AN APPROACH FOR IDENTIFYING SERVICE COMPOSITION PATTERNS FROM EXECUTION LOGS

by

Ran Tang

A thesis submitted to the Department of Electrical and Computer Engineering
In conformity with the requirements for
the degree of Master of Applied Science

Queen’s University
Kingston, Ontario, Canada
(September, 2010)
Copyright ©Ran Tang, 2010
Abstract

Different types of web resources, such as Web Services, HTTP-based APIs and websites, can be located on the web to provide various services, such as information access and online banking. Such services are the basic building blocks to compose more complex functionality that cannot be achieved by a single service. Many service-oriented applications can be composed to fulfill similar functional requirements. Among various applications, a set of services can be frequently used together to deliver a unique functionality. Such set of services are called a service composition pattern. More specifically, a service composition pattern consists of a set of services and the control flow among the services. The reuse of the service composition patterns can facilitate the composition of new applications, improve existing applications and optimize maintenance process of services. To facilitate the identification and reuse of service composition patterns, we propose an approach that mines the service composition patterns from execution logs produced by service-oriented applications during runtime. Since the execution logs can be produced by heterogeneous web resources, we propose a unified description schema to describe various web resources in order to identify functionally similar services of different types. This helps reveal complete service composition patterns. Then we identify frequently associated services using Apriori algorithm and heuristics. Finally, we recover the control flow among the services using the event graph and process mining techniques. The effectiveness of the approach is evaluated through two case studies. The result shows that the unified description schema facilitates the identification of similar services of different types and our approach can effectively identify service composition patterns.
Acknowledgements

I gratefully thank my supervisor Dr. Ying Zou, not only for her guidance and supervision for this thesis research, but also for her exceptional support and help in every aspect throughout my study at Queen’s.

I am very thankful to Dr. Ahmed Hassan, Dr. Bram Adams, Ms. Joanna Ng, Mr. Leho Nigul and Ms. Janet Wong for their valuable advice and feedback on my research work.

I would like to thank the members of the Software Reengineering Research Group: Mr. Xulin Zhao, Mr. Hua Xiao, Ms. Jin Guo, Mr. Brain Chan, Mr. Lionel Marks, Mr. Derek Foo, Ms. Liliane Barbour, Mr. Mohamed Al Guindy, Mr. Andrew Carter, Mr. Chris Fiorentino, Mr. Hao Yuan, Ms. Gehan Kamel and Mr. Bipin Upadhyaya, for their companionship that has made my master study a colorful experience.

I would like to express my gratitude to Dr. Ahmad Afsahi, Dr. Carlos Saavedra and Dr. Mohammad Zulkernine for taking the time to sit on my thesis examination committee.

Finally, I am grateful to my family and friends for their earnest care and encouragement.
# Table of Contents

Abstract ............................................................................................................................................ ii
Acknowledgements ......................................................................................................................... iii
Chapter 1 Introduction ..................................................................................................................... 1
  1.1 Background ............................................................................................................................ 1
    1.1.1 Web Resources ................................................................................................................ 1
    1.1.2 Service Composition ....................................................................................................... 2
    1.1.3 Execution and Monitoring of Service-Oriented Applications ........................................ 3
  1.2 Motivation and Problem Definition ....................................................................................... 5
  1.3 Challenges .............................................................................................................................. 7
  1.4 Organization of Thesis ......................................................................................................... 10
Chapter 2 Related Work ................................................................................................................. 11
  2.1 Service Oriented Architecture .............................................................................................. 11
    2.1.1 Service Discovery ......................................................................................................... 11
    2.1.2 Service Composition ..................................................................................................... 12
    2.1.3 SOA Monitoring ........................................................................................................... 14
    2.1.4 Service Composition Patterns Identification ................................................................. 15
  2.2 Resource Description ........................................................................................................... 17
    2.2.1 Schemas of Resource Description ................................................................................. 17
    2.2.2 Techniques for Generating Resource Description Records .......................................... 20
  2.3 Information Extraction ......................................................................................................... 22
    2.3.1 Information Extraction Systems .................................................................................... 23
  2.4 Summary .............................................................................................................................. 25
Chapter 3 A Unified Schema for Describing Heterogeneous Web Resources .............................. 26
  3.1 Overview of Web Resources ............................................................................................... 26
  3.2 A Unified Schema for Describing Web Resources .............................................................. 32
  3.3 Techniques for Describing Web Resources Using the Proposed Schema .......................... 35
    3.3.1 Parsing Description Files .............................................................................................. 35
    3.3.2 Identifying Operations of Web Resources ................................................................. 36
    3.3.3 Generating Descriptive Tags for Operations ................................................................. 42
    3.3.4 Constructing Formal Interface for Operations ............................................................ 44
List of Figures

Figure 3-1 Example of an informational web page ................................................................. 27
Figure 3-2 Example of a web page that contains a web form .................................................. 28
Figure 3-3 Example of a WSDL file for a SOAP-based Web Service ...................................... 29
Figure 3-4 Example of an HTTP-based API represented using single-page description .......... 31
Figure 3-5 The unified description schema ............................................................................. 32
Figure 3-6 Example of HTML fragments of HTTP-based API operations ................................. 36
Figure 3-7 Tree structure of the HTML fragments .................................................................. 37
Figure 4-1 Overview of the approach to mine service composition patterns .............................. 48
Figure 4-2 Example of identifying pattern service set ................................................................ 51
Figure 4-3 Alternative control flow structure in service-oriented application ............................ 54
Figure 4-4 The execution instances with timeline of service invocation events .......................... 56
Figure 4-5 Identifying precedence and parallelism relations using event graph ....................... 57
Figure 4-6 Example of an overall execution flow ..................................................................... 61
Figure 4-7 The identified service composition pattern with control flow structures .................. 63
List of Tables

Table 4-1 Example of execution instances ................................................................. 50
Table 4-2 Identified execution flow for each execution instance ......... 60
Table 5-1 Summary of the set of operations .............................................................. 66
Table 5-2 Result of identifying operations for HTTP-based APIs from web pages .......... 68
Table 5-3 Result of identifying pairs of similar operations ....................... 69
Table 5-4 The service-oriented applications used in the case study .................... 70
Table 5-5 Result of identifying service composition patterns ....................... 72
Table 5-6 Summary of the identified service composition patterns .................... 74
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajax</td>
<td>Asynchronous JavaScript and XML</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CEI</td>
<td>Common Event Infrastructure</td>
</tr>
<tr>
<td>DAML-S</td>
<td>DARPA Agent Markup Language for Services</td>
</tr>
<tr>
<td>DDC</td>
<td>Dewey Decimal Classification</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IAFA</td>
<td>Internet Anonymous FTP Archives</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LSDIS</td>
<td>Large Scale Distributed Information Systems</td>
</tr>
<tr>
<td>MARC</td>
<td>MAchine-Readable Cataloging</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural language processing</td>
</tr>
<tr>
<td>OWL-S</td>
<td>Ontology Web Language For Services</td>
</tr>
<tr>
<td>PICS</td>
<td>Platform for Internet Content Selection</td>
</tr>
<tr>
<td>PHP</td>
<td>HyperText Preprocessor</td>
</tr>
<tr>
<td>POS tagging</td>
<td>Part-Of-Speech tagging</td>
</tr>
<tr>
<td>PPM</td>
<td>Polymorphic Process Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>REST</td>
<td>REpresentational State Transfer</td>
</tr>
<tr>
<td>SA-REST</td>
<td>Semantic Annotations for REST</td>
</tr>
<tr>
<td>SA-WSDL</td>
<td>Semantic Annotations for WSDL</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TSIMMIS</td>
<td>The Stanford-IBM Manager of Multiple Information Sources</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery and Integration</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>USDL</td>
<td>Universal Service Description Language</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WADL</td>
<td>Web Application Description Language</td>
</tr>
<tr>
<td>WIEN</td>
<td>Wrapper Induction ENviroment</td>
</tr>
<tr>
<td>WSBPEL</td>
<td>Web Services Business Process Execution Language</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Definition Language</td>
</tr>
<tr>
<td>WSOI</td>
<td>Web Service Offering Infrastructure</td>
</tr>
<tr>
<td>WSOL</td>
<td>Web Service Offering Language</td>
</tr>
<tr>
<td>WSRM</td>
<td>Web Services Reliable Messaging</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Service Oriented Architecture (SOA) is an IT architectural style to support the integration of repeatable tasks (i.e., services). A service-oriented application is composed of multiple services to provide complex functionality that cannot be achieved by a single service. Given the functional and non-functional requirements, the SOA professionals discover desired services and compose them into a service-oriented application using specification languages, such as Web Services Business Process Execution Language (WSBPEL) [15]. In the following sub-sections, we give an overview of the background.

1.1.1 Web Resources

Services are the basic building blocks for service compositions in the paradigm of SOA. A service can be implemented using various types of web resources, such as SOAP-based Web Services and HTTP-based APIs. Web resources can be located and accessed through the Internet to provide various functionalities, such as information access and online banking. The web resources are identified by Universal Resource Identifier (URI). A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource [83]. A Uniform Resource Locator (URL) is a special type of URI which provides a means of locating the resource by describing its network location. Various types of web resources co-exist on the web and are described in heterogeneous formats. For example, Web Service Description Language (WSDL) [18] is used to describe SOAP (Simple Object Access Protocol) [97]-based Web Services that makes remote procedure calls. HTTP-based APIs are simpler web resources, implemented as
a set of standard HTTP requests. Examples of HTTP-based APIs are twitter [106], Flickr [34] and various Yahoo APIs [119]. The HTTP-based APIs can be described using web pages or WSDL 2.0. Informational websites are implemented by various technologies, such as Ajax [38], HTML and XML. Web resources of varied types [55] are discovered to compose service-oriented applications according to the functional and non-functional requirements. A particular functional requirement can be fulfilled by different types of web resources. For example, some service providers offer online billing service as SOAP-based Web Services and others offer similar functionality using HTTP-based APIs. The SOA professionals choose the best matching web resource to compose a service-oriented application.

1.1.2 Service Composition

Service composition is the process to combine available services to meet the requirements for fulfilling an objective. Although the existing services may be designed for the construction of a specific service-oriented application, new unplanned applications can be assembled from available services to fulfill a new requirement. The availability of abundant services inspires new applications.

Service composition can be conducted manually or automatically. The service-oriented applications are traditionally developed using conventional programming languages, such as Java and C#. This traditional approach is flexible to integrate various types of web resources. However, conventional programming languages (e.g., Java and C#) are not designed for the service composition. The SOA developers have to implement a great amount of low-level details, such as data conversions and preparation of SOAP payload. In addition, such low-level details are intermingled with business logic in the code. This makes the application difficult to comprehend and maintain. The service-oriented applications can be composed with the aid of service
composition tooling in order to hide the complexity of the low-level details. The service composition tooling includes three main elements: a composition model and language for specifying the service composition, a development environment, and a runtime execution environment. The process model is an example of composition model widely used in practice. Using the process model, the service composition can be expressed as a sequence of connected tasks using WSBPEL. The development environment provides a graphical user interface to drag and drop service components. The runtime execution environment interprets and executes the composed applications. One example of the service composition tooling is the WebSphere suite provided by IBM. Most service composition tooling is limited to composing SOAP-based Web Services. JOpera for Eclipse is a rapid service composition tooling that offers a visual composition language and a runtime execution platform for building and executing service-oriented applications using reusable services, which are not strictly limited to SOAP-based Web Services.

The automatic service composition can be achieved using a service composition middleware. The service composition middleware receives request from the user and responds to the request by providing a composed service-oriented application to fulfill the request. The request can be described with diverse description languages. For example, the requests can be expressed using natural languages or finite state automata. The middleware generate composition plans based on the availability of services by matching the syntactic and semantic meanings of the interfaces of operations. The most suitable composition plan is chosen to implement as the composed service-oriented application and deliver to the user.

1.1.3 Execution and Monitoring of Service-Oriented Applications
The execution of service-oriented applications can be categorized into two paradigms: the centralized execution paradigm and the distributed execution paradigm [31]. The centralized execution paradigm is similar to the client-server architecture. A central scheduler controls the execution of all the component services. The eFlow [100] platform developed by HP is an example of this execution paradigm. In contrast, no central scheduler exists in the distributed execution paradigm. The hosts of component services have their own coordinators to collaborate with the coordinators of other hosts in order to achieve the correct execution order. The SELF-SERV [92] system is an example of this execution paradigm. The centralized execution paradigm and distributed execution paradigm can be combined to form a hybrid paradigm. In the hybrid paradigm, a coordinator may control a set of component services.

If a service-oriented application is composed with the support of service composition tooling, it is deployed and executed in a runtime execution environment, such as IBM WebSphere Process Server [52]. The runtime execution environment interprets and executes the application described in a composition specification language (e.g., WSBPEL). Such execution environments provide built-in monitoring facility which emits events during the execution of a service-oriented application. By subscribing to the events, we can trace the execution of the application (e.g., the invocation of various services). Traditionally, applications are individually deployed on a proprietary infrastructure owned by the organization which operates and utilizes the application. With the increasing adoption of “Platform as a service” paradigm, which provides a centralized runtime execution environment, more and more service-oriented applications are deployed on centralized runtime execution environments, such as Microsoft Azure Services Platform [69] and Google App Engine [40]. For instance, a platform named Heroku [98] hosts over 45,000 applications in 2007. Compared to the traditional deployment model, the “Platform as a service”
paradigm can reduce the cost and the complexity of managing the underlying hardware and software. The adoption of the “Platform as a service” paradigm facilitates the monitoring of the execution of a large number of services-oriented applications to obtain the execution logs.

