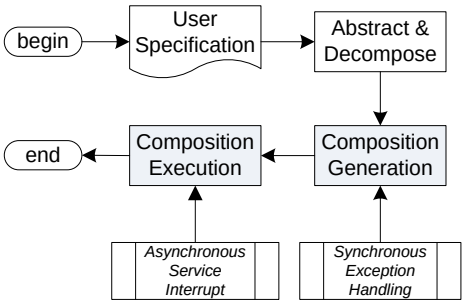

Figure 4-2. Service Composition Process with Changes

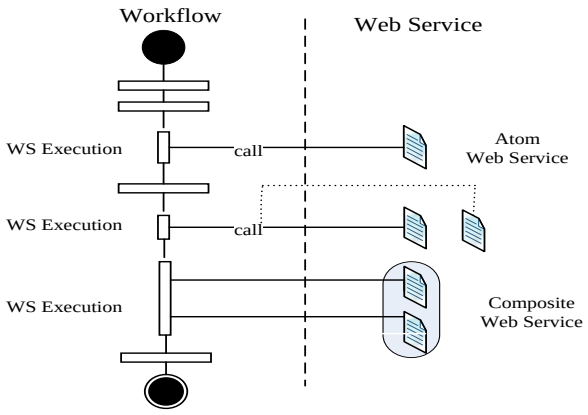

Figure 4-3. Dynamic Web Service Substitution

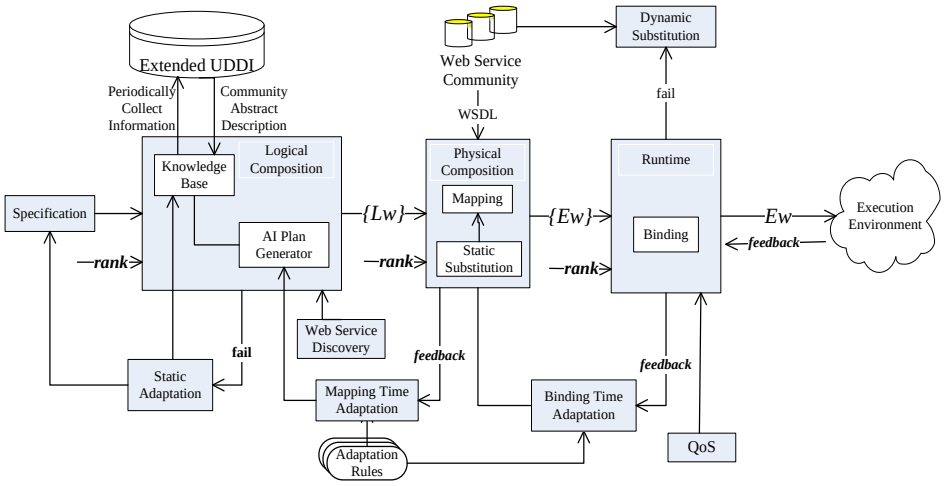

Figure 4-4. System Architecture

CHAPTER 5
SUBSTITUTION-BASED CONTEXT-AWARE SERVICE ADAPTATION

Web services are self-contained software which are distributed online and can be accessed via various protocols. A key aspect of Service Oriented Computing (SOC) is to integrate web services in a loosely-coupled fashion to build new applications efficiently. Building a suitable integration is the main goal of web service composition. Primary research on service composition is concerned about constructing a composite workflow from existing web services that matches the user's goal. However, most available solutions are generated at the design time and assume static information. These solutions might not be precise due to the intrinsic dynamic characteristics both for web services and their execution environment. Thus, even when a composite workflow is theoretically valid and executable during the design time, no guarantee can be provided to produce the expected results in the runtime environment. Therefore, new mechanisms need to be studied for managing dynamic web service composition.

Recently, the concept of web service adaptation has been widely studied for dynamic web service composition. In many cases, designing a proper adapter for a composite workflow is a key issue since 1) message exchanges between web services are often expressed in different format and granularity; 2) autonomous web services are generally heterogeneous and continues to evolve; 3) invocation sequences of internal service activities are often different; 4) mismatches between web services are not only at the level of individual interactions, but also across interdependent interactions [57]. To address these problems, various solution approaches have been proposed which include compatibility analysis [42, 49, 50, 58], replaceability/ substitutability analysis [45, 50, 59], mismatch pattern analysis between service interfaces or behaviors [52, 60, 61,

62, 63], and process mediation [64, 65, 66, 67, 68], etc. Complementary to the existing approaches, recently, integrating context information into web service composition has attracted much attention [69, 70, 71]. By considering context in the service adaptation process, researchers can predict more possible changes at runtime and generate automatic solutions to deal with these situations. A synergetic integration of the concepts of context-awareness and service adaptation achieves a holistic effect on the overall success of dynamic service composition.

Following this trend for context-aware web service composition, we propose a context-aware adaptation framework to integrate context-awareness into the existing adaptation systems. In addition, we observe that, sometimes the traditional adaptation methods are too costly due to the complex adaptation process. Therefore, this chapter also proposes a light-weight adaptation approach based on web service substitution. Adaptation can often be accomplished through modular substitutions without complex changes of web services. Effective substitutions rely on the support of the concept of web service communities, which are grouping of similar web services. The main contributions of our work are 1) maintaining the interoperability of communicating web services in a composite workflow; 2) coping with the dynamic availability of component services and runtime environment changes.

5.1 Adaptive Web Service Composition

Various research issues have been widely studied for adaptive web service composition, including web service modeling, web service registry, web service discovery, etc. Many frameworks have been proposed for service composition, including our previous solution [26, 72, 73]. Generally, the user's requirements are passed to a composition engine. The engine analyzes the requirements and generates composite

BPEL processes from registered web services. However, those BPEL processes are often unbound, i.e., they are depicted with only abstract web services rather than concrete ones. Dynamic binding is conducted when a BPEL process is sent to execution engine. In this process, one observes that dynamic binding needs the support of a well-organized service registry to find candidate concrete services when context changes. Thus, the concept of web service community was proposed to refine the current web service registry and support the web service substitution process. A traditional service registry like UDDI is coarse grained and not precise enough. Services within the same registry can be classified into fine-grained communities based on service similarities. With addition of the concept of service community, a common adaptive composition workflow can be extended with dynamic binding as shown in Figure 5-1.

In this figure, the white boxes represent the traditional workflow for adaptive web service composition, while the grey modules are our extensions. First, a web service registry is further refined to web service communities, each of which contains functionally similar web services. When choosing the concrete services for an unbound BPEL process, the Static Binding module selects a proper web service community rather than concrete services to generate the pre-bound BPEL processes. An abstract service proxy is generated to represent this chosen service community. While putting those pre-bound processes into the BPEL Execution Engine, the Dynamic Binding process selects the most suitable concrete services in the pre-selected web service community. A concrete service proxy is generated for each concrete web service when communicating with users. Context Monitor collects runtime information and triggers re-

binding if necessary. This two-stage dynamic binding implements the functionality of the service query. This enhanced system gains the following benefits: 1) the pre-execution static binding of web service communities rather than real web services provides more alternative services and can improve the availability of concrete services during runtime; 2) when one web service fails at runtime, re-selection of substitutable services can be implemented by limiting the search space within the same web service community; 3) the service proxies hide the details of service adaptation, making it transparent to the outside.

As introduced in the previous section, adaptation can be achieved using different approaches, including resolving interface and behavior mismatches in the dynamic binding process by “wrapping” the differences. However, this “wrapping” process is sometimes expensive and its overhead cannot be justified. This observation is analogous to fixing the computer. It is most likely easier and less laborious to replace the malfunctioning modules than to repair them. Similarly, with the explosive number of online web services, substitution-based adaptation is practical and costs less effort than direct adaptation. Our existing work on web service community provides the data-source for service substitution. The general service adaptation framework is introduced in Section 5.2 while the substitution-based adaptation is detailed in Section 5.3.