1.2 Motivation and Problem Definition

Organizations compose service-oriented applications using various web resources. Different service-oriented applications can be composed to fulfill the similar functional requirements from various organizations. For example, a travel reservation application may contain a service to reserve tickets for sports events in the destination. Another travel reservation application may use a service to search for nearby restaurants. Variations exist in the two different applications. However, most of travel reservation applications deliver some common functionality, such as booking transportation and accommodation. The set of services delivering such common functionality can be identified as a service composition pattern. In general, a service composition pattern consists of a set of services and their control flow. The control flow defines the execution order in which the services should be invoked (e.g., sequential order, parallel order or alternative order). There are three major benefits for identifying and reusing the service composition patterns, namely, facilitating the composition of new applications, improving existing applications and optimizing maintenance process of services.

1) Facilitating the composition of new applications. The services contained in a pattern are frequently executed together by the service-oriented applications in the defined execution order. The service composition patterns reflect the best practices. They are well tested by large amount of adoptions. Therefore, an application potentially has better quality when it is composed using such patterns. In the current practice of service composition [62], the service-oriented applications are composed from scratch without
reuse of existing patterns. Many organizations may have similar requirements and subsequently compose their own service-oriented applications. Substantial amount of personnel and resource can be wasted in composing similar applications. With the mechanism for identifying and reusing existing service composition patterns, the organizations can benefit from the existing patterns and reduce the cost for composing applications. Specifically, the SOA professional can use the existing service composition patterns as the skeleton of their new applications and add/ remove services to suit the particular requirements.

2) **Improving existing applications.** Service composition patterns can improve existing applications. A service-oriented application may be inadequately composed and some part of the application does not reflect the best practice. However, it can be risky to abandon the entire application. In order to improve the quality of the existing application by comparing an existing application against a pattern, the application maintainer can identify the discrepancy and consider applying the pattern in the existing application.

3) **Optimizing maintenance process of services.** The services used in patterns are more heavily used than other services. Thus, the patterns can be used to optimize the allocation of maintenance personnel and resources. Service maintainers can allocate more personals to maintain and improve the services used in a pattern. Since the services in a pattern are frequently used together, the maintainer can make their exchanged data and the interfaces more compatible to avoid the conversion overhead.

In the current practice of service composition, there are limited mechanisms or techniques for identifying and documenting heavily used service composition patterns. This limitation
hinders the reuse of existing service composition patterns to ease and optimize future development process.

A great amount of research effort has been devoted to facilitate the discovery and the reuse of service composition patterns. The research can be divided into top-down and bottom-up approaches. In the top-down approach [24], the business process management architects compare the business processes from different organizations to identify commonalities and document them as patterns. However, this static approach does not consider the frequency of executing the applications when identifying the patterns. In the bottom-up approaches [5] [3] [22] [113] [114] [88], the execution logs of applications are analyzed to mine business processes. However, the existing research focuses on recovering the control flows of a single business process without extracting patterns shared among multiple applications.

To identify service composition patterns frequently used in practice, we extend existing bottom-up approaches to mine service composition patterns from execution logs produced by service-oriented applications. Instead of recovering a single business process from the logs, we identify frequently executed patterns used by multiple service-oriented applications. To facilitate the reuse of a service composition pattern as an independent and ready-to-use component, we further infer the control flow within the pattern.

### 1.3 Challenges

Various challenges need to be overcome to derive an effective mechanism for identifying heavily used service composition patterns used by service-oriented applications. We focus on addressing the following challenges:

1) Various web resources are used as services to compose service-oriented applications.

   Service-oriented applications may use different but functionally similar web resources to
fulfill the same requirements. Identifying similar web resources of different types is essential to discovering complete service composition patterns. For example, assume two applications contain three identical services (i.e., A, B, C) to fulfill the similar requirements. In addition, one application may use a SOAP-based Web Service named D_1 to provide online billing functionality while another application may utilize an HTTP-based API D_2 to fulfill the same functionality. Without identifying the two similar services D_1 and D_2, we can only identify the incomplete service composition pattern, i.e., \{A, B, C\}. However, the complete service composition pattern is \{A, B, C, D\}. Web resources are described in various formats. Some types of web resources (e.g., HTTP-based APIs) are not described using a machine-readable format. Therefore, it is challenging to identify functionally similar services. Although a number of existing approaches (e.g., [101]) handle the automatic discovery of similar SOAP-based Web Services, such approaches are limited to discover a single type of web resources. To identify web resources of different type with similar functionality, a SOA professional needs to manually examine the functionality by reviewing the descriptions of different web resources in various formats. This is a labor-intensive process.

2) A large number of execution logs are needed to correctly identify complete control flows of the service composition patterns. Specifically, existing approaches detect parallelism relations between services if the services appear in random order in different execution logs [22] [3]. Such approaches require a large number of execution logs that reflect all possible execution order among the services. The number of needed execution logs can be extremely large if there are many branches in the parallel control flow structure.
Furthermore, not all runtime execution environments support the execution of parallel services in random order.

To address the aforementioned challenges and identify service composition patterns, the approach described in this thesis features the following parts:

1) **A unified description schema for describing heterogeneous web resources:** To assist the automatic discovery of various web resources with similar functionality, we propose a schema [103] to uniformly describe different types of web resources. The unified representation provides a better chance to discover web resources with similar functionality than limiting the service discovery within the web resources of the same type. Moreover, we develop techniques to automatically extract information from web resources and construct the new description record of each web resource using the unified description schema.

2) **Automatic techniques to extract and describe HTTP-based APIs:** To adopt the proposed unified description schema, it is essential to efficiently generate the description record for each web resource. It is a challenging task to generate a description record from the existing description of an HTTP-based API due to the lack of a machine-readable standard format to describe HTTP-based APIs. Our technique [103] automates the identification of operations using the similarity among HTML tag structures. Based on the identified operations, we can automatically construct the description records for the web resources.

3) **An automatic approach to identify patterns from execution logs:** Although a number of existing approaches analyze the execution logs of service-oriented applications to mine business processes, the existing research focuses on recovering the control flow of
a single business process without extracting patterns shared among multiple applications. We identify frequently executed patterns used by multiple service-oriented applications [102]. To facilitate the reuse of a service composition pattern as an independent and ready-to-use component, we further infer the control flow of a pattern.

4) **A technique to improve the efficiency for identifying control flow structures of service composition patterns:** We use an event graph to represent the execution of a service-oriented application and identify various control flow structures [102]. When the services are executed concurrently, such a technique can identify the parallelism relations between services using a single execution of the application instead of a large number of execution logs.

### 1.4 Organization of Thesis

The remaining chapters of this thesis are organized as follows:

- Chapter 2: We give a brief summary of the existing work related to our approach of identifying service composition patterns.

- Chapter 3: We give an overview of various types of web resources. Furthermore, we present the approach to uniformly describe web resources and find similar web resources to facilitate the identification of service composition patterns.

- Chapter 4: We present the approach to identify service composition patterns from execution logs.

- Chapter 5: We present case studies for evaluating our approach.

- Chapter 6: We conclude and discuss future work.
Chapter 2
Related Work

In this chapter, we give an overview of the Service Oriented Architecture, service discovery, service composition, SOA monitoring and identification of service composition patterns. We also discuss the research related to resources description standards and the techniques for extracting information from existing descriptions for web resources.

2.1 Service Oriented Architecture

Service Oriented Architecture (SOA) aggregates services of various capabilities that may be controlled by different organizations. SOA increases reusability and flexibility as well as reduces costs in business-to-business cooperation. Web Service is a well-known software model used to implement the Service Oriented Architecture in the web environment. We discuss four aspects of Service Oriented Architecture in the following subsections.

2.1.1 Service Discovery

Service discovery is the process to locate desired services according to the functional and non-functional requirements imposed by the potential service consumer. Universal Description Discovery and Integration (UDDI) [11] is a widely used service discovery system. UDDI is a repository that registers entries about Web Services. A Web Service entry in the UDDI contains one or more tModel. The tModel can contain multiple properties. Each property is represented by a name-value pair. To help find a service in a UDDI repository, the UDDI system the keyword searching facility and allow a user to browse the relevant UDDI categories to locate relevant Web Services. Xu et al. [118] propose a service discovery approach that uses the tModel to include
Quality of Service (QoS) information in the UDDI. Blum [12] proposes a QoS-aware service discovery approach that extends UDDI. Bansal et al. [9] extend the Ontology Web Language For Services (OWL-S) to define the semantic Universal Service Description Language (USDL) for automated service discovery. Tosic et al. [105] propose Web Service Offering Language (WSOL) to allow formal specification of service management information, such as classes of service, functional and accessibility constraints, price, penalties and other management responsibilities. The authors propose the Web Service Offering Infrastructure (WSOI) to demonstrate the usability of WSOL in managing and discovering Web Services. Arabshian et al. [7] classify the services using ontology in order to guide users to search for services. Maximilien and Singh [68] provide an ontology-based approach to meet end-to-end QoS requirement. Jakob et al. [54] propose a broker-based service discovery approach to find services with non-functional attributes. Another broker-based service discovery approach is described by Xu et al. [118].

In this thesis, we propose a unified description schema and the techniques for describing web resources. Different from the existing approaches, our schema uniformly represents various types of web resources rather than limiting to one type of web resources. This feature increases the probability to successfully discover web resources that match the requirement. Furthermore, comparing to the heavyweight ontology-based approaches (e.g., [9]), our description schema is more lightweight and the description records for various web resources can be automatically generated.

### 2.1.2 Service Composition

A composite process can be created using a set of services. The process of building such a composite process is commonly called service composition [31]. Service composition has been an
active research area for a long time. Research in service composition can be categorized as manual service composition and automatic service composition.

**Manual Service Composition.** Casati et al. [17] propose eFlow to provide SOA developers with a simple, easy to use mechanism for defining a composite service starting from basic services. The composite services can be preassembled or created on the fly, and can dynamically adapt to changes in the business environment, such as introducing new basic services. Schuster et al. [89] propose Polymorphic Process Model (PPM) to facilitate multi-enterprise service compositions. The main advantage of this model lies in its ability to support both service-based composition and reference process specification-based composition. A reference process is not a concrete service or process. It is an abstract definition of the required activity and allows more flexibility in service composition. JOpera for Eclipse [55] is a tool that assists with composing RESTful Services and other types of web resources.

**Automatic service composition.** Ontologies are used for service composition. Arpinar et al. [8] use ontologies to semi-automatically compose services to form Web Processes by matching the interfaces of individual services. Flügge and Tourchaninova [35] identify the control relations among identified tasks and generate ad-hoc processes from ontologies. Gómez-Pérez et al. [39] develop a framework for composing Semantic Web Services. Majithia et al. [64] use the semantic information in web services to allow automatic composition of web services. Sirin et al. [94] propose a semi-automatic process to present matching services to the user at each step of the composition and use semantic descriptions of the services to help user compose web services. Zhang et al. [120] describe an approach to generate complex web services using interface-matching. Sirin [93] uses AI planning techniques to compose semantic Web Services. Zhao et al.
[121] discuss the challenges of composing RESTful services and propose a formal model for
describing individual RESTful services and automating the composition.

From the perspective of service composition, our work of identifying service composition
patterns can be also used for service composition. Differently from the existing work, which
generates service-oriented applications from scratch, we identify service composition patterns and
allow the reuse of the patterns to compose new applications. Given a similar requirement, a
complete service composition pattern can be reused to fulfill the requirement. Furthermore, the
service composition patterns can complement existing approaches by recommending services.
Hence, our work tackles the service composition problem from the perspective of mining
historical data.

2.1.3 SOA Monitoring

The software industry explores the approaches of monitoring for their proprietary intra-
organizational management application suites [71]. The research communities propose different
instrumentation techniques for collecting performance monitoring data. Sahai et al. [86] propose
an approach to monitor service-oriented applications using message tracking in inter-
organizational processes. IBM Tivoli [50] and Sahai et al. [87] use message interception and code
instrumentation techniques to monitor the service-oriented applications. Machiraju et al. [63]
propose a network intermediary framework for distributed SOA monitoring.

Monitoring service-oriented applications is critical for identifying service composition
patterns because it is essential to obtain execution logs. Our approach is based on the centralized
execution paradigm with a central scheduler. The monitoring is conducted by the central
scheduler in the runtime execution environment. Various monitoring approaches discussed above
are useful for obtaining execution logs from applications deployed in a distributed environment without a central scheduler.

2.1.4 Service Composition Patterns Identification

A great amount of research effort has been devoted in the areas related to service composition patterns. The definition of service composition patterns varies in the literature. In the business process management domain [2] [44], a composition pattern catalogs different forms of service interactions through control flows (e.g., sequential pattern, and loop pattern). Such a pattern does not refer to a particular existing service. Differently, we identify service composition patterns with the combination of services and their control flow. Hence, the identified patterns are more concrete and ready to be used for composing or improving service-oriented applications.

The research related to identifying service composition patterns can be divided into top-down and bottom-up approaches. In the top-down approach [24], the business process management architects can compare the documented business processes from different organizations to identify commonalities and document them as composition patterns. However, this static approach does not consider the frequency of executing the service-oriented applications when identifying the patterns. The bottom-up approaches include services association mining and process mining.

**Service Association Mining.** Dong et al. [28] propose an approach to find patterns between Web Services using interface matching. This approach identifies a pair of related services by examining similarity of their outputs/inputs. The approach is limited to identify a pair of services with matched interfaces. However, the matched services may not have semantic relations. Different from [28], we identify the services that are strongly associated in the service execution by analyzing historical execution logs. Liang et al. [61] mine service association rules from
service transactional data. Their approach aims to discover a binary association relation between two service sets. Our approach finds a sub-process as a pattern for the purpose of reuse.

**Process Mining.** Identifying a service composition pattern heavily relies on the ability to recover a complete business process. Zou et al. [122] use a model-driven business process recovery framework to recover the business process automatically. Hung and Zou [49] recover complete business processes from multiple tiers business applications. Agrawal et al. [5] propose a technique for mining process from execution logs. This approach builds a dependency graph using the order in which activities are recorded in the historical execution logs. For example, the activity A depends on another activity B if A can only happen after B. However, their work did not further identify control flow structures based on the dependency graph. Cook and Wolf [22] develop a similar approach to recover business processes from event logs. Their approach is capable of identifying parallel structures assuming that business tasks in a parallel structure are invoked in random order. Schimm [88] recovers hierarchically structured business processes. Aalst et al. [3] propose an approach to recover processes using Petri net theory. Similar to Agrawal [5], Aalst et al. use tasks in the process logs to construct Petri net to indicate the relationship among tasks. Aalst et al. can identify dependency relations, non-parallel relations and parallel relations to describe the execution order among tasks. In [114], Weijters and Aalst use metrics to construct processes using logs. Specifically, a dependency/frequency table is constructed based on the order of occurrences between each pair of tasks. The graph that represents task execution order is induced from the dependency/frequency table. This approach can handle execution data with noise. Greco et al. [41] mine multiple variants of process to represent different usage scenarios. Bose and Aalst [14] propose an approach to cluster instances
to handle less structured process models. Di Francescomarino et al. [36] develop an approach to recover business process of web applications by mining GUI-form traces.

Similar to existing approaches, we can identify sequential, alternative, parallel and loop control flow structures. We improve the technique for identifying parallelism. Different from the existing approaches which focus on recovering a single entire process, we mine the frequently executed patterns used in multiple service-oriented applications. Similar to the existing approaches (e.g., [5]), we use the process mining techniques to recover the control flow within a pattern.

2.2 Resource Description

Resource descriptions specify the information and functionality provided by the resources. The descriptions allow the potential consumers to discover and utilize the resources. The descriptions of resources are also known as metadata, i.e., the data about data. The existing work related to resource descriptions can be categorized as the schemas of resource descriptions, and the methods to generate resource description records.

2.2.1 Schemas of Resource Description

The library, museum and research communities propose various metadata standards to facilitate the description of information resources. Both traditional offline resources (e.g., books) and online resources (e.g., web pages) can be described by metadata. The Dewey Decimal Classification (DDC) system [109] is an early metadata schema for the libraries to index the books in their collections. Using DDC, a metadata record describes the title, the author, the subject matter, the abstracts and the physical location of the book. MAchine-Readable Cataloging (MARC) [67] is a metadata schema used by libraries to allow the representation and communication of bibliographic information in a machine-readable form.
Much research has been conducted to derive metadata schemas to describe web resources. The working group of the Internet Engineering Task Force (IETF) proposes Internet Anonymous Ftp Archive (IAFA) templates [26] to allow effective access to FTP archives. This template describes the contents available in the FTP archive. Platform for Internet Content Selection (PICS) [81] is a mechanism for communicating ratings of web pages to the Internet clients. The metadata records of PICS contain information about the content of web pages. Dublin Core (DC) [30] describes various information resources on the web or offline, such as books, video, sound, image, text files, and web pages. To represent a resource, the Dublin Core schema defines the semantics of 15 elements. Examples of these elements include Title, Subject, Description, Creator, Publisher, Contributor and Date. The Dublin Core metadata can be expressed in different syntax. For example, many Dublin Core-based projects embed the Dublin Core metadata directly into web pages using the HTML META tag. Social bookmarking [25] is a method for Internet users to organize, store, manage and search for bookmarks of web resources. Descriptions may be added to these bookmarks in the form of metadata, so users may understand the content of the web resources.

The aforementioned metadata schemas are primarily intended to describe documents or “document-like objects”. They focus on representing the bibliographic information of the resources and have limited capacity to describe the functionality of web resources accurately. Different from the aforementioned schemas, our schema describes the general bibliographic information about the web resources and uses the tag-based description and the formal interface to represent the functionality of web resources more accurately.

The Resource Description Framework (RDF) [108] by World Wide Web Consortium (W3C) permits encoding, exchanging, and reusing structured metadata. It is used as a general method for
conceptual description or modeling of information using a variety of syntax formats. However, the RDF is not a metadata schema. A metadata schema includes the definition of controlled vocabularies with the possible values and semantics. RDF can be used to express various metadata schemas (e.g., Dublin Core). Ogbuji [74] uses RDF in conjunction with other standards, such as WSDL to describe a Web Service. This approach can take advantage of the existing RDF based search engines and classification systems. Another work named “RDF in attributes” (RDFa) [80] is a general purpose approach for embedding RDF data in HTML.

**Description Standards for Web Services.** A number of standards, such as WSDL [19] and DAML-S [23], are used to share the metadata about a Web Service. Web Services are functionality-oriented resources. WSDL represents the metadata of the functionalities and the invocation details of the Web Service. DAML-S uses ontologies to enable automation of service compositions on the Semantic Web. Using the DAML-S elements, the capabilities of Web Services can be described without ambiguity. This feature enables automated service discovery and composition. The Large Scale Distributed Information Systems (LSDIS) Lab develops the METEOR-S [75] project to extend the existing industrial Web Services standards by adding semantics for the purpose of Web Service automation. Semantic Annotations for WSDL (SA-WSDL) [58] utilizes data schemas with semantic information to facilitate data mediation. These schemas describe Web Services only but do not facilitate the ability to compose various kinds of web resources.

**Description Standards for HTTP-based APIs.** Several schemas of machine readable description can be utilized to describe HTTP-based APIs, e.g., Web Application Description Language (WADL) [110], WSDL 2.0 and HTML for RESTful Services (hRESTS) [59]. The WADL proposed by Sun Microsystems is an XML-based format to provide a machine-readable
description of HTTP-based APIs. WSDL 2.0 provides additional support to HTTP-based APIs comparing to its precedent, WSDL1.1. WADL and WSDL 2.0 are structured with considerable complexity. hRESTS uses a microformat approach to provide a simple machine-readable description of HTTP-based APIs. The operations, inputs and outputs are described. MicroWSMO [65] is a semantic annotation mechanism for HTTP-based APIs. The MicroWSMO extends hRESTS to provide links to semantic information. SA-REST [85] is similar to SA-WSDL and facilitates mashups of HTTP-based APIs. Despite the numerous efforts to derive a machine-readable description schema for HTTP-based APIs, the schemas do not gain popularities with the API providers. As a result, most HTTP-based APIs remain to be described in text shown in web pages, which is not machine-readable. Our work facilitates the transformation of the existing descriptions from text into a structured machine-readable schema.

2.2.2 Techniques for Generating Resource Description Records

To deploy a metadata schema, the metadata records for the resources can be generated manually or automatically. In the manual way, professional metadata creators and resource authors are the metadata creators. Metadata professionals have formal training and are proficient to use the metadata standards. Studies show that the metadata professionals generally produce high-quality metadata [112]. However, it is impractical to hire many metadata professionals to describe the explosive amount of resources available on the web. The resource authors can provide descriptions of their own work. A number of digital library projects (e.g., National Digital Library of Theses and Dissertations [73]) support author-generated metadata. Greenberg et al. [43] show that although the authors have the ability to create acceptable metadata, they are reluctant to do so due to various reasons.
Automatic generation of metadata is widely studied. Greenberg [42] introduces extraction techniques to facilitate automatic metadata generation for web pages. Craven [21] proposes techniques for processing information in the HTML META tags of a web page. GRDDL [45] is a mechanism for extracting RDF information from web pages. Automatic information extraction from web pages is widely used by search engines. The search engine indexing collects, parses, and stores data to facilitate fast and accurate information retrieval. Sergey and Lawrence [90] introduce the techniques used by a large scale search engine for extracting and organizing information from web pages. Raghavan and Garcia-Molina [79] discuss the techniques to extract information from HTML forms (i.e., web forms). The typical way to generate WSDL is from the source code. Many SOAP toolkits [115] generate WSDL files from existing program interfaces. ASSAM [46] is a tool to enable the annotations of services with WSDL-based descriptions. Heß and Kushmerick [47] use machine learning and clustering techniques to augment services with semantic metadata. Specifically, using a clustering technique, the web services are categorized into classifications based on the details available in WSDL. The automation of generating descriptions for HTTP-based APIs is less explored. Maleshkova et al. [66] propose SWEET to enable developers to identify different parts (e.g., operations, input and output) of the HTTP-based APIs. The approach of identifying operations is mostly manual, with limited tool support. Our approach to creating a machine-readable description for HTTP-based APIs increases the level of automation. Our approach is complementary to other techniques that extract and process metadata from various web resources. We combine the existing techniques to describe all types of web resources.
2.3 Information Extraction

Information extraction transforms text into a structured format. Eikvil [32] surveys different approaches of information extraction. Various approaches are proposed to extract information from free text and structured text.

**Free Text.** Natural Language Processing (NLP) techniques are widely used to extract key information from natural language texts. One example is extracting the key information (e.g., the perpetrators, their affiliation, the location and the victims) from news articles about terrorist attacks [32]. The extraction rules are typically formed from patterns involving linguistic syntactic relations between words or semantic classes of words. To analyze the free text and extract information, several steps are required, including analyzing syntax, Part-Of Speech (POS) tagging and recognizing domain objects (e.g., person and company names). The rules or patterns can be hand-coded or generated from training examples annotated by a human expert.

Among the natural language processing techniques, the POS tagging is the process of labeling words in a text (i.e., corpus) as a particular part of speech, based on the definition and context of the words [99]. The process of POS is non-trivial because some words can represent more than one part of speech in different contexts and some parts of speech are complex or unspoken. The POS tagging technique is critical for extracting key information from free text. We use the POS tagging technique to process the free text paragraphs encountered in the descriptions of web resources.

**Structured text.** Structured text is textual information in a database or file described in a predefined and strict format [32]. Such information can be easily extracted using the format description. Instead of the heavyweight NLP techniques, simple techniques are sufficient for
In our research, the existing description (i.e., WSDL) of SOAP-based Web Services is treated as structured text with a known format. Hence, information can be easily extracted from WSDL. We develop techniques that transform the existing description of web resources into the unified description schema, which is treated as structured text.

**Web pages.** The nesting HTML tags are used to structure the web pages. Using server-side scripting, machine generated web pages are well structured. The HTML tags are often used as a hint to extract information from web pages. Many web pages do not use textual paragraphs with rich grammatical characteristics. Hence, the traditional NLP techniques are not well suited to process such web pages and the syntactic and the semantic analysis can be utilized to a limited extent. In our work, we extract information of web resources from web pages by combining techniques in order to leverage the structures of web pages and the linguistic characteristics of textual paragraphs.

### 2.3.1 Information Extraction Systems

**Wrapper Generation System.** The Artificial Intelligence (AI) communities automate the learning of websites using machine learning techniques. Much work is devoted to manual, semi-automatic and automatic generation of wrappers to extract information from websites. A number of wrapper generation tools are developed. These systems do not analyze the linguistic characteristics of the text. Instead, they derive patterns using the structure of the document.

The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS) system [104] is a framework for manually building website wrappers. Languages and tools are developed to support the wrapping process. This system assumes that a human developer examines a website
and manually codes wrappers to extract information from the web pages. ShopBot [29] is a comparison-shopping agent, specialized to extract information from web vendors. In the learning phase of ShopBot, the vendor sites are analyzed to learn a symbolic description of each site. The Wrapper Induction ENviroment (WIEN) [60] is a tool for assisting in constructing wrapper by automatically learning web pages. SoftMealy [48] is a system that learns to extract data from semi-structured web pages. The wrapper induction takes a set of labeled information as input and derives a rule that matches the training example. STALKER [72] is a supervised learning algorithm for inducing extraction rules. The user selects a few sample pages and marks up the relevant data. The sequences of tokens (words, HTML tags) and wildcards can then be used as landmarks to locate an item to extract from a page. The wrapper-induction algorithm generates extraction-rules expressed as simple landmark-grammars. Junglee [57] presents the concept of “virtual database technology”. It treats the web as an extension of the database and develops a proprietary system to extract information from web pages using hints provided by HTML tags.