5.2 Service Adaptation Framework

Numerous architectures have been proposed to model the service adaptation process. Generally, these architectures involve different sub-modules, such as adaptation model [74], mismatch pattern [52, 60, 61, 62, 63], adaptation template [60, 75] and adaptation trigger [76]. However, to the best of our knowledge, no existing work has integrated those modules together to build a complete system. In addition, context

information has not been seamlessly combined with the adaptation process. Thus, this chapter improves current research solutions by integrating various adaptation components into a whole framework and extends it with context information, as discussed in following sub-sections.

5.2.1 Overall Framework

Generally, when a composite BPEL process is executed, the execution engine will monitor its results and generate adaptation requirements if necessary. These adaptation requirements, together with user's requirements and the outdated workflow, are sent to an adaptation system to synthesize an adaptation specification. A well designed adaptation system is the key module in the web service adaptation process. The architecture for our framework is shown in Figure 5-2.

The *Adaptation Model* describes a set of predefined adaptation rules and the contexts requirement when referring to those rules. It is possible that an execution sequence exists among different rules. In addition, some rules can have pre-/post-conditions for execution. Thus, contexts are not restricted to describe the execution environment, but also include the dependencies and constraints.

To analyze service incompatibility, classification of *Mismatch Pattern* has been widely studied in the research domain of service mediation, which includes interface mismatches and behavior mismatches. Moreover, these patterns can be further refined into syntactic (structure), semantic, functional and non-functional. Traditionally, the *Adaptation Trigger* sub-module involves a set of adaptation events which aim to determine when to trigger the adaptation process. An improvement is added to this module by integrating the *Context Monitor* to set up the thresholds for context and monitor their changes. To generate adaptation specification automatically, the

Adaptation Template module provides code or pseudo-code that describes the implementation of an adapter that can resolve the difference captured by the Mismatch Pattern. These templates are generally predefined via different mechanisms, like Aspect-Oriented service adaptation [52, 77, 78]. In addition, they can be further updated with the system's feedback information. Finally, these four modules communicate with the Reasoning Framework to generate the final adaptation specification in the following sequences:

1. When an adaptation requirement comes into the Adaptation Specification Generator, the system analyzes the requirement and consults the Mismatch Pattern module to detect the differences between current workflow and required functionality.
2. Given the produced differences, the Adaptation Module can choose a set of proper adaptation rules with corresponding context requirements. The dependencies and constraints relationships among rules can eliminate conflicting rules.
3. With the provided mismatch pattern and adaptation model, the execution goes to the Adaptation Template module to select a proper template for adaptation.
4. The results from step 1 to 3 are sent to the Reasoning Framework to generate the final adaptation plan, which waits for the execution signal until threshold is met.
5. The Adaptation Trigger monitors the contexts of the workflow execution and listens to specific adaptation events to trigger the adaptation plan in step 4.

5.2.2 Context-aware Service Adaptation

The goal of web service adaptation is to modify an existing web service to fulfill some requirements that were not originally designed for. Although this modification process has been well studied, issues involving the incorporation of context information into the service adaptation process for higher accuracy and efficiency of workflow execution remain open. We argue the need for context-awareness and illustrate its importance in the following example.

Consider the case of adapting a library query service into a warehouse query service. In addition to the adaptation of the innate features such as functionality and UI, the differences between the two contexts where the services are running must also be considered. The original library query service may be running in a more open and publicly accessible environment. The requested throughput may also be high but the reliability of the service and the query results may not be well taken care of. However, when converting this service into a query service for a warehouse, the runtime environment requires higher security, and thus authorization and authentication components are more critical for the system than the throughput. The service user may also require more reliable and accurate results on a queried product than they do for a library book. A better service adaptation can be achieved if contexts can be associated with the old service to the new environment. This association infers two major actions when applying context-awareness on the process of web service adaptation: Context Mapping and Context Adaptation.

Context mapping checks the contexts of the services before and after the adaptation. The goal of the mapping is to judge whether service contexts before adaptation are compatible with the required contexts after adaptation. This check can be done by running a pre-defined test suite on current services before adaptation. The test suite may be composed of multiple test cases that inspect the runtime environment of current web services. Each test case checks the compatibility of certain aspects that are required by the adaptation model. The inspection results are mapped to the requirement of target service contexts, and the compatibility for each aspect is decided.

Based on the compatibility results, the service context adaptation can be performed to adjust the incompatible aspects of current context.

Service context adaptation is performed to adjust the current context to fulfill the requirements of target service. The incompatible contexts can be put into two categories: adjustable contexts and in-adjustable contexts. Adjustable context refers to those incompatible, but controllable contexts in the view of service adapter. For example, the higher throughput can be achieved by adding more service instance in the service contracts when adapting the query service. However, not all the service context aspects are adjustable. Taking the above example again, authorization and authentication module may not be available in the previous service and cannot be provided by the service adapter in the adaptation process. It is up to the adaptation model to decide whether to give up the current adaptation when incompatible and un-adjustable context are found. Incorporating the context information into the service adaptation may also be of great help when choosing the best service to adapt from multiple candidates.

The idea of context-aware web services adaptation is described briefly here. It is believed that such mechanism can be promising to help achieving a dynamic and automatic adaptation and composition process for web services. However, the more detailed design for this mechanism is still under study.

5.3 Substitution-based Service Adaptation

Given a component service A in a composite workflow in Figure 5-3, when A becomes invalid, the ideal solution is not to fix A directly, but to find a functionally similar service A’ to replace A to maintain the original composite workflow. This is the main idea for traditional web service substitution. Here, “functionally similar” means the functionality of service A’ is equivalent to or a superset of functionality of service A,

which we refer to as the Equivalent and Strong Similar relationships, respectively. However, it is possible that such a service A' might not exist in a web service community. In such a case, a traditional solution would turn to service adaptation. Our work adds an intermediate step before substitution and adaptation, called substitution-based adaptation. This is practical when some web services, like service B and C, exist in a service community, whose component functionality is a subset of service A (Weak Similar). In Figure 5-3, the functionality of service A is formed by the integration of service X and Y. Service B contains X and service C contains Y. Thus, if service B and service C can be virtually decomposed into sub-services X, M, Y and N, then a virtual service A'' can be generated by virtually recompose X and Y.

To accomplish this goal, it is necessary to merge and split web services. Generally, it is hard to decompose a web service into sub ones since the implementation details are not available to application integrators. However, since service providers provide the service descriptions when publishing their services into service registries, developers can manipulate these descriptions to analyze service functionalities. In our system, web services in OWL-S description are translated into automata representation. In Chapter 2, the composition operation of automata is introduced. For the decomposition of automata, we adopt a manually approach.

The complexity for automata decomposition is much higher than that of composition [79, 80, 81]. Not all the automata can be decomposed into simple ones automatically. Composite web services often have a simpler automata structure for decomposition. In our implementation, we resort to manual decomposition whenever necessary.

To manually decompose an automaton, the key issue is to find the decomposition position. For example, given an automaton with language a1a2…an, it is possible to decompose at any position j, $j \in [1, n-1]$. However, in most cases, only one of the decomposition satisfies the requirement in the adaptation process. Thus, to avoid unnecessary blind decomposition, a method to determine the value for position j is proposed in our implementation. To illustrate our methods, we still use the example in Figure 5-3. Suppose the regular languages for automata of web service A, B and C are LA= {abcdef}, LB = {abcmn}, LC = {stdef}, respectively. According to the Boyer-Moore Fast String Searching Algorithm, it is easily to find that {abc} and {def} are sub-string of LA. Thus, the starting and ending position of {abc} and {def} are the decomposition position. Then a manual decomposition is conducted to generate the virtual services X and Y, which are then composed together to generate A".