The wrapper generation systems require manually labeled training example. The intended use of these wrapper generation systems is to train a wrapper to extract information from an ever-updating website, e.g., for a news broadcasting website [32]. The training needs to be performed once and the resulting wrapper can be used for a relatively long period of time. In this case, manually providing the training set might be worthwhile. However, this approach does not suit the information extraction for describing a web resource. The web resource description is not updated in a frequent pace. It is not cost-effective to manually training a wrapper. Hence, our approach aims to automatically identify the information from web resources without the manual training process.
**Web Page Clone Detection.** Clone detection searches for duplicate fragments of source code in software system. Much work is devoted to detecting and resolving clones in web pages and web applications. Boldyreff and Kewish[13] detect and reduce duplications in websites. Ricca and Tonella [84] present a related approach to identify web pages with similar structure in order to create a page template to facilitate dynamic generation of individual pages by server-side scripts. Baxter et al. [10] locate clones by detecting similarities in parse trees built from the code. Cordy et al. [20] propose an approach to detect near-miss clones in HTML pages. These approaches are intended to detect clones introduced by the practice of copy-and-paste. They aim to reduce the unwanted clones to improve the quality of the web pages and web applications. Our approach does not consider the similar structures in web pages as harmful. Instead, we utilize the similar structures introduced by server-side scripting to detect the operations of HTTP-based APIs described in text. Embley et al. [33] propose an approach similar to our approach to extract information from web pages with multiple records. Their approach assumes a vast number of records on the web page and chooses the HTML tag with most children as the HTTP tag that contains multiple records. This assumption is not always correct for the description of operations for web resources. Our approach traverse the web page to ensure the operations can be identified.

### 2.4 Summary

In this chapter, we survey various academic and industrial research related to our work. Research on Service Oriented Architecture, metadata standards and information extraction are covered. Particularly, we discuss the existing work on extracting information from web pages and compare them with our approach. Furthermore, we discuss the existing research on process mining in greater details and compare them with our approach.
Chapter 3

A Unified Schema for Describing Heterogeneous Web Resources

To assist the automatic discovery of various web resources with similar functionality, we propose a schema to uniformly describe different types of web resources. The unified representation provides a better chance to discover web resources with similar functionality than limiting the service discovery within the web resources of a single type. Moreover, we develop techniques to automatically extract information from web resources and construct the new description record of each web resource using the unified schema.

In this chapter, Section 3.1 gives an overview of various web resources. Section 3.2 proposes a unified description schema to represent the web resources. Section 3.3 discusses the techniques for extracting information from web resources and describing them using the unified description schema.

3.1 Overview of Web Resources

In this section, we introduce various types of web resources: web pages (e.g., informational web pages and web pages with web forms), SOAP-based Web Services, and HTTP-based APIs. In our work, we study these web resources due to their extensive usages.

Web pages. A web page includes various kinds of information, such as text, images, audio and video. The information in a web page is presented in HTML format and provides navigation to other web pages via hyperlinks. A typical layout of a web page is shown in Figure 3-1. Visually, a web page can be divided into three parts: the header, the content part and the footer. The header contains common information to all web pages within one website, such as the logo
Figure 3-1 Example of an informational web page

of the website and navigation menus. The content part delivers unique information. The footer includes contact and copyright information. It can be identical throughout the website. Moreover, an HTML file of a web page can carry information invisible to the end-users, such as comments, META tags, scripts and style information. The META tags contain the title, textual description and keywords of the web pages.

A web page is rendered by a web browser into a vivid visual format. When composing a service-oriented application, the web pages are used as a user interface to interact with the end-users. We categorize web pages into two groups according to their interaction with end-users: (1) Informational web pages which give factual information on a specific topic or event; and (2) Web forms which are used to collect information from the end-users by allowing end-users to enter
Figure 3-2 Example of a web page that contains a web form
data and send the data to the server for processing. A web form consists of multiple elements
represented by HTML tags, e.g., text input field, checkbox, and submit button. Web forms can be
combined with various client-side or server-side scripting languages to allow developers to create
dynamic websites. An example of web form is shown in Figure 3-2.

**SOAP-based Web Services.** A SOAP-based Web Service is a self-contained software
component. It communicates with other applications using the SOAP protocol over the HTTP
which served as a transport protocol. The functionality offered by a SOAP-based Web Service is
described using WSDL. In WSDL, the abstract definition of the endpoints and messages is
separated from the concrete network deployment or data format bindings [18]. Hence, as shown
in Figure 3-3, a WSDL file can be divided into two sections: the abstract section and the concrete
Figure 3-3 Example of a WSDL file for a SOAP-based Web Service

In the abstract section, a collection of operations is defined in the “interface” part. The operation name, input parameters and output are wrapped by predefined WSDL tags. In the concrete section, the underlying transport protocol and the address of the service are defined. A client program connecting to a Web Service can read the WSDL to determine the available operations. The client can then use SOAP to call the operations listed in the interface specified in WSDL.

SOAP-based Web Services can handle asynchronous processing and invocation. SOAP-based Web Services are especially useful to integrate legacy systems into service-oriented applications. In particular, the Web Service infrastructure provides industrial standards (e.g., Web Services Reliable Messaging (WSRM)) and APIs (e.g., Java APIs for XML Web Services with the support of client-side asynchronous invocation) to ease the integration. The transaction
management and security are well supported. Hence SOAP-based Web Services can be used to satisfy complex non-functional requirements.

**HTTP-based APIs.** An HTTP-based API communicates with other applications using a defined set of HTTP requests and the corresponding responses. An HTTP request uses the HTTP verb and URI to indicate the functionality (i.e., operation) to invoke. The response is encoded in XML or JavaScript Object Notation (JSON) format. In addition, the response can be described in proprietary formats specified by the providers of the APIs. Many leading Internet companies, such as Yahoo, expose their data and functionalities as HTTP-based APIs and make them a valuable source for composing service-oriented applications.

Industrial efforts attempt to standardize the descriptions for HTTP-based APIs using Web Application Description Language (WADL) and WSDL2.0. However, no specification languages are widely accepted. Typically, HTTP-based APIs can be described using web pages in two ways: single-page description and multiple-page description. Single-page description is used to describe an HTTP-based API with a moderate number of operations. If an API contains a large number of operations listed in a single web page, it leads to an oversized web page which hinders the transmission of the description over the Internet and makes it difficult to read. Hence, multiple-page description is used by complex HTTP-based APIs with comprehensive operations. Each web page describes one operation. A portal web page lists hyperlinks to the web pages of individual operations. The description of an operation includes the description of the functionality, the input and the output. An example description of an HTTP-based API is shown in Figure 3-4. In the figure, three operations of the API are described using identical appearance style one after another.
RESTful services are special HTTP-based APIs. They are resource-centric as opposed to other operation-centric HTTP-based APIs. In RESTful services, no operations are explicitly defined. Each URI represents a resource that is associated with HTTP verbs: POST, GET, PUT and DELETE. The HTTP verbs allow create, retrieve, modify and delete the resource. A RESTful service is suitable for representing and manipulating information. If we need to provide functionality to perform actions, such as shipping an order, it is difficult to use RESTful services. In practice, few HTTP-based APIs strictly follow the resource-centric style, although many HTTP-based APIs are referred as RESTful services [78].
3.2 A Unified Schema for Describing Web Resources

To facilitate the discovery of various web resources with similar functionality, we propose a unified description schema to describe resources of different types as shown in Figure 3-5. We define two parts in the unified schema: the general description part and the operation description part.

The general description of a web resource provides a bibliographic description about the web resource: the type, the name, the provider and the URI of the web resource. Such descriptions are common to all types of web resources. There are standards suitable for representing the general description. For example, using 15 text fields (e.g., title, type and publisher), the Dublin Core metadata schema can describe various resources, e.g., books and web
Among the four fields of general description, the type, the name, the provider fields are self-descriptive. The URI field of the general description refers to the URI that identifies the web resource on the Internet. The URI of a SOAP-based Web Service points to its WSDL file. For an HTTP-based API, the URI field is filled by the URI of its description web page. For web pages, the URI field refers to the web page itself. To invoke a particular functionality of the web resource, another URI may be required because each operation of the web resource may have a different URI, which is described in operation description part.

**The operation description** describes the functionalities offered by a web resource. A web resource can deliver one or more functionalities. An operation represents a primitive unit of functionality used to compose a service-oriented application. Different types of web resources contain varied number of operations. Most SOAP-based Web Services and HTTP-based APIs provide complex functionalities and contain multiple operations. The functionality of an informational web page is providing information to the end-users. We treat such an informational web page as one operation. A web page may contain one or more web forms, each of which provides a particular functionality (e.g., searching or submitting information). Hence, we consider each web form as an operation. For a RESTful service, the HTTP verbs associated with the resources can be converted as operations. For example, if the resource “book” is associated with two HTTP verbs (i.e., GET and DELETE), we can obtain two operations: getBook and deleteBook.

To describe each operation, we use the tag-based description, the formal interface and the excerpt of existing description. The tag-based description uses a set of descriptive tags (i.e.,
keywords) to informally represent an operation, including the functionality description, the input
description and the output description. The input/output descriptions help describe different
operations with the same name or similar functionality description. For example, two operations
are named as “displayOrders”. One operation with the parameter “productId” is different from
the other one with the parameter “customerId”. The tag-based description can concisely convey
the functionality of the operation to the resource consumers. In addition, it facilitates
automatically compare the functionalities of operations in order to discover similar web
resources.

The formal interface provides information to support the invocation of an operation. The
formal interface is intended for machine consumption in order to facilitate automatic invocation.
The SOAP-based Web Service is originally described with a formal interface. Hence, a client
program can be automatically generated to invoke operations in a SOAP-based Web Service. In
contrast, other web resources (e.g., the HTTP-based APIs) do not have a formal interface and the
SOA professionals need to manually write the request to invoke the operation. The fields
contained in the formal interface depend on the type of web resources. To invoke an HTTP-based
API operation and a web form, the URI, the HTTP verb, and the input parameters of the operation
are required. Only URI is required to access an informational web page because the HTTP verb is
fixed to be GET and no parameter is required. The formal interface is described in an XML
format which can be interpreted by a machine to automatically invoke the operation.

An excerpt is taken from the existing description of an operation. It provides more readable
and detailed information, such as examples and demonstrations. It also offers a shortcut for a
SOA professional to understand the operation without having to search for the operation in the
entire document. The excerpt is available only for SOAP-based Web Services and HTTP-based
APIs. In the example shown in Figure 3-4, the existing description of the operation “get a review” can be extracted as an excerpt.

3.3 Techniques for Describing Web Resources Using the Proposed Schema

To represent various web resources in a unified format, we analyze the existing descriptions of web resources to extract the information required by the proposed unified description schema. In the following subsections, we discuss our technique that (1) analyzes the existing description file of the web resources to identify the operations offered by the web resources; (2) extracts descriptive tags from the existing description for each operation; (3) constructs a formal interface for each operation to facilitate automatic invocation; and (4) assembles the descriptive tags, the formal interfaces and the description excerpts to construct a complete description for the web resources.

3.3.1 Parsing Description Files

To analyze the existing descriptions of web resources, we need to parse the description file in order to build a Document Object Model (DOM) tree structure. The existing description files can be in HTML or WSDL format. The description file in WSDL format is easy to parse due to the well conformance to the WSDL specification. Unlike the WSDL file, malformed HTML files are quite common in the web. An HTML file may contain mismatched HTML tags although it can be correctly displayed by web browsers due to the fault-tolerance capability of web browsers. To generate DOM tree structure from an HTML file, we use HTML syntax checker [56] to correct the malformed HTML files. Then we parse the HTML into a DOM tree structure. For example, the HTML fragment shown in Figure 3-6 is converted to a DOM tree structure depicted in Figure 3-7. Each node in the DOM tree structure corresponds to an HTML tag.
3.3.2 Identifying Operations of Web Resources

To describe web resources using the proposed schema, we need to identify their operations. The operations of the SOAP-based Web Services and informational web pages are relatively easy to identify. For SOAP-based Web Services, the <operation> tags in the WSDL file explicitly indicate the definition of the operations of a SOAP-based Web Service. Therefore, we can

**Figure 3-6 Example of HTML fragments of HTTP-based API operations**
directly identify them using the <operation> tags. An informational web page is interpreted as providing one operation (i.e., the “retrieve information” operation). No processing is required to analyze such a web page. For the web pages that contain web forms, the form tag <form> defines a web form. Therefore, each form tag, <form>, is interpreted as an operation.

It is more challenging to identify operations from the existing description of an HTTP-based API due to the lack of a machine-readable standard format to describe HTTP-based APIs. A web page that describes an HTTP-based API is essentially plain text annotated by decorative HTML tags, such as paragraph tag, <p> and division tag, <div>. HTML tags do not indicate the meaning of the enclosed plain text. For the example shown in Figure 3-6, the operation names are wrapped by anchor tag, <a>. However, the operation names in another HTTP-based API might be wrapped by the paragraph tag, <p> or other HTML tags. Due to the diversified use of HTML tags to describe the operations, a generic rule cannot be derived to identify operations using the type of
HTML tags. The existing approach [66] requires the developer to view the description and manually identify each operation.

To automate the identification of operations, we compare the similarity among the HTML tag structures. The API provider describes all operations in an identical style of appearance using server-side scripting (e.g., PHP [77]). The HTML fragments for different operations have similar HTML tag structures. Figure 3-6 shows two similar HTML fragments corresponding to two operations: “Shop.activity.ShipOrder” and “Shop.activity.VerifyCreditCard”. Figure 3-7 shows the tree structure of the HTML tag structures that correspond to the two operations shown in Figure 3-6. The left part of the tree (i.e., the nodes labeled as 1 to 10 and their descendants) represents the operation named “Shop.activity.ShipOrder”. The right part of the tree (i.e., the nodes labeled as 11 to 20 and their descendants) represents the operation named “Shop.activity.VerifyCreditCard”. From the tree, we can observe that the HTML tag structures of the two operations are similar.

Using the similarity of HTML fragments, we design and develop an algorithm to identify operations in two steps: (1) locate a set of HTML fragments with similar HTML tag structures; and (2) verify whether the located HTML fragments correspond to operations in order to filter out the false positives when other irrelevant content may contain similar HTML tag structures. If the identified HTML fragments are false positive, we iteratively locate another set of HTML fragments with similar HTML tag structures. The algorithm ends when we find the operation or until we verify all the similar HTML fragments. Because of the difference between the single-page description and multiple-page description, we develop two versions of the algorithm: single-page algorithm and multiple-page algorithm. The pseudo code of the two algorithms is shown below.
//Single-page algorithm
Process_Single_Page (Node) {
  Obtain the children list of Node
  Foreach candidate tag structure
    Find matching tag structures among the children of Node
    If found matching tag structures
      Search for smaller tag structures within the candidate
      Verify the found tag structures using heuristics
      If passed verification
        Return found tags structures as operations
  Foreach Child of Node
    Process_Single_Page (Child)
}

//Multiple-page algorithm
Process_Multiple_Page (Portal_Page_Root, Linked_Pages_Root[]) {
  Foreach Linked_Pages_Root[x]
    Remove_header (Portal_Page_Root, Linked_Page_Root[x])
    Remove_footer (Portal_Page_Root, Linked_Page_Root[x])
  Foreach Linked_Pages_Root[x]
    Foreach Linked_Pages_Root[y], y>x
      Compare tag structures of page x and page y
      If similar
        Similar_to[y] = Similar_to[x]
        Count[ Similar_to[y]]++
  Find the page P with max count[P]
  Return page P and the pages similar to P
}
Locating similar HTML tag structures. To locate HTML fragments with similar HTML tag structures, the single-page algorithm and multiple-page algorithm work differently as discussed below.

A portal web page and all web pages linked by the portal web page are the input to the multiple-page algorithm. We have no prior knowledge about whether such web pages correspond to an HTTP-based API or not. If these web pages correspond to an HTTP-based API, most of the web pages linked by the portal page should be the descriptions of individual operations. Then such web pages should have similar HTML tag structures. However, the portal web page may link to web pages in other websites. For example, a portal web page of an HTTP-based API (e.g., http://www.example.com/api) may link to the website of an enabling technology, JSON (i.e., http://www.json.org/) as a reference. The description of the individual operations should be within the same website. Hence, we filter out web pages in other domains. It is common that the header and footer of all web pages in the same website are identical. The identical header and footer would mislead the algorithm to misidentify irrelevant web pages as operation descriptions. We remove the headers and the footers in all web pages to avoid such a case. There are no HTML tags available in the web pages that explicitly indicate the boundary of header and footer. We remove the header by comparing the beginning part of the linked page with the portal page. The identical content at the beginning is the header to remove. The same technique applies to the footer. The content part of each linked page remains. After filtering, we compare the HTML tag structures of the linked web pages. If a set of linked pages have similar HTML tag structures, they are fed to the verification step to determine whether such identified HTML fragments correspond to operations.
A single web page is served as the input to the single-page algorithm. Similarly, we have no prior knowledge about whether it is a description of an HTTP-based API. The algorithm is more complex as opposed to the multiple-page algorithm. In multiple-page description, only each linked page is possible to map to an operation. The potentially similar HTML tag structures are separated by different web pages. Therefore, we do not need to consider HTML fragments of varied sizes. In contrast, in a single-page description, we need to enumerate HTML fragments of varied sizes as candidate to search for similar HTML tag structures.

Assume that there are \( n \) children HTML tags and these children are numbered from 1 to \( n \). The candidate structure \((a \rightarrow b)\) is a sequence of HTML tags starting from the \( a \)-th child to the \( b \)-th child where \( 1 \leq a \leq b \leq n \). For example, when we search for similar HTML tag structures among the children of the \(<body>\) tag shown in Figure 3-7, the candidate \((3 \rightarrow 5)\) is the HTML tag structure “p ul p”. We can find one match for this candidate as labeled in Figure 3-7. However, the candidate and the matching structure is a fraction of the similar HTML tag structure under the \(<body>\) tag. To avoid detecting a fraction of similar structure, we start from the largest possible candidate that contains most HTML tags among the children of the current tag being processed, i.e., the \(<body>\) tag in this example. The total number of children tags of the \(<body>\) tag is \( n \) (i.e., \( n = 20 \)). Hence, we start from candidate with the size \( n / 2 = 10 \), which is the largest size with possible matches. If no matches can be found for the candidates of size \( n / 2 \), we decrease the size by one, and so forth.

If similar tag structures are discovered among the children of an HTML tag, the identified similar tag structures may not map to operations because a structure can be two or more operations concatenated together. We further detect whether it can be broken down into smaller similar structures.
If there is no similar structures among the children of an HTML tag, we recursively process each child of the HTML tag using a depth-first traversal. In the example shown in Figure 3-7, we start from the root tag in the tree structure: the <html> tag. We cannot find any similar structures among its children: the <head> tag and the <body> tag. Then we search for similar structures among the children of the <head> tag with no success. Finally, we find similar tag structures among the children of the <body > tag.

**Verifying HTML fragments with similar tag structure.** The identified HTML fragments with similar HTML tag structures may not correspond to operations since any web page that contains a list of items (e.g., a web page introducing all albums of a singer) can contain similar HTML tag structures. We use the following two heuristics to examine if the identified HTML fragments are operations:

1. If the words contained in an HTML fragment are too few, it is unlikely that the HTML fragment is used to describe an operation. The operation description of the HTTP-based APIs we have studied have more than 30 words. Therefore, we use 30 as the threshold.

2. If the HTML fragment does not contain keywords related to an API, such as “parameter”, “argument”, “response” or “output”, it is unlikely that the HTML fragment is mapped to the description of an operation.

### 3.3.3 Generating Descriptive Tags for Operations

As described in the schema for describing operations, the descriptive tags for an operation are divided into three parts: the functionality description, the input description and the output description. To generate descriptive tags, we extract words from the existing description of the operation and combine with the descriptive tags retrieved from external sources, such as social bookmarking repositories.
**Extracting descriptive tags from existing description.** To extract descriptive tags for the functionality description, the input description and the output description of a web resource operation, we locate the three parts in the existing description. Then we extract and process descriptive tags for each part. The three parts for an SOAP-based Web Service operation can be located using WSDL tags. For the operation that represents an informational web page, the functionality description is always “retrieve information” since all informational web pages provides information to end-users without input parameter. The META tags indicate the content or the informational web page. We treat the META tags as the output description of the “retrieve information” operation.

The boundaries between the three parts are not explicitly defined in the description of an HTTP-based API operation. However, the three parts are delimited by keywords, e.g., “parameter”, “argument”, “response” and “output”. We use these delimiter words to locate the three parts. In the example shown in Figure 3-6, the text appearing before the delimiter word “argument” belongs to the functionality description. The text appearing after the delimiter word “response” belongs to the output description. The text in between is the description for the input parameters.

It is challenging to locate the functionality description and output description for web forms due to the varied style and layout of the web pages containing the web forms. The description for the input parameters is extracted from the HTML tag `<input>`.

The number of words in the textual descriptions of each part of the description may be considerably large and many of them may be irrelevant. We consider that nouns and verbs are more important and extract them as descriptive tags. Some words can be used both as noun and adjective, such as “deliverable”. To find out the actual usage of a word in the sentence, we apply
a Part-Of-Speech (POS) Tagger [99]. POS tagger reads text in a natural language and assigns parts of speech to each word, such as noun, verb and adjective. In this way, we correctly filter out other types of words. We treat a noun and its modifiers as an atomic phrase, such as “music festival”, which represents a unique meaning. Furthermore, we remove all stop words. Finally, we detect synonyms using WordNet [116] and keep one of them.

**Collecting descriptive tags from external sources.** The descriptive tags that are automatically extracted may not be as accurate as the descriptive tags that are manually added by human end-users. The social bookmarking repository contains descriptive tags for web resources created by a large number of end-users. Therefore, it provides a good source to obtain high-quality descriptive tags. We use these descriptive tags to augment the extracted descriptive tags. Many social bookmarking websites, such as Delicious [25], allow the public to access their repository data via published APIs. The descriptive tags for the web resources can be retrieved by submitting the URI of the web resources. If a descriptive tag is both extracted from the existing description of a web resource and collected from social bookmarking repository, we mark it as a more reliable descriptive tag.

### 3.3.4 Constructing Formal Interface for Operations

To generate the formal interface as specified in Figure 3-5, we locate the URI, HTTP verbs and input parameters depending on the resource type. More specifically, a URI is required for invoking an operation extracted from HTTP-based API, a web form or an informational web page. The URI of a web form is defined in the “action” attribute of the `<form>` tag. For an HTTP-based API operation, we locate the URI using string pattern matching with regular expression specified in [107]. We do not only match URI beginning with “http”. We also detect the URI which does not begin with “http”. In a number of API descriptions, the URIs appearing in the
operation descriptions are incomplete due to the omission of the base URI when describing each individual operation. For example, all operations of an API can share the base URI http://www.example.com/api/v1. This base URI is indicated once at the beginning of the description of the API. However, it is not described in each operation. The URI of one operation can be described as a partial URI, e.g., “/checkbalance?p=para”, which needs to be combined with the base URI to form a valid URI. The regular expression is also used to detect part of URI.

It is necessary to use the HTTP verbs to construct an HTTP request for invoking an operation of an HTTP-based API, a web form or an information web page. For an HTTP-based API operation, we locate the HTTP verbs by matching the keywords GET, PUT, POST and DELETE. The HTTP verb of a web form can be located in the “method” attribute of the HTML tag <form>. The HTTP verb used to access an informational web page is GET.

In addition, the input parameters are needed to invoke an operation of an HTTP-based API or a web form. For an HTTP-based API operation, the description of each parameter follows the same HTML tag structures. The parameter name is usually located at the beginning of the description of the parameter. Therefore we detect parameter descriptions and extract the first word of each parameter description as the parameter names. The parameters of a web form are indicated by the HTML tag <input>.

3.4 Summary

In this chapter, we present a unified description schema that can represent various types of web resources to facilitate service discovery. The schema uses two parts to describe a web resource: general description and operation description. Each operation is described using descriptive tags, formal interface and the description excerpts. We also discuss the techniques that
automatically generate description records using the schema. We describe the technique to extract information from HTTP-based APIs in details.
Chapter 4

Mining Service Composition Patterns from Execution Logs

In this chapter, we present an approach to identify frequently used service composition patterns from execution logs. Section 4.1 gives an overview of our approach. Section 4.2 presents in detail the approach for identifying a set of services commonly used together. Section 4.3 discusses the approach for identifying control flows among the services in a service composition pattern.

4.1 Overview of the Approach to Mine Service Composition Patterns

Figure 4-1 gives an overview of our approach which is broken down into three major steps: (1) collect and pre-process execution logs; (2) identify a set of associated services that are frequently executed together; and (3) recover the control flows among these services.

Collecting and pre-processing execution logs: The composed service-oriented application is deployed and executed in a runtime execution environment, such as IBM WebSphere Process Server [52]. Once a service-oriented application is deployed, it can be executed in many execution instances. Each execution instance is uniquely identified with an identifier (i.e., the instance id). Execution instances of different applications may co-exist. In each execution instance, events can be triggered. We record the triggered events in the logs using the logging facility provided by the runtime execution environment. An execution log contains different types of events. For example, resource adapter events record the interactions between the service-oriented application and a legacy system. Business rule events track the runtime status of business rules. Service invocation events indicate the timeline of a service execution. We are interested in
the service invocation events. In particular, an ENTRY event is triggered when a service is
invoked. An EXIT event occurs when a service completes the execution and returns the result.
Each event is recorded with the time of triggering, the identifier of the service which triggers the
event, the execution instance id and the identifier of the service-oriented application. The built-in
logging facility of the runtime execution environment does not allow specifying the types of
events to record. All the events are captured in the execution logs. We process the logs to extract
the service invocation events.

Different service-oriented applications may use operations from different types of web
resources to provide similar services. To identify functionally similar services, we extract the
URIs of the services from the ENTRY events in the execution logs. We search the URIs in the
existing collection of description records. If a URI is not found, we obtain the existing description
of the corresponding web resource from the Internet and transform it to the unified description
schema using the techniques described in the previous chapter. Then we detect similar services by
comparing the descriptive tags of all services that appear in the execution logs. If two services are

Figure 4-1 Overview of the approach to mine service composition patterns
found as similar, we use the same identifier to label them in the pre-processed execution logs. In this way, the subsequent steps will treat the two services as identical.

**Identifying frequently associated services:** Each execution instance invokes a set of services. To detect a service composition pattern, we find a set of associated services that frequently appear together in many execution instances. Such frequently associated services form the service set for a service composition pattern (i.e., the pattern service set). The number of services in a service composition pattern is important. A service composition pattern with very few services (e.g., two services) may provide incomplete functionality with minimal value for reuse. We propose that a service composition pattern contains at least four services. However, the required minimum number of services may vary in different settings. We generate different service sets as candidates (i.e., candidate service sets) and calculate their frequencies. We select the candidate service set with highest frequency as the intermediate pattern service set. From the intermediate pattern service set we construct the final pattern service set.