An adaptation process should take place in a timely manner and maintain a consistent state. To guarantee this requirement, one precondition is that one service can be replaced or modified only when it has no running instances. Several adaptation machines are developed for this consistency issue. One example of existing solution is the CoBRA framework [82], which involves an adaptation manager to record the state information and a set of service proxies to guarantee the system consistency. Our work is very similar to CoBRA by keeping the service proxies and State Cache. However, we integrate the concept of service community so the extended adaptation process is further divided into five stages as shown in Figure 5-4.

The existence of a service proxy hides the details of communicating services to each other. Thus, the caller service is not aware of the called service. In Figure 5-4,

Service B sends out a request to the service proxy. This request is further redirected to service A for real execution. When an adaptation trigger initiates the adaptation process, the running instance of service A sends a “stop service” signal to its service proxy, which will forward the signal to Service B for further references. Then service A is ready to enter the substitution-based adaptation process.

The Service Ranking phase is in charge of two tasks: 1) at the beginning of this phrase, a signal is sent to the service proxy to inform the temporary unavailability of service A, so that the Adaptation Monitor can halt the execution of service A after its current running instance completes. At the same time, service proxy informs service B to stop sending service request to it, so service B blocks its execution; 2) the web service community which contains service A is informed to rank all the services that are Equivalent or Strong Similar to A. In the Service Selection phase, the context information is collected to help select the previous ranked services. In the Halt Execution stage, the running instance of service A has completed all the tasks. Thus, it sends a request to the State Cache to store its current state and environment context. This is a key step in the adaptation process, since consistency is guaranteed via 1) restore from the State Cache if the substitution process fails; 2) the candidate service A’ can start regular execution with its initial state set to the one that is stored in the State Cache. In our system, state information mainly includes the input, output, pre-condition and effects (IOPE) of web services. The Service Substitution phase is the duration that the service A is removed from the service proxy and service A’ is linked to the proxy. In this stage, the restore step retrieves all the necessary information from the State Cache and set it to the initial state for service A’. Only when A’ is compatible and accepts this

state, the state information can be removed from the State Cache. Otherwise, service A' should be removed from the proxy indicating the failure of the substitution-based adaptation. In this case, the system switches to the direct adaptation process. When the restore succeeds, the adaptation process enters the final phase, Resume Execution. In this stage, the service proxy is aware of existence of service A', so it notify the pending service B to resume execution again. The new request from service B is forward to service A' while service A' is transparent to service B. With the proposed five phases for substitution-based adaptation, our work provides the consistency for the composite workflow execution. In addition, it reduces the complexity compared with direct service adaptation.

5.4 Related Work

Web service adaptation has been widely studied recently. To find a practical solution for this issue, many solutions have been proposed based on WS-BPEL descriptions. For example, a methodology for the automated generation of service adapters is introduced in [54]. The generated adapters are capable of solving behavioral mismatches among BPEL processes. Their research also deals with the lock situation between two communicating processes. Another BPEL-based solution is the system for runtime adaptation with QoS monitoring in [83]. This system first monitors BPEL processes according to the QoS attributes and then replaces existing partner services based on various pluggable replacement strategies. The chosen replacement services can be syntactically or semantically equivalent to the BPEL interface. Their work is similar to our substitution-based adaptation process. However, they only consider the interface -level equivalence, while our work integrates the behavioral-level similarities. TRAP/BPEL, a general framework which adds automatic behavior into existing BPEL

processes automatically and transparently, is introduced in [84]. The authors define an autonomic BPEL process as a composite web service that is capable of responding to changes in its execution environment. The solution depends on a generic proxy between applications and services, which is a bottleneck in the whole system. In addition, the paper mainly focuses on “adding” service into the BPEL processes, while lacks details on “removing” or “modifying” service in the BPEL processes.

Besides the adaptation mechanisms specific for WS-BPEL, several general solutions are proposed based on component models [82, 85, 86]. For example, CASA, a Contract-based Adaptive Software Architecture, provides a framework for enabling dynamic adaptation of applications, in response to changes in their execution environment [86]. The authors treat each sub-service as a single component in a composite service. Thus, runtime adaptation is fulfilled by re-composition of application components. Though automated adaptation solutions are the purpose for many existing researches, the authors of [53] present a semi-automated support for identifying and resolution of mismatches between service interfaces and protocols, and for generating adapter specification. This is because possible deadlocks can happen due to ordering mismatches between service protocols. A mismatch tree is generated for mismatches that require the developers’ intervention for the resolution of mismatches with deadlocks.

Another support for service adaptation is the dynamic binding of concrete services. It enables runtime binding of service compositions according to some functional and non-functional preferences and constraints. Thus, it can support runtime recovery actions through adaptation via rebinding. The work in [87] is similar to our dynamic binding process. However, there is a general service proxy for each concrete web

service in WS Binder, while we have two types of service proxy: abstract and concrete. Thus, our system not only provides a structured service organization, but also hides more details of the services from the users.

5.5 Application

To illustrate the light-weight adaptation process, a simple travel planning application is implemented. The developing tool is Eclipse with java language under Windows system. This application, represented as a composite web service, aims to provide the travel planning help with given input parameters. First, an atomic component, weather, is consulted to obtain the weather of the source and destination cities. Based on the weather information, transportation service can generate the best choice for the traveler, which includes flight, train and car plans. A hotel booking service is also included. For simplicity, the application uses the TravelStartDate and TravelEndDate as the hotel check-in and check-out date. At last, the SiteRouting service generates a sight-viewing route for the traveler. The UML class diagram for this application is shown in Figure 5-5 (a), which is generated using UML2 plug-ins for Eclipse. The TravelPlanning class is associated with the interface Transportation, which is implemented by three classes: Flight, Train and Car.

Two demonstrations are provided to illustrate the usage of the substitution-based adaptation process, which considers interface-level mismatches and behavior-level mismatches. The interface-level mismatch resolution is shown in Figure 5-5(b). In the original application, the weather service returns the weather in Fahrenheit. Support this service no longer exists or is out-of-service temporality, the application can choose another service which has exactly the same functionality except that the return value is

in Celsius. This kind of data type mismatches can be predefined in adaptation rules. Thus, the application can automatically switch to new service without harm.

The behavior-level mismatch is shown in Figure 5-5(c). We implemented two other composite web services, which are TravelPromotion and DailyAcitivity. The first one is composed of three atomic services, which are HotelPromotion, DiningPromotion and ThemeParkPromotion. Given the city and promotion duration interval, the three services return the prices for hotel, dining and theme park, respectively. Then the user can book corresponding hotel, restaurant and theme parks via TravelPromotion service. DailyActivity Service provides the functionality for arranging daily tourist activities in a specific city. It is straightforward to see that HotelPromotion service is a subset to TravelPlanning service, and a set of consecutive DailyActivity services form a subset functionality of TravelPlanning. Thus, when HotelBooking and SiteRouting services are no longer available, our implementation uses HotelPromotion and DailyAcitivity to substitute them and maintain the correctness of original workflow as shown in Figure 5-6. In the figure, a dashed line means that the original sequential execution is replaced due to substitution, while the solid line reflects the new workflow after the adaptation process.

The goal of this case study is to show the usefulness and practicability of our methodology on the light-weight substitution-based adaptation for issues in Figure 5-3. As seen in Figure 5-6, invalid services, such as SiteRouting, can be replaced by functional equivalent ones via adding certain control flows. No direct adaptation is required to change the published functionality of DailyActivity service. Though a prototype of the entire methodology is not available yet, we believe this light-weight

adaptation can gain the following benefits: 1) it reduces the complexity of traditional web service adaptation process; 2) it increase the flexibility of web service composition through dynamic re-binding; 3) the possibility of exceptions or errors in the adapted workflow is reduced since it prevents the developers modifying the functionality of published services.

5.6 Summary

The rapid growth of service-oriented systems is becoming a reality for today's distributed applications. These applications are developed by assembling reusable software services to create compositions of dynamically bound services. In particular, adaptation is of utmost importance in this process. The general idea for web service adaptation is to solve mismatches which occur at different interoperability levels among services by synthesizing a mediating adaptor. In this chapter, we propose a context-aware substitution-based adaptation approach for dynamic web service composition. We integrate current adaptation solutions with context information to enhance the adaptation framework. In addition, we observe that direct adaptation is sometimes expensive and might be eliminated in some situations. Thus, a light-weight substitution-based adaptation approach is proposed. Currently, a prototype system is under construction to demonstrate the usage of our idea. This system mainly focuses on functional modules related to the substitution-based adaptation process, rather than the traditional adaptation framework. Two finished components are web service community and OWLS2NFA tool. The remaining implementation includes BPEL workflow (re-)binding, adaptation monitoring, abstract/concrete service proxies, etc. Future work also includes further studies on designing and integrating a context model to facilitate the web service adaptation process.

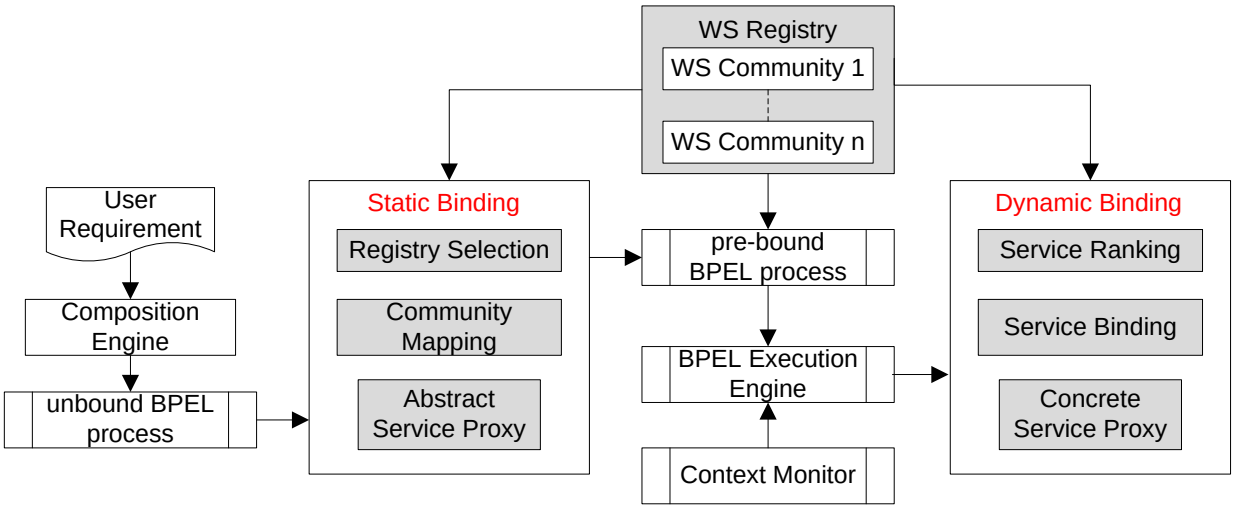

Figure 5-1. Flowchart of Adaptive Composition Process

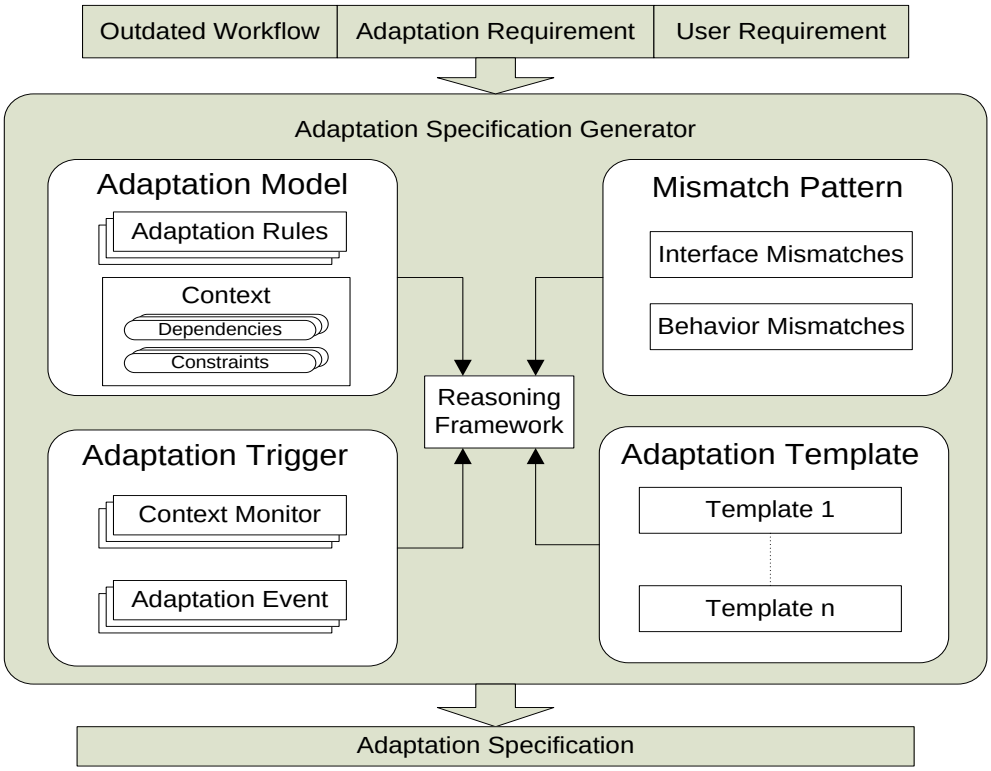

Figure 5-2. Architecture of Web Service Adaptation Framework

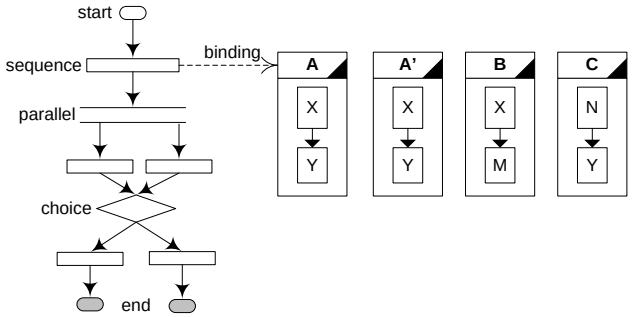

Figure 5-3. Substitution-based Adaptation

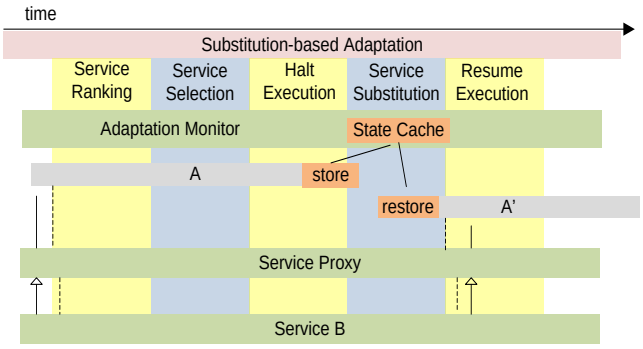

Figure 5-4. Illustration of Substitution-based Service Adaptation Process

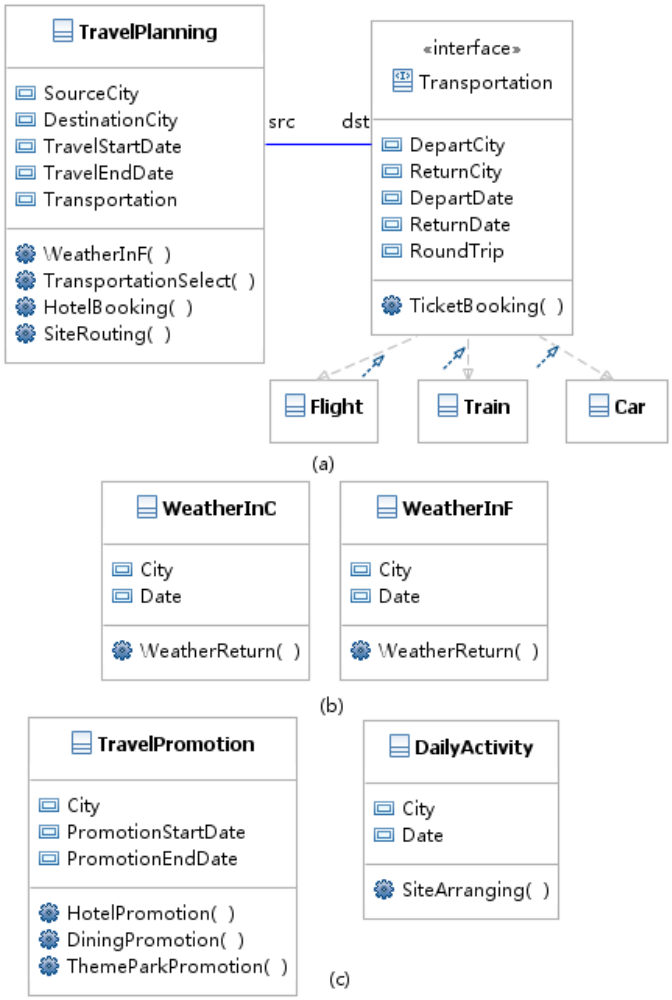

Figure 5-5. TravelPlanning Application

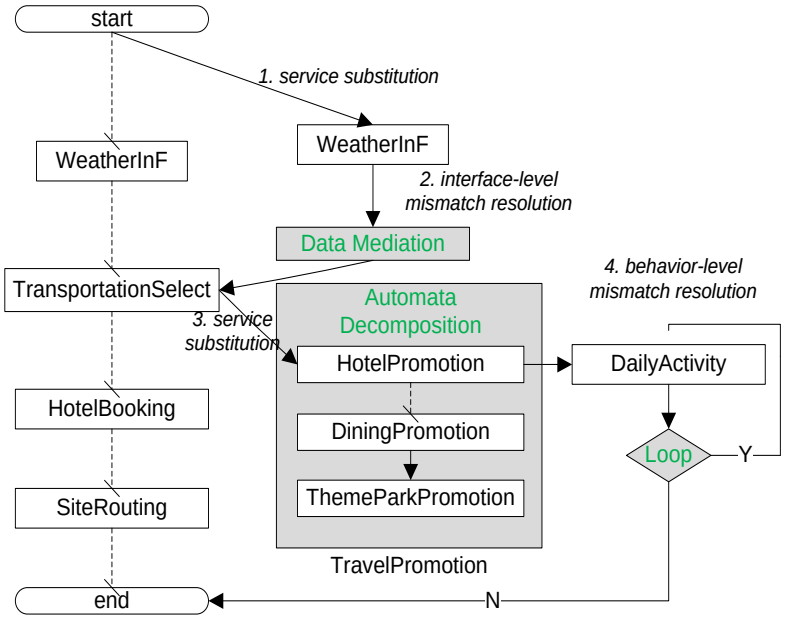

Figure 5-6. TravelPlanning Workflow via Substitution-based Adaptation

CHAPTER 6
EXAMPLE APPLICATION

To show the practicability of our composition framework, the chapter adopts the e-Hospital example as shown in following sections.

6.1 e-Hospital Example

Like the real life hospitals, we assume patient's information, medical history, prescribing, doctors, and billings can be represented as web services. In this application, a patient can request a specific doctor through the QoS requirements in his goal specification. Each doctor maintains a queue of patients. When one doctor becomes unavailable, his queue of patients could be transferred to any doctors at run time. Those patients who specify a particular doctor will either wait for the doctor or be transferred to others after negotiation with the system. Likewise, the patient can make a payment by cash, check or credit card. The entire workflow is shown in Figure 6-1.

To implement the above application, a workflow of participating services will be generated for each patient. Traditionally, this generated workflow is static and is not able to be executed if environments changes. For example, during this workflow generation process, the assignment of a specific or any doctor for a patient is bounded to the workflow. However, if one doctor is not available, the current web service solution without exception handling will have to redo the entire composition step from the very beginning until the final candidate workflow is found. Re-composition of the workflow is a costly process and a new method is necessary to address this problem. In our proposed solution, by integrating substitution and adaptation with composition, rather than regeneration, the e-Hospital problem can achieve high flexibility with only minor