**Recovering the control flow among the services identified as a pattern:** The control flow indicates the execution order of the services in a pattern and increases the value for reuse of the pattern in service composition. To recover the control flow of a pattern, we use the execution instances that contribute to the pattern. For each execution instance, we identify the execution flow among the services in the pattern. We combine execution flows from all execution instances to obtain the overall execution flow of the service composition pattern. We further infer the control flow structures (e.g., sequential structure, parallel structure) from the overall execution flow.
Table 4-1 Example of execution instances

<table>
<thead>
<tr>
<th>Execution Instance ID</th>
<th>Application ID</th>
<th>Services contained in the execution instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>{A, B, D, E, F, G, H, I, P}</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>{A, C, D, E, F, G, H, I, P}</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>{A, B, D, E, F, G, H, I, Q}</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>{A, C, D, F, E, G, H, I, Q}</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>{A, X, Y, Z, D}</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>{A, X, Y, Z, D}</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>{A, B, D, E, F, G, D, E, F, G, H, I, S}</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>{A, C, D, E, F, G, D, E, F, G, H, I, S}</td>
</tr>
</tbody>
</table>

4.2 Identification of Frequently Associated Services

In this section, we discuss our approach to identify a set of frequently associated services (i.e., a pattern service set) in detail. In our approach, we use Apriori [6] algorithm and develop a set of heuristic rules.

4.2.1 Extracting Execution Instances from Execution Logs

We analyze the execution logs to extract execution instances which may belong to different applications. Each execution instance contains a set of services. Using the execution instance id recorded in the events, we collect all the events triggered by the same execution instance into one group. We extract the identifiers of the services recorded in the events. As a result, we can obtain the services invoked in each execution instance. For example, Table 4-1 shows eight execution instances corresponding to the execution of four different service-oriented applications. Each letter represents a service. It is possible that a service is invoked more than once (e.g., in a loop in instances 7) in one execution instance. The multiple occurrences of a service should not affect our approach for identifying associated services because we do not consider the number of occurrences or the order of services to determine the associated services.
4.2.2 Identifying Pattern Service Set

We enumerate possible combinations of services to form candidate service sets of varying sizes. From the candidate service sets, we select the intermediate pattern service set. We construct the final pattern service set from the intermediate pattern service set. More specifically, we generate candidate service sets starting from size 1, and then increasing to size 2 and so forth. A service that appears in at least one execution instance becomes a candidate service set of size 1. We use “support” to measure the frequency of a candidate service set. A support value counts the number of execution instances that invoke all the services in the candidate service set. In the example shown in Table 4-1, the candidate service set \{D\}, is invoked by 8 execution instances. Therefore, the support of \{D\} is 8. The supports of all candidate service sets of size 1 are shown in Figure 4-2.a.

Given the candidate service sets of size n, we generate candidate service sets of size n + 1 in the following steps:

1) Collect candidate service sets that have the first n - 1 identical services and only one different service to form one group. In the example shown in Figure 4-2.b, the size of each...
candidate service set is 2 (i.e., \( n = 2 \)). We group the candidate service sets \{A, B\}, \{A, C\}, \{A, D\} and \{A, F\} together due to their identical first (i.e., \( n - 1 \)) service, A.

2) Expand candidate service sets of size \( n \) to size \( n + 1 \). In the new candidate service sets, the first \( n - 1 \) services are the same as the initial sets of size \( n \). The \( n \)-th and \((n + 1)\)-th services are pair-wise combinations of the \( n \)-th services of the initial sets of size \( n \). In the example shown in Figure 4-2.b, the 2nd services in Group A are B, C, D and F. The pair-wise combinations of the 2nd services are BC, BD, BF, CD, CF, and DF. These combinations become the 2nd and the 3rd services for the new candidate service sets. As a consequence, the new candidate service sets of size 3 are \{A, B, C\}, \{A, B, D\}, \{A, B, F\}, \{A, C, D\}, \{A, C, F\} and \{A, D, F\}.

3) Calculate the support of each new candidate service set. For example, the supports of the new candidate service sets of size 3 are listed in Figure 4-2.c.

4) Filter out infrequent candidate service sets. The number of possible candidate service sets can grow exponentially when the size of candidate service sets increases. To improve the efficiency in identifying the most frequent candidate service set, we filter out the infrequent candidate service sets of size \( n \) to reduce the number of candidate service sets of size \( n + 1 \). We define a threshold support shown in equation (4-1). The threshold support value is the average number of execution instances of a service-oriented application. If a candidate service set is more frequently executed than the average frequency, such a candidate service set is possible to be qualified as a pattern service set. We filter out the candidate service sets with a support value less than or equal to the threshold value. In the example shown in Table 4-1, the threshold support is 2 (i.e., \( 8 / 4 \)). Hence, a candidate service set with the support less than or equal to 2 (i.e., support <= 2) is filtered out as shown in Figure 4-2.
\[
\text{threshold support} = \frac{\# \text{ execution instances}}{\# \text{ service-oriented applications}} \quad (4-1)
\]

5) Repeatedly generate larger candidate service sets until no further candidate service sets can be achieved. We record the highest support value (e.g., H) achieved by candidate service sets of size 4. Any candidate service sets of size greater than 4 should have a support value no more than H. If no candidate service sets with size \( m (m > 4) \) can achieve the support value H, there is no need to further expand from candidate service sets of size \( m \) to candidate service sets of size \( m + 1 \) because further expansion will not achieve any candidate service sets with a support value H. Hence we stop the expansion. We rank all the candidate service sets with size greater or equal to 4 based on their supports. We select the one with the highest support (i.e., H) as the intermediate pattern service set \( S \). If more than one candidate service sets have the highest supports, we choose the one with the most services. In the example shown in Figure 4-2, the candidate service set \( \{A, D, E, F, G, H, I\} \), is selected as the intermediate pattern service set \( S \), which indicates that the services A, D, E, F, G, H and I, are frequently executed together.

6) Construct a larger pattern service set by merging the candidate service sets that contain the intermediate pattern service set \( S \) as a common subset and an additional different service. This step is used to handle the alternative control flow structure as shown in Figure 4-3. For example, we expand the set \( S \) and obtain 8 candidate service sets containing \( S \) as shown in Figure 4-2. We detect 2 sets from them to merge. Suppose we have the candidate service sets \( S_1 \) (i.e., \( S_1 = S \cup \{\text{service1}\} \)) and \( S_2 \) (i.e., \( S_2 = S \cup \{\text{service2}\} \)). We use \( S_{\text{merged}} \) (i.e., \( S_{\text{merged}} = S \cup \{\text{service1, service2}\} \)) as the final pattern service set if the following two conditions are met:

1) \( S_1 \) and \( S_2 \) are executed by the same applications.
(2) The support values of the four service sets (i.e., S, S\(_1\), S\(_2\) and S\(_{\text{merged}}\)) satisfy the condition shown in equation (4-2).

\[
\text{support}(S) = \text{support}(S_1) + \text{support}(S_2) - \text{support}(S_{\text{merged}})
\]  

(4-2)

Condition (2) indicates that service\(_1\) and service\(_2\) are likely in the two execution branches of an alternative control flow structure. When the alternative control flow structure is not within a loop, either service\(_1\) or service\(_2\) is executed in one execution instance. In this case, support\((S)\) is the sum of support\((S_1)\) and support\((S_2)\). When the alternative structure is within a loop, both service\(_1\) and service\(_2\) can be executed in two or more iterations in a single execution instance. Therefore, in equation (4-2) we need to reduce support\((S_{\text{merged}})\) from the sum of support\((S_1)\) and support\((S_2)\) to avoid double counting of such execution instances.

For example, \{A, D, E, F, G, H, I\} shown in Figure 4-2.y is the intermediate pattern service set S. We further check its expanded candidate service sets in Figure 4-2.z. We find that the sum of support values for \{A, D, E, F, G, H, I, B\} (i.e., S\(_1\)), and \{A, D, E, F, G, H, I, C\} (i.e., S\(_2\)), is 6. The support of S\(_{\text{merged}}\) (i.e., \{A, D, E, F, G, H, I, B, C\}), is 0. The support of S (i.e., \{A, D, E, F, G, H, I\}) is 6. Condition (2) is met. Moreover, S\(_1\) and S\(_2\) are both executed by Application 1,
Application 2 and Application 4, as indicated in Table 4-1. Condition (1) is satisfied. We use $S_{\text{merged}}$ (i.e., \{A, B, C, D, E, F, G, H, I\}) as the final pattern service set.

Using the aforementioned steps, we identify a final pattern service set. We also identify the execution instances that contribute to the pattern. Specifically, 1) if no merging is archived in step 6), we use S as the final pattern service set and the execution instances containing S are identified as the execution instances that contribute to the pattern. 2) If merging is achieved in step 6), the final pattern service set is $S_{\text{merged}}$ and the execution instances containing $S_1$, $S_2$ or $S_{\text{merged}}$ are identified as the execution instances that contribute to the pattern. We use such execution instances to infer the control flow of the service composition pattern in the next section.

### 4.3 Recovery of Control Flows of a Service Composition Pattern

In this section, we discuss our approach to recover the control flow of a service composition pattern. To identify the control flow, we analyze the execution instances that contribute to the pattern to identify their execution flows. An execution flow is a graph in which the nodes represent services and the edges denote service ordering relations. Examples of execution flows of execution instances can be found in Table 4-2. We combine the execution flows of the execution instances to obtain the overall execution flow of the pattern. We further infer the control flow structures from the overall execution flow.

We depict in Figure 4-4 the execution instances that contribute to the service composition pattern identified in last section. Instance 5 and 6 listed in Table 4-1 do not contribute to the pattern and they are not included in the figure. For each execution instance, we show the service invocation events in the figure. Such execution instances initially contain other services (as shown in Table 4-1) that do not belong to the final pattern service set. We disregard such services and do not depict them in Figure 4-4.
4.3.1 Identifying the Execution Flow of an Execution Instance

We identify three ordering relations among services in an execution flow: precedence relation, iterative relation and parallelism relation.

**Precedence Relation:** A precedence relation connects a service to its successor service and represents that the service executes immediately before the successor. Precedence relation is denoted by a directed edge from service A to the successor service B (i.e., $A \rightarrow B$) in the execution flow graph.

To ensure that the precedence relation is identified between two services A and B only when service B is executed immediately after service A, two conditions need to be satisfied: (1) the ENTRY B event occurs after the EXIT A event; (2) No services can start and complete its execution between the EXIT A event and the ENTRY B event.
To identify service ordering relations, we can examine each pair of services and check if the services meet the aforementioned conditions. However, it may be a time-consuming process. To ease the identification of the service ordering relations, we build an event graph to describe the ordering among events triggered by each execution instance. In an event graph, a node represents an ENTRY event or an EXIT event. For example, Figure 4-5 shows an event graph constructed using instance 7 from Figure 4-4. If a service is invoked more than once, it has more than one
ENTRY event and EXIT event. We treat these events as distinct events and create a node for each of them. A directed edge from node A to B represents that the event B immediately follows the event A in the execution instance.

A precedence relation is represented as a precedence edge that emanates from an EXIT event node and is incident on an ENTRY event node. In the example shown in Figure 4-4, the precedence relation between service A and service B (i.e., A→B) is represented by precedence edge 1. Generally, a precedence edge has a source service set and a destination service set. Each service in the source service set is the predecessor of each service in the destination service set. In other words, a precedence relation starts from a service in the source service set and leads to a service in the destination service set.

To identify the source service set of a precedence edge, we find all EXIT event nodes that can reach the precedence edge without going through an ENTRY event node. The services represented by such EXIT event nodes belong to the source service set. In the example shown in Figure 4-4, precedence edge 4 has a source service set that contains two services, namely services E and F. To identify the destination service set of a precedence edge, we locate all ENTRY event nodes that can be reached from the precedence edge without going through an EXIT event node. The services represented by such ENTRY event nodes are included in the destination service set. In the example shown in Figure 4-4, precedence edge 4 has a destination service set that contains service G.

In non-concurrent service execution, the source service set and the destination service set only have one service. Thus one precedence edge indicates exactly one precedence relation. In concurrent service execution, the source service set and destination service set may have more than one service. A precedence edge may indicate multiple precedence relations. In the example
shown in Figure 4-4, from precedence edge 4, we can infer two precedence relations, namely, \(E \rightarrow G\) and \(F \rightarrow G\).

Using the aforementioned approach, we construct an event graph for each execution instance shown in Figure 4-4. The identified precedence relations for each instance are shown using arrows in Table 4-2.

**Iterative Relation:** A loop exists in the execution flow graph when several services are repeated in the execution instance. For example, the execution flow of execution instance 7 contains a loop because some services are repeated. An iterative relation is a special type of precedence relation because it leads to a successor service that is the entry service of a loop in the execution flow. An iterative relation is initially identified as a precedence relation using the event graph. In the example shown in Figure 4-5, precedence edge 5 indicates a precedence relation \(G \rightarrow D\), which is actually an iterative relation. To distinguish an iterative relation from the precedence relations, we use depth-first traversal to traverse the execution flow starting from the first service in the execution instance. The traversal algorithm traverses the edges representing precedence relations (i.e., the edges with arrows in Table 4-2). During the traversal, if an edge leads to the ancestors of the current node, it indicates the entry service of a loop. We identify such an edge as an iterative relation. In Table 4-2, the execution flow of execution instance 7 contains an iterative relation from service \(G\) to service \(D\) (i.e., \(G \rightarrow D\)).