modification of the working composition plan. This proposed solution approach will be shown in Section 6.2 by integrating the substitution and adaptation process.

6.2 Solution

To demonstrate the usage of substitution and adaptation for web service composition, we implement the application using Axis 2 and Eclipse IDE. To implement the web-based simulation for this e-Hospital application, we need to do the following things. First, the real life services are modeled as web services with public interfaces. Second, these web services need to be classified into different web service communities. Third, we need to decide an orchestration between communicating services. Finally, we need to maintain the correctness of the workflow if some services become invalid of unavailable due to changes.

In this particular e-Hospital example, there are roughly three observable web services for the user: e-Patient, e-Doctor and e-Billing. The e-Patient service is an atomic web service, which parses the users' goal to get the machine understandable e-Patient's description. It acts like an agent for the user. The e-Doctor service is an abstract composite service, which is responsible for the simulation of the core logic. It is composed of four sub web services: InfoChecking, DoctorChooser, MediChecking and PrescGenerator. The InfoChecking web service is responsible for checking the e-Patient's information. The DoctorChooser web service is used to choose a specific doctor according to the patient's symptom description and specific requirements about doctors, like a general practitioner or a specialist. After a specific doctor is chosen, the MediChecking service is able to retrieve this doctor's previous knowledge and prescription histories for a certain symptom. Then, the PrescGenerator can produce a new prescription for the e-Patient. It also needs to update his/her prescription histories

and the patient's medical history. Thus, the PrescGenerator service includes two operations: GeneratePrescription and UpdateInformation. The e-Billing service contains three operations: BillingChooser, CalculatePrice and SendInvoice. The BillingChooser operation is used to choose from paying by cash, check or credit card based on users' preferences. The CalculatePrice operation is used to produce the price after an abstract e-Doctor service has generated a prescription. The SendInvoice operation provides a user with a formal certificate of bill processing. The graphical WSDL descriptions for this service are given in Fig 6-2.

After these web services are identified, they are classified into different web service communities. A community of web services gathers web services that address the same users' needs and support their binding through a common interface. As introduced in our previous work, members of a service community are decided by similarities between web services. For example, in this e-Hospital application, we require that all the e-Doctors with the same specialty be put together in the same sub community. Figure 6-3 shows a simple structure of e-Doctor community with only three specialties.

Other web service communities are defined in a way similar to the e-Doctor web service community. Then an orchestration for those communicating web services is proposed using the graphical BPEL diagrams as shown in Figure 6-4. We can see that the dependencies of web services in the executable workflow are straightforward.

Finally, we show that the correctness of the composite workflow can be maintained via synchronous workflow exception handling and asynchronous web service interrupt approaches. When a patient comes to see a doctor, the e-Patient

service parses his descriptions as a required functionality for the e-Hospital system. The system generates an abstract workflow with the knowledge of the e-Doctor community. An abstract community description shows the general professions of e-Doctors in the same department, however, each e-Doctor has different specializations described in physical WSDLs. Thus, mapping abstract descriptions to physical WSDLs may cause synchronous workflow exceptions either due to functionality conflicts or due to communication contradictions. The static substitution chooses similar e-Doctors in the community while the mapping time adaptation generates a virtual nurse to act as a message translator and router. Before the physical plan is executed, all the context constraints are checked when binding physical WSDLs to concrete e-Doctors. In this process, possible asynchronous web service interrupts are solved. For example, a specific D-doctor service community has only two e-Doctors: Dr. Smith and Dr. Taylor. When Dr. Smith has a queue of more than 10 patients, he can't accept any more patients even if he is a suitable service, otherwise he can't finish office hours on time. This contradiction between patient and office hour context is solved by dynamic substitution if Dr. Taylor, with a queue of only 2 patients, accepts Dr. Smith's patients if he is a nominator of Dr. Smith. This process is shown as dashed line a in Figure 6-5. During execution time of Dr. Taylor service, if he has an urgent conference and has to stop servicing, other e-doctors from different communities could be wrapped since Dr. Smith from the same community is also not available. The possible solution is to use binding time adaptation to "wrap" Dr. White as a candidate service if she is a colleague of Dr. Taylor. The dotted line b in Figure 6-5 describes this solution.

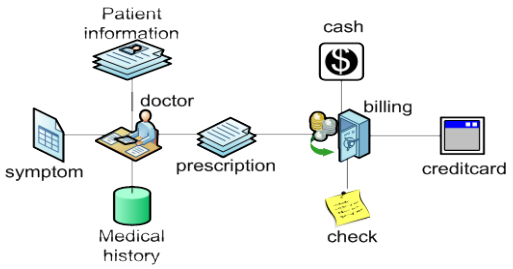

Figure 6-1. Organization of e-Hospital application

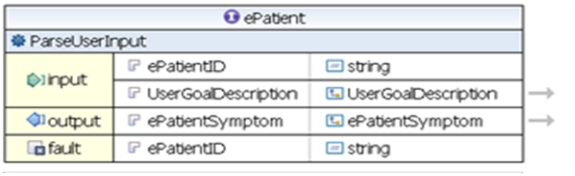

Figure 6-2. WSDL for e-Hospital Web Service

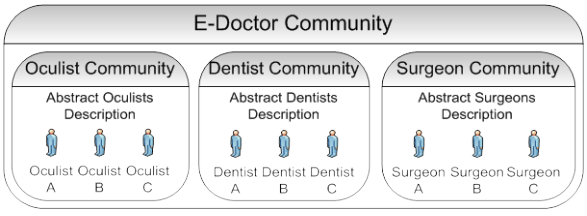

Figure 6-3. e-Doctor Service Community Structure

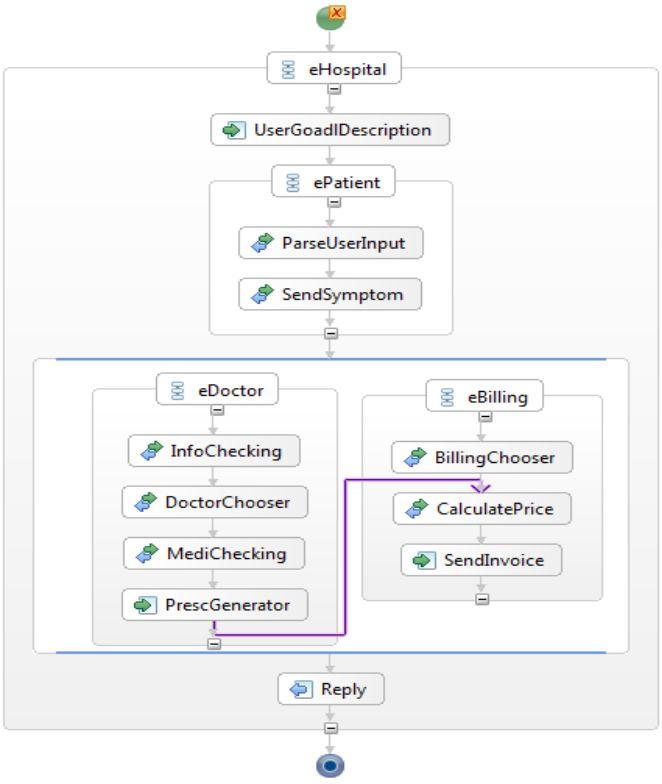

Figure 6-4. WS-BPEL for e-Hospital Application

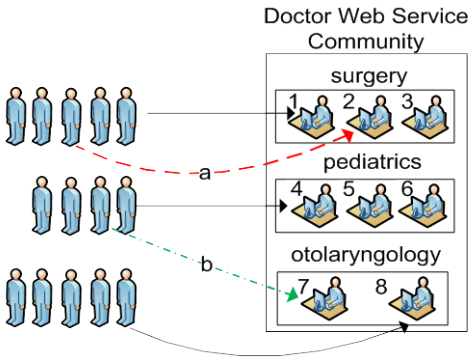

Figure 6-5. Exception Handling for e-Hospital Application

LIST OF REFERENCES

1. Z.Maanar, D.Benslimane, and N.C.Narendra, “What Can Context Do For Web Services?,” *Communications of the ACM*, vol. 49, no.12, pp.98-103, Dec. 2006.

2. B. Soukkarieh and F. Sedes, “Integrating a Context Model in Web Services,” *IEEE International Conference on Web Services*, pp.1195-1196, Jul. 2007.

3. OWL-S API, http://www.mindswap.org/2004/owl-s/api/

4. J. Pathak, S.Basu, and V. Honavar, “Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements,” *Intl. Conference on Service Oriented Computing*, pp. 314-326, Dec. 2006.

5. G. Chafle, K. Dasgupta, A. Kuma, S. Mittal and B. Srivastava, “Adaptation in Web Service Composition and Execution,” *4th IEEE Intl. Conference on Web Services*, pp. 549-557, Sept. 2006.

6. WSDL, http://www.w3.org/TR/wsdl

7. SWRL, http://www.w3.org/Submission/SWRL/

8. WSMO, http://www.wsmo.org/TR/d2/v1.3/

9. OWL-S, http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

10. X. Dong, A. Halevy, J. Madhavan, E. Nemes and J. Zhang, “Similarity Search for Web Services,“ *Very Large Data Bases Conference*, pp. 372-383, Aug. 2004.

11. E. Stroulia and Y. Wang, “Structural and Semantic Matching for Assessing Web-Service Similarity,“ *Intl. Journal of Cooperative Information Systems*, vol. 14, no. 4, pp. 407-437, 2005.

12. J. Hau, W. Lee and J. Darlington, “A Semantic Similarity Measure for Semantic Web Services,“ *Web Service Semantics Workshop at WWW*, 2005.

13. X. Wang, T. Vitar, M. Hauswirth and D. Foxvog, “Building Application Ontologies from Descriptions of Semantic Web Services,“ *IEEE/WIC/ACM International Conference on Web Intelligence*, pp.337-343, Nov. 2007.

14. D. Grigori, J. C. Corrales, and M. Bouzeghoub, “Behavioral matchmaking for service retrieval,“ *4th IEEE International Conference on Web Services*, pp.145-152, Sept. 2006.

15. A. Wombacher, P. Frankhauser and E. Neuhold, “Transforming BPEL into annotated deterministic finite state automata for service discovery,” *2nd IEEE International Conference on Web Services*, pp. 316-323, Jul. 2004.