**Parallelism Relation:** A parallelism relation represents the concurrent execution of two services. A parallelism relation is denoted by an undirected edge between services \(A\) and \(B\) (i.e., \(A - B\)) in the execution flow graph. Parallelism relation is identified through precedence edge in the event graph. When there are multiple services in the source service set, a parallelism relation exists between each pair of the services. Similarly, parallelism relations can be identified
Table 4-2 Identified execution flow for each execution instance

<table>
<thead>
<tr>
<th>Instance id</th>
<th>Execution Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A → B → D → E → F → G → H → I</td>
</tr>
<tr>
<td>2</td>
<td>A → C → D → E → F → G → H → I</td>
</tr>
<tr>
<td>3</td>
<td>A → B → D → F → E → G → H → I</td>
</tr>
<tr>
<td>4</td>
<td>A → C → D → F → E → G → H → I</td>
</tr>
<tr>
<td>7</td>
<td>A → B → D → F → E → G → H → I</td>
</tr>
<tr>
<td>8</td>
<td>A → C → D → F → E → G → H → I</td>
</tr>
</tbody>
</table>

In the destination service set. In the example shown in Figure 4-5, we infer a parallelism relation (i.e., E → F) from the two services E and F, in the source service set of precedence edge 4.

Using this technique, we identify parallelism relations in each execution instance shown in Figure 4-4. The result listed in Table 4-2 shows that the parallelism relations is captured in execution instances 3, 7, and 8 because the services are concurrently executed as shown in Figure 4-4. This technique may not detect all parallel control flow structure because the services in a parallel control flow structure can sometimes execute in sequence. We address this by merging execution flows.

4.3.2 Merging Execution Flows

To describe all possible execution flows of a service composition pattern, we merge execution flows of all execution instances that contribute to the pattern. The overall execution flow contains all services in the pattern. For example, by merging all execution flows listed in Table 4-2, the overall execution flow shown in Figure 4-6 contains all services. However,
conflicts can occur during the merging process. We use the following rules to address the conflicts.

1) A pair of services has bi-directional precedence relations (e.g., a precedence relation from service A to service B and a precedence relation from service B to service A). This conflict occurs when the services in a parallel control flow structure are misidentified as having precedence relations because they are not concurrently executed. Instead of concurrent execution, they are found to be executing in sequence in random order. We replace such bidirectional precedence relations with a parallelism relation. In Table 4-2, two precedence relations will connect service E and service F (i.e., $E \rightarrow F$ and $F \rightarrow E$) after merging instances 2 and 4. We replace them with a parallelism relation between services E and F.

2) Two services are in both a precedence relation and a parallelism relation. The conflict occurs when the services in a parallel control flow structure are misidentified as having a precedence relation when they are not concurrently executed in some execution instance. The parallelism relation indicates that the services are in a parallel control flow structure. We remove the precedence relation. For example, services E and F are detected as having a parallelism
relation in instance 3. However, a precedence relation between service E and service F is identified in execution instance 1 because service E execute before service F in this particular execution instance. The parallelism relation prevails.

4.3.3 Inferring Control Flow Structures

To enable SOA developers to use the identified service composition patterns to compose new applications, we enhance the overall execution flows with control flow structures: sequential structure, loop structure, parallel structure and alternative structure. We use graph traversal algorithm to traverse all services in the overall execution flow by following edges representing precedence relations. The starting service is the service executes before any other services. The traversal begins at the starting service in the overall execution flow. During the traversal, we identify different types of control flow structures using edges representing precedence relations, iterative relations and parallelism relations.

**Sequential structure:** When a service emanates an edge denoting a precedence relation, this service is in a sequential structure with its successor service. In the example shown in Figure 4-6, services H and I are in a sequential structure.

**Loop structure:** Services in a loop structure can be repeated in an execution instance. We identify loop structure using iterative relation detected in previous subsection. An edge representing iterative relation emanates from the exit service and leads to the entry service of a loop structure. In the example shown in Figure 4-6, the iterative relation between services G and D (i.e., G→D) indicates that the services along the path from D to G are in a loop structure.

**Parallel structure:** In a parallel structure, multiple execution paths can be executed concurrently and invoked in any order. When two or more execution paths have common starting and ending services and parallelism relations exist between services in different paths, we identify
Figure 4-7 The identified service composition pattern with control flow structures

these paths as in a parallel structure. In the example shown in Figure 4-6, the execution path along the services D, E, G (i.e., D→E→G) and the execution path along the services D, F, G (i.e., D→F→G) are converted to a parallel structure because the services E and F have a parallelism relation. The services E and F can be executed concurrently in any order and both need to be completed before performing service G.

**Alternative structure:** Multiple possible execution paths are executed under a certain condition: only one path is executed at one time. When multiple execution paths branch out from one service, the execution paths can be either in a parallel structure or in an alternative structure. If these paths are not identified as in a parallel structure (i.e., no parallel relations exist between different paths), we identify it as in an alternative structure. In the example shown in Figure 4-6, the path along the services A, B, D (i.e., A→B→D) and the path along services the A, C, D (i.e., A→C→D) are recognized as an alternative structure.

Using this approach, the control flow structures can be inferred from in the overall execution flow. To ease the reuse of the service composition patterns, we manually represent the identified service composition patterns along with the control flow structures using BPEL. As an example,
control flow structures are identified from the overall execution flow shown in Figure 4-6 and the resulting service composition pattern is depicted in Figure 4-7.

4.4 Summary

In this chapter, we show our approach that identifies service composition patterns frequently used in practice from execution logs. The approach combines the association mining and process mining techniques. Specifically, our approach utilizes the Apriori algorithm with heuristics to identify a set of associated services. Then the control flow of the associated services is identified. Our approach improves the identification of parallel control flow structures.
Chapter 5

Case Studies

We conduct case studies to evaluate the effectiveness of our approach that describes various web resources using a unified description schema and identifies service composition patterns from execution logs.

5.1 Case Study for Evaluating the Unified Description Schema

In this section, we present the case study to evaluate the effectiveness of using the unified description schema for describing various web resources. The objectives of this case study are:

1. verify whether our approach can correctly describe web resources. The techniques for extracting information from SOAP-based Web Services, web forms and informational web pages have been discussed and evaluated in the existing literature (e.g., [117][79][21]). Hence, we focus on evaluating our technique that extracts information and describe HTTP-based APIs;
2. evaluate whether the unified description schema can help discover functionally similar web resources of different types.

5.1.1 Setup

We collect 37 web pages that contain the existing descriptions of HTTP-based APIs from different application domains, such as enterprise application domain, communication domain, and Internet application domain. The providers of the HTTP-based APIs are from different sectors, such as government agencies, IT companies, and individual developers. We avoid selecting many HTTP-based APIs from the same providers to ensure that the case study result is not skewed due to the particularity of a provider. We group the web pages into categories as specified in [78]. To
Table 5-1 Summary of the set of operations

<table>
<thead>
<tr>
<th></th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two HTTP-based API operations</td>
<td>6</td>
<td>4</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>Two SOAP-based Web operations</td>
<td>49</td>
<td>41</td>
<td>95</td>
<td>185</td>
</tr>
<tr>
<td>One HTTP-based API operation and one SOAP-based Web Service operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To evaluate the effectiveness of the unified description schema for discovering similar operations from different types of web resources (e.g., SOAP-based Web Services and HTTP-based APIs), we extract a set of operations from different types of web resources and represent them using the unified description schema. We combine every two operations in the set as a pair. To show whether similar operations from different types of web resources can be identified, all the pairs can be categorized into three groups based on the types of web resources of each pair as shown in Table 5-1. The first group consists of pairs with two HTTP-based API operations. The second group includes the pairs which consist of two SOAP-based Web Service operations. In the third group, a pair consists of one operation from an HTTP-based API, and another operation from a SOAP-based Web Service. We manually examine the existing descriptions of each pair of operations and determine whether such a pair of operations is similar or dissimilar.

5.1.2 Evaluation Criteria
We measure the effectiveness of our approach on identifying HTTP-based API operations using precision and recall. Precision is defined in equation (5-1). It measures if any irrelevant web page content is misidentified as the description of HTTP-based API operations. Recall is defined in equation (5-2). It evaluates whether our approach can correctly identify all web pages that contain HTTP-based API operations without omissions.

\[
\text{precision} = \frac{\# \text{correctly identified API descriptions}}{\# \text{identified API descriptions}} \tag{5-1}
\]

\[
\text{recall} = \frac{\# \text{correctly identified API descriptions}}{\text{total \# of API descriptions}} \tag{5-2}
\]

We also define the criteria for our approach to identify similar operations: If descriptive tags in the functionality descriptions, the input descriptions and the output descriptions of two operations are matched, the two operations are considered similar. Otherwise the two operations are considered not similar. We measure the effectiveness of discovering similar operations using accuracy and coverage. Accuracy is defined in equation (5-3), which measures the percentage of the correctly identified similar operations. Coverage is defined in equation (5-4). It measures whether our approach can find all pairs of similar operations.

\[
\text{accuracy} = \frac{\# \text{correctly identified pairs of similar operations}}{\# \text{identified pairs of similar operations}} \tag{5-3}
\]

\[
\text{coverage} = \frac{\# \text{correctly identified pairs of similar operations}}{\text{total \# of pairs of similar operations}} \tag{5-4}
\]

5.1.3 Analysis and Discussion of the Results

The results for identifying HTTP-based API operations from web pages are shown in Table 5-2. The high precisions show that our approach can correctly identify operations from
Table 5-2 Result of identifying operations for HTTP-based APIs from web pages

<table>
<thead>
<tr>
<th>Category</th>
<th>#Total Webpages</th>
<th>#Webpages containing operation descriptions</th>
<th>#Webpages from which operations are identified</th>
<th>#Webpages correctly Identified</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td>30</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Communication</td>
<td>30</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>100%</td>
<td>87.5%</td>
</tr>
<tr>
<td>Internet</td>
<td>30</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>100%</td>
<td>81.8%</td>
</tr>
<tr>
<td>Others</td>
<td>30</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>100%</td>
<td>88.9%</td>
</tr>
<tr>
<td>Total</td>
<td>120</td>
<td>37</td>
<td>33</td>
<td>33</td>
<td>100%</td>
<td>89.2%</td>
</tr>
</tbody>
</table>

HTTP-based APIs descriptions. Our approach does not misidentify any web pages that contain no HTTP-based API operations. The recalls show that our approach may fail to recognize web pages that contain HTTP-based API operations. An investigation shows that the HTML tag structures of operations are different in the existing descriptions of some HTTP-based APIs. Such HTTP-based API descriptions are hand-crafted web pages. There are no similar HTML tag structures in such HTTP-based API descriptions. However, most HTTP-based API descriptions are automatically generated using server-side scripting. Hence, this should not affect the usefulness of our approach.

Table 5-3 lists the result to show the ability of our approach to discover similar operations. The accuracy is very high, i.e., 100%. To qualify the similarity between operations, our approach matches descriptive tags of the functionality, the input and the output. This strict requirement helps avoid false positive. The recall shows that our approach misses some pairs of similar operations, especially in the cases involving HTTP-based APIs due to the difficulty in accurately extracting descriptive tags from the existing descriptions written in natural language.

5.1.4 Limitations

Because the process to download web resources from the Internet is manual in our case study, the case study is conducted on a small experiment set. This affects the generalization of
Table 5-3 Result of identifying pairs of similar operations

<table>
<thead>
<tr>
<th></th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two HTTP-based</td>
<td>Two SOAP-based Web operations</td>
<td>One HTTP-based API operation and one SOAP-based Web Service operation</td>
<td>Total</td>
<td>100%</td>
</tr>
<tr>
<td>API operations</td>
<td>66.6%</td>
<td>100%</td>
<td>86.7%</td>
<td>84.0%</td>
</tr>
</tbody>
</table>

In the future, we plan to automate the process to discover various resources from Internet in order to obtain a large experiment set.

5.2 Case Study for Evaluating Identifying Service Composition Patterns

This section describes the case study for evaluating the approach that identifies service composition patterns from execution logs. The objectives of this case study are: 1) verify if our approach can accurately recover the control flow of a service-oriented application from execution logs; and 2) validate if our approach can accurately identify service composition patterns.

5.2.1 Setup

In the case study, 74 service-oriented applications are studied. The applications are based on a set of reference business processes provided by a business process management architect from our industrial collaborator [4]. An overview of these service-oriented applications is listed in Table 5-4. The applications specify activities and procedures used in various sectors. For example, the financial management applications manage financial policies, procedures, accounts and assets. Since we do not have the resources to implement the entire application, we use stub services without internal implementations to simulate the applications. The simulated applications can generate the execution logs that correctly reflect the execution of applications in production environments. The execution logs are collected from the testing environment when testing the 74...
Table 5-4 The service-oriented applications used in the case study

<table>
<thead>
<tr>
<th>Industry</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financial Management</td>
<td>10</td>
<td>Manage financial policies, procedures, accounts and assets.</td>
</tr>
<tr>
<td>Human Capital Management</td>
<td>11</td>
<td>Manage recruiting new employees and administering existing employees.</td>
</tr>
<tr>
<td>Supply Chain Management</td>
<td>10</td>
<td>Manage product, market research, and procurement of materials and services.</td>
</tr>
<tr>
<td>Banking</td>
<td>11</td>
<td>Open bank accounts.</td>
</tr>
<tr>
<td>Insurance</td>
<td>10</td>
<td>Handle customers’ applications for insurance.</td>
</tr>
<tr>
<td>Government</td>
<td>12</td>
<td>Administer employer/unemployment services.</td>
</tr>
<tr>
<td>Retail</td>
<td>10</td>
<td>Manage inventory, product, order, and price.</td>
</tr>
</tbody>
</table>

service-oriented applications. A WebSphere Process Server installation is used as the runtime execution environment to host the applications.