16. A. Wombacher, P. Fankhauser, B. Mahleko and E. Neuhold, “Matchmaking for Business Processes based on Choreographies,” *IEEE International Conference on e-Technology, e-Commerce and e-Service*, pp.359- 368, Apr. 2004.

17. Z. Shen and J. Su, “Web Service Discovery Based on Behavior Signatures,” *Intl. Conference on Services Computing*, pp. 279- 286, Jul. 2005.

18. L. Lei and Z. Duan, “Transforming OWL-S Process Model into EDFA for Service Discovery,” *4th IEEE International Conference on Web Services*, pp. 137-144, Sept. 2006.

19. Enterprise UDDI Services in Windows Server 2003, http://www.microsoft.com/windowsserver2003/technologies/idm/uddi/default.mspx

20. jUDDI, http://ws.apache.org/juddi/

21. IBM WebSphere Application Server, http://www-01.ibm.com/software/webservers/appserv/was/

22. HP SOA Systinet Business Service Registry, https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-130-27^1461_4000_100__

23. M. Eid, A. Alamri and A. EI. Saddik, “A reference model for dynamic web service composition systems,” *Intl. Journal of Web and Grid Services*, vol. 4, no.2, pp. 149 –168, Jun. 2008.

24. P. W. Chan and M. R. Lyu, “Dynamic Web Service Composition: A New Approach in Building Relialbe Web Service,” *22nd International Conference on Advanced Information Networking and Applications*, pp. 20-25, Mar. 2008.

25. D. VanderMeer, A. Datta and K. Dutta, “FUSION: A System Allowing Dynamic Web Service Composition and Automatic Execution,” *IEEE Intl. Conference on E-Commerce*, pp.24-27, Jun. 2003.

26. L. Chen and R. Chow, “A Web Service Similarity Refinement Framework using Automata Comparison”, *Proceedings of 4th Intl. Conference on on Information Systems, Technology and Management*, pp. 44-55, Mar. 2010.

27. E. D. Valle and D. Cerizza, “Cocoon Glue: A Prototype of WSMO Discovery Engine for the Healthcare Field,” *Proceedings of 2nd WSMO Implementation Workshop*, pp.1-12, Jun. 2005.

28. B. Benatallah, M. Hacid, A. Leger, C. Rey and F. Toumani, “On automating Web Services discovery,” The VLDB Journal, vol. 14, no. 1, pp. 84-96, Mar. 2005.

29. C. Atkinson, P. Bostan, O. Hummel and D. Stroll, “A Practical Approach to Web Service Discovery and Retrieval,” IEEE *Intl. Conference on Web Services*, pp. 241-248, Jul. 2007.

30. T. S. Mahmood, G. Shah, R. Akkiraju, A. Ivan and R. Goodwin, “Searching Service Repositories by Combining Semantic and Ontological Matching,” *IEEE Intl. Conference on Web Services*, pp.13-20, Jul. 2005.

31. U. Thaden, W. Siberski and W. Nejdl, “A Semantic Web based Peer-to-Peer Service Registry Network,” Technical Report, Learning Lab Lower Saxony, 2003.

32. K. Sivashanmugam, K. Verma and A. Sheth, “Discovery of Web Services in a Federated Registry Environment,” *IEEE Intl. Conference on Web Services*, pp.270-278, Jul. 2004.

33. Z. Chen, L. Chia, B. Silverajan and B. Lee, “UX-An Architecture Providing QoS-Aware and Federated Support for UDDI,” *IEEE Intl. Conference on Web Services*, pp. 171-176, Jun. 2003.

34. A.S.Bilgin and M. P. Singh, “A DAML-Based Repository for QoS-Aware Semantic Web Service Selection,” *IEEE Intl. Conference on Web Services*, pp. 368-375, Jul. 2004.

35. Y. Yamato and H. Sunaga, “Context-Aware Service Composition and Component Change-over using Semantic Web Techniques,” *IEEE Intl. Conference on Web Services*, pp. 687-694, Jul. 2007.

36. M. Keidl and A. Kemper, “Towards Context-Aware Adaptable Web Services,” *13th International Conference on World Wide Web*, pp. 55-65, May 2004.

37. Y. Yamato, H. Ohnishi and H. Sunaga, “Study of Service Processing Agent for Context-Aware Service Coordination,” *IEEE Intl Conference on Services Computing*, pp. 275-288, Jul. 2008.

38. Z. Maamar, S. Mostefaoui and H. Yahyaoui, “Toward an Agent-Based and Context-Oriended Approach for Web Service Composition,” *IEEE Trans. On Knowledge and Data Engine*, vol. 17, no. 5, pp. 686-697, May 2005.

39. G. Zheng and A.Bouguettaya, “A Web Service Mining Framework,” *IEEE Intl. Conference on Web Services*, pp.1096-1103, Jul. 2007.

40. W.Gaaloul, K.Baina and C.Godart, “Log-based mining techniques applied to Web service composition reengineering,” *Service Oriented Computing and Applications*, vol. 2, no. 2-3, pp. 93-110, 2008.

41. Y. Taher, D. Benslimane, M.C. Fauvet and Z. Maamar, “Towards an Approach for Web services Substitution,” *10th Intl. Database Engineering and Applications Symposium*, pp. 166-173, Dec. 2006.

42. Y. Li and H. V. Jagadish, “Compatibility Determination in Web Services,” *ICEC E-Government and E-Services Workshop*, 2003.

43. R. Hamadi and B. Benatallah, “A Petri Net-based Model for Web Service Composition,” *Proceedings of the 14th Australasian Database Conference*, pp. 191-200, Feb. 2003.

44. J. Pathak, S. Basu, R.R. Lutz, V. Honavar, “Selecting and Composing Web Services through Iterative Reformation of Functional Specifications,” *18th IEEE Intl. Conference on Tools with Artificial Intelligence*, pp. 445-454, Jun. 2006.

45. J. Pathak, S. Basu, V. Honavar, “On Context-Specific Substitutability of Web Services,” *5th IEEE Intl. Conference on Web Services*, pp. 192-199, Jul. 2007.

46. U. Kuter, E. Sirin, B. Parsia, D. S. Nau and J. A. Hendler, “Information Gathering During Planning for Web Service Composition,” *Journal of Web Semantics*, Vol 3, pp 183-205, 2005.

47. J. Pathak, S. Basu, R. R. Lutz and V. Honavar, “MoSCoE: A Framework for Modeling Web Service Composition and Execution,” *Intl. Conference on Data Engineering*, pp. 143, Apr. 2006.

48. C. C. Chiang, “Automated software wrapping,” *Proceedings of the 45th annual southeast regional conference*, pp. 59-64, Mar. 2007.

49. L. Bordeaux, G. Salaun, D. Berardi and M. Mecella, “When are Two Web Services Compatible?,” *Proceedings of the 5th Intl. Workshop on Technologies for E-Services* (TES), pp. 15-28, Aug. 2004.

50. B. Benatallah, F. Casati and F. Toumani, “Representing, analyzing and managing Web service protocols,” *Data & Knowledge Engineering*, vol. 58, no. 3, pp. 327-357, 2006.

51. B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad and F. Toumani, “Developing Adapters for Web Services Integration,” *International Conference on Advanced Information Systems Engineering*, pp. 415-429, Jun. 2005.

52. W. Kongdenfha, R. Saint-Paul, B. Benatallah and F. Casati, “An Aspect-Oriented Framework for Service Adaptation,” *Intl. Conference on Service Oriented Computing*, pp. 15-26, Dec. 2006.

53. H.R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera and F. Casati, “Semi-Automated Adaptation of Service Interactions,” *16th World Wide Web Conference*, pp. 993-1002, May 2007.

54. A. Brogi and R. Popescu, “Automated Generation of BPEL Adapters,” *Intl. Conference on Service Oriented Computing*, pp. 27-39, Dec. 2006.

55. D. Skogan, R. Gronmo, I. Solheim, “Web Service Composition in UML,” *8th Intl. Enterprise Distributed Object Computing Conference*, pp. 47-57, Sept. 2004.

56. Y. Yamato and H. Sunaga, “Context-Aware Service Composition and Component Change-over using Semantic Web Techniques,” *IEEE Intl. Conference on Web Services*, pp. 687-694, Jul. 2007.

57. Z. Zhou, S. Bhiri, W. Gaaloul and M. Hauswirth, “Developing Process Mediator for Supporting Mediated Web Service Interactions,” *Proceeding of the 6th European Conference on Web Services*, pp. 155-164, Nov. 2008.

58. D. Konig, N. Lohmann and S. Moser, “Extending the Compatibility Notion for Abstract WS-BPEL Processes,” *Proceeding of the 17th Int. conference on World Wide Web*, pp. 785-794, Apr. 2008.

59. K. Belhajjame, “Addressing the Issue of Service Volatility in Scientific Workflows,” *Proceeding of the 5th Intl. Conference on Service-Oriented Computing*, pp. 377-382, Sept. 2007.

60. W. Kongdenfha, H.R.M. Nezhad, B. Benatallah, F. Casati and R. Saint-Paul, “Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adaptors,” *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 94-107, Apr .2009.

61. X. Li, Y. Fan and F. Jiang, “A classification of Service Composition Mismatches to Support Service Mediation,” *Proceedings of the 6th Intl. Conference on Grid and Cooperative Computing*, pp. 315-321, Aug. 2007.

62. E. Cimpian and A. Mocan, “WSMX process mediation based on choreographies,” *Proceedings of the Workshop on Web Service Choreography and Orchestration for Business Process Management at the BPM*, pp. 130-143, Aug. 2006.

63. E. Rahm and P. A. Bernstein, “A survey of approaches to automatic schema matching,” *Journal of the Very Large Databases*, vol.10, no. 4, pp. 334-350, Dec. 2001.

64. S. K. Williams, S. A. Battle and J. E. Cuadrado, “Protocol mediation for adaptation in semantic web services,” Technical report, HP Technical Report. HPL-2005-78, 2005.

65. R. Fanner, A. Raybone, R. Uddin, M. Odetayo and K. M. Chao, “Mediation architecture for integration of heterogeneous discipline focused workflow languages,” *Proceedings of IEEE Intl. Workshop on Service-Oriented Applications, Integration and Collaboration*, pp. 507-510, Oct. 2007.

66. R. Vaculin and K. Sycara, “Towards automatic mediation of OWL-S process models,” *Proceeding of IEEE Intl. Conference on Web Services*, pp. 1032-1039, Jul. 2007.

67. J. Zdravkovic, “Process Integration for the Extended Enterprise,” PhD thesis, Royal Institute of Technology, 2006.

68. B. Lin, N. Gu and Q. Li, “A requester-based mediation framework for dynamic invocation of web services,” *Proceeding of the 4th Intl. Conference on Service-Oriented Computing*, pp. 445-454, Dec. 2006.

69. A. Bucchiarone, R. Kazhamiakin, C. Cappiello, E. Nitto and V. Mazza, “A Context-driven Adaptation Process for Service-based Applications,” *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, pp. 50-56, Jun. 2010.

70. C. Dorn, D. Schall and S. Dustdar, “Context-aware Adaptive Service Mashups,” IEEE Asia-Pacific Services Computing Conference, pp. 301-306, Dec. 2009.

71. F. Daniel and M. Matera, “Mashing Up Context-Aware Web Applications: A Component-Based Development Approach,” *Proceedings of the 9th international conference on Web Information Systems Engineering*, pp. 250-263, Sept. 2008.

72. L. Chen and R. Chow, “Building A Valid Web Service Composition Using Integrated Service Substitution and Adaptation,” *Intl. Journal of Information and Decision Sciences*, vol. 2, no. 2, pp. 113-131, 2010.

73. L. Chen, Yan Li and R. Chow, “Enhancing Web Service Registries with Semantics and Context Information,” *Proceedings of 7th Intl. Conference on Services Computing*, pp.641-644, Jul. 2010.

74. F. Fleurey, V. Dehlen, N. Bencomo, B. Morin and J.M. Jezequel, “Modeling and Validating Dynamic Adaptation,” *Proceedings of the Workshops and Symposia at MODELS*, pp. 97-108, Sept. 2008.

75. A. Staikopoulos, O. Cliffe, R. Popescu, J. Padget and S. Clarke, “Template-Based Adaptation of Semantic Web Services with Model-Driven Engineering,” *IEEE Transactions on Services Computing*, vol. 3, no. 2, pp. 118-130, 2010.

76. K. Wang, M. Dumas and C. Qouyang, “The Service Adaptation Machine,“ Proceedings of the 6th IEEE European Conference on Web Services, pp. 145-154, Nov. 2008.

77. S. C. Previtali, “Dynamic Updates: Another Middleware Service?,” *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007*, pp. 49-54, Mar. 2007.

78. F. Irmert, M. Meyerhofer and M. Weiten, “Towards Runtime Adaptation in SOA Environment,” *4th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, co-located at the 21th European Conference on Object-Oriented Programming*, pp. 17-26, Jul. 2007.

79. S. Baranov and L.Bregman, “Automata decomposition and synthesis with PLAM,” *9th EUROMICRO symposium on micro-processing and microprogramming on Open system design: hardware, software and applications*, pp. 759-766, 1993.

80. Y. Hirakawa and K. Okada, “An automaton decomposition method for program structure simplification,” *Systems and Computers in Japan*, vol.16, no.5, pp. 21-31, 1985.

81. R. P. Tucci, “A note on the decomposition of infinite automata,” *International Journal of Computer Mathematics*, vol.24, no.2, pp.141-149, 1988.

82. F. Irmert, T. Fischer and K. M. Wegener, “Runtime Adaptation in a Service-Oriented Component Model, *Adaptive and Self-Managing Systems workshop*, pp. 97-104, May 2008.

83. O. Moser, F. Rosenberg and S. Dustdar, “Non-Intrusive Monitoring and Service Adaptation for WS-BPEL,” *Proceeding of the 17th Intl. Conference on World Wide Web*, pp. 815-824, Aug. 2008.

84. O. Ezenwoye and S. M. Sadjadi, “TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services,” Technical Report FIU-SCIS-2006-06-02, Florida International University, 2006.

85. R. Mateescu, P. Poizat and G. Salaun, “Behavioral Adaptation of Component Compositions based on Process Algebra Encodings,” *22nd IEEE/ACM Intl. Conference on Automated Software Engineering*, pp. 385-388, Nov. 2007.

86. A. Mukhija and M. Glinz, “Runtime Adaptation of Applications through Dynamic Recomposition of Components,” *Proceedings of the 18th Intl. Conference on Architecture of Computing Systems*, pp. 124-138, Mar. 2005.

87. M. D. Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo and E. D. Nitto, “WS Binder: a Framework to enable Dynamic Binding of Composite Web Services,” Proceedings of the Intl. Workshop on Service Oriented Software Engineering, in conjunction with the 28th ICSE, pp. 74-80, May 2006.

BIOGRAPHICAL SKETCH

Lu Chen received her Ph.D. on computer science in University of Florida in May 2011. She was working under the supervision of Dr. Randy Chow. Her current research area is on web service composition. Sub topics include web service similarity comparison, web service community, web service substitution and adaptation.

Lu Chen received her B.S.and M.S. in Huazhong University of Science and Technology in 2003 and 2006, respectively.