In the process of testing, SoapUI [96], a standard SOAP-based invocation tool is used to automatically execute the service-oriented applications for a large number of times. The execution logs are recorded through the Common Event Infrastructure (CEI) built in the IBM WebSphere Process Server. To verify if our approach can correctly identify service composition patterns from the 74 service-oriented applications, we use a testing setting with varying numbers of execution instances for the applications and apply our approach.

5.2.2 Evaluation Criteria

To measure the correctness of the identified service composition patterns, we measure the correctness of the identified pattern service sets and the correctness of the identified pattern control flows.

We use two metrics to measure the correctness of an identified pattern service set: relevancy and coverage. Relevancy is defined in equation (5-5). It measures whether our approach can
correctly identify the execution instances that contribute to a pattern service set. Specifically, it measures whether the identified related execution instances are relevant to the identified pattern. We manually verify if our approach correctly filters out the execution instances without patterns. Since we can access the documented business process for each execution instance, we verify whether an execution instance contributes to a pattern by comparing the business process with the pattern. The relevancy should be high if our approach correctly identifies service composition patterns and the related execution instances.

\[
\text{relevancy} = \frac{\# \text{ correctly identified instances}}{\# \text{ identified execution instances}} \quad (5-5)
\]

Coverage is defined in equation (5-6). It calculates whether the identified pattern is widely adopted among the applications. To judge the correctness of our approach, the coverage is used in conjunction with knowledge about functionality of the applications which produce the execution logs. If we know that the execution logs are collected from a set of applications that fulfill similar functionality, we could expect that the coverage be high. In contrast, if the execution logs are collected from a set of applications with diverse functionalities, we expect the coverage to be low.

\[
\text{coverage} = \frac{\# \text{ instances that contribute to the pattern}}{\# \text{ total execution instances}} \quad (5-6)
\]

We use accuracy to measure the correctness of the control flow of an identified service composition pattern. It is defined in equation (5-7). Among all the execution instances that contribute to the pattern, we count the correct execution instances. An execution instance is correct if the control flow of its underlying application contains the control flow of the identified pattern.

71
5.2.3 Analysis and Discussion of Results

We recover the control flow of each service-oriented application from the execution logs that contain 10,000 execution instances for each application. The accuracy of the recovered control flows is verified by comparing with the documented business processes for each application. Specifically, the nodes and edges of the recovered control flow are compared with the documented control flow.

Our approach can correctly handle most special cases, such as self-loop (i.e., loop containing only one service). As a result, our approach accurately identifies the control flows of 73 out of 74 service-oriented applications listed in Table 5-4. However, our approach fails to handle a special case in which an alternative structure has two branches with identical services in different order. Specifically, in the first branch, the service A executes before service B. In the other branch, the service B executes before service A. As a consequence, A and B appear in the execution log in random order. Our approach misidentifies A and B as in a parallel structure since we use the random execution order as one criterion to identify parallelism relations.

The results for identifying service composition patterns are shown in Table 5-5. The high relevancies show that our approach can accurately identify pattern service sets. The coverage of pattern service sets is moderate because in our case study the execution logs is collected from a combination of similar and different applications. Therefore, the patterns are not predominately used in all applications. The accuracy of pattern control flow is high but not 100% accurate due to the cases where two or more services are arranged slightly differently between similar
Table 5-5 Result of identifying service composition patterns

<table>
<thead>
<tr>
<th>#</th>
<th>Domain of pattern</th>
<th># Total execution instances</th>
<th># Execution instances identified to be related to the pattern</th>
<th>Accuracy of pattern resource set</th>
<th>Accuracy of control flow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>%Relevancy</td>
<td>%Coverage</td>
</tr>
<tr>
<td>1</td>
<td>Financial Management</td>
<td>15K</td>
<td>10,285</td>
<td>100%</td>
<td>68.6%</td>
</tr>
<tr>
<td>2</td>
<td>Human Capital Management</td>
<td>16K</td>
<td>11,110</td>
<td>100%</td>
<td>69.4%</td>
</tr>
<tr>
<td>3</td>
<td>Supply Chain Management</td>
<td>15K</td>
<td>9,778</td>
<td>100%</td>
<td>65.2%</td>
</tr>
<tr>
<td>4</td>
<td>Banking</td>
<td>16K</td>
<td>10,413</td>
<td>100%</td>
<td>65.0%</td>
</tr>
<tr>
<td>5</td>
<td>Insurance</td>
<td>15K</td>
<td>9,559</td>
<td>100%</td>
<td>63.7%</td>
</tr>
<tr>
<td>6</td>
<td>Government</td>
<td>17K</td>
<td>12,423</td>
<td>100%</td>
<td>73.0%</td>
</tr>
<tr>
<td>7</td>
<td>Retail</td>
<td>15K</td>
<td>10,230</td>
<td>100%</td>
<td>68.2%</td>
</tr>
</tbody>
</table>

applications, although they share the pattern service set. Suppose that two services (i.e., A and B) are in a parallel structure in some applications. However, in other applications, A and B are arranged as sequence (i.e., A→B). We cannot detect the two conflicting situations and we treat the two services as in a parallel structure. It is a limitation that we cannot detect the applications with sequential relation between A and B. However, a sequential relation is essentially a special case in the parallel structure of the two services.

Table 5-6 gives a summary of the service composition patterns identified by our approach. For each service composition pattern, its domain of application, its diagram representation and a short description is provided.

5.2.4 Limitations

Due to the difficulties to collect real-world execution logs from production environments, the execution logs used in the case study are collected from a testing environment. The simulated applications in the testing environment may not fully reflect the characteristics of the real-world
<table>
<thead>
<tr>
<th>#</th>
<th>Pattern domain</th>
<th>Pattern diagram</th>
<th>Pattern Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Financial Management</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that provide fixed asset data for tax, regulatory or statutory purposes.</td>
</tr>
<tr>
<td>2</td>
<td>Human Capital Management</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications which handle staffing process.</td>
</tr>
<tr>
<td>3</td>
<td>Supply Chain Management</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that conduct portfolio management of all the products and services offered by the organization to its customers.</td>
</tr>
<tr>
<td>4</td>
<td>Banking</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that process loan applications.</td>
</tr>
<tr>
<td>5</td>
<td>Insurance</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that check and complete customer applications.</td>
</tr>
<tr>
<td>6</td>
<td>Government</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that process unemployment service claims.</td>
</tr>
<tr>
<td>7</td>
<td>Retail</td>
<td><img src="#" alt="Diagram" /></td>
<td>A pattern shared by applications that help retail store to promote products.</td>
</tr>
</tbody>
</table>
applications in the production environments. In the future, we plan to examine the various possibilities to obtain real-world execution logs and apply our approach.

Our approach of identifying service composition patterns achieves 100% in the relevancy metric between the identified patterns and its related execution instances. This metric only demonstrate that the identified pattern service set is frequently used in the related execution instances. However, the usefulness of the identified patterns is not evaluated. In the future, we should improve the evaluation of identified patterns by using external evaluator without knowledge about the approach.

5.3 Summary

In this chapter, we present our case studies that evaluate the effectiveness of our approach that describes various web resources using a unified description schema and identifies service composition patterns from execution logs. We also analyze the result and discuss the limitations of the case studies. The case study results show that our approach can effectively extract information from HTTP-based API. The unified description schema is useful in discovering similar services although it cannot discover all similar HTTP-based APIs. Finally, the result of identifying patterns shows that we can correctly identify composition patterns from execution logs.
Chapter 6

Conclusion and Future Work

Service composition patterns exist in similar applications of the same application domains. Identifying and reusing the service composition patterns can facilitate the composition of new applications, improve existing applications and optimize maintenance process of services. We present an approach that uniformly describes different types of web resources and automatically identifies service composition patterns from execution logs produced by service-oriented applications. In this chapter, we summarize the contributions of the thesis and discuss future work.

6.1 Thesis Contributions

A unified description schema for describing heterogeneous web resources: It is critical to discover and locate web resources with similar functionality to facilitate service composition and replacement. To identify web resources with similar functionality, an SOA professional often manually review the descriptions of web resources in various formats to examine the functionality. This process cannot be automated due to the absence of a standard format for describing various web resources. Although a number of existing approaches handle the automatic discovery of similar SOAP-based Web Services, such approaches are limited to discover a single type of web resources. To assist the automatic discovery of various web resources with similar functionality, we propose a schema to uniformly describe different types of web resources. The unified representation provides a better chance to discover web resources with similar functionality than limiting the service discovery within the web resources of a single type.
**Automatic techniques to extract and describe HTTP-based APIs:** To deploy the proposed unified description schema, it is essential to efficiently generate the description record for each web resource. It is a challenging task to generate a description record from the existing description of an HTTP-based API due to the lack of a machine-readable standard format to describe HTTP-based APIs. Our technique automates the identification of operations using the similarity among HTML tag structures. Based on the identified operations, we automatically construct the description records for the web resources.

**An automatic approach to identify service composition patterns from execution logs:** Although a number of existing approaches analyze the execution logs of service-oriented applications to mine business processes, the existing research focuses on recovering the control flow of a single business process without extracting patterns shared among multiple applications. We identify frequently executed patterns used by multiple service-oriented applications. To facilitate the reuse of a service composition pattern as an independent and ready-to-use component, we further infer the control flow of a pattern.

**A technique to improve the efficiency for identifying control flow structures of service composition patterns:** Existing approaches detect parallelism relations between services if the services appear in random orders in different execution logs. Such approaches require a large number of execution logs to reflect all possible execution orders among the services. Furthermore, not all runtime execution environments support the execution of parallel services in random order. Our approach extends the aforementioned technique and uses the event graph of the service invocation events to detect concurrency. In the cases where parallel services are executed concurrently, our approach can identify parallelism using a single execution of a
service-oriented application without the assumption of the random execution order among parallel services.

6.2 Limitations and Future Work

The work in this thesis has several limitations. In this section, we describe the limitations and discuss the possible directions of future work to extend our approach for addressing the limitations.

Examining the accessibility of execution logs in production environments. We rely on the availability of execution logs to mine service composition patterns. It can be challenging to collect a large amount of execution logs produced by multiple service-oriented applications in a production environment. Our approach assumes the users have the access to the execution logs of multiple applications. However, the access to execution logs may be restricted due to confidentiality consideration and other concerns. To further understand and extend the applicability of our approach, we plan to conduct a thorough survey on the availability of execution logs to different organizations. Because the “Platform as a Service” providers accommodate a large number of service-oriented applications, we are particularly interested in the related issues for the “Platform as a Service” providers to access and utilize the execution logs produced by third-party service-oriented applications. Our goal is to establish a mechanism to share execution logs without compromising the confidentiality of information processed by individual service-oriented applications.

Automating the selection of thresholds. The threshold for the minimum number of services required in a service composition pattern can vary in different use cases. Our current approach does not automatically detect the optimal threshold. When applying our approach to a new collection of execution logs, the threshold needs to be fine-tuned to obtain suitable service
composition patterns. In the future, we plan to devise a technique to automatically or semi-
automatically adjust the threshold of the minimum number of services. One possible solution is to
use metrics (e.g., average number of services in the underlying applications of the execution logs)
to calculate an initial threshold, and gradually adjust it according to a user’s feedback.

**Incorporating more types of web resources into the unified description schema.** Our
unified description schema may not be able to describe all types of web resources available in the
Internet. In this thesis we investigate four types of web resources. It is expected that the schema
need to be revised and updated to accommodate new types of web resources. We plan to study
more types of web resources (e.g., RSS) to extend the schema to accommodate various types of
web resources.

**Integrating with web crawlers.** To generate the description records for various web
resources, our current approach require manually download the existing web resource
descriptions from the web. To assist automatic discovery of various web resources, our approach
for describing various web resources can be combined with the web crawlers, e.g., WebSPHINX
[111]. The web crawlers can automatically browse and fetch web pages. Our approach can be
used to identify HTTP-based APIs, SOAP-based Web Services and other web resources and
further describe them using the unified description schema.

**Optimizing the detection of similar HTML tag structures.** To detect the similar HTML
tag structures in a single-page description of an HTTP-based API, our technique require
traversing the entire DOM tree of the web page. This can be time-consuming given a large
number of web pages to identify. In the future, we plan to investigate the probabilities for
different type of HTML tags to contain similar HTML tags structures. We can develop pruning
techniques to reduce the amount of required traversal. Furthermore, we can optimize the order to search the DOM tree in order to locate similar structure more efficiently.

**Exploring the usage of identified control flow of service-oriented applications.** In addition to facilitating pattern reuse, the techniques of identifying control flow may be used for other purpose. Specifically, we can recover the runtime control flow of the applications and compare with the documented ones in order help maintain the applications.

### 6.3 Summary

In this chapter, we summarize the contributions of this thesis and envision possible directions of future work for addressing the limitations of our research. We propose a unified description schema for describing heterogeneous web resources and develop automatic techniques to extract and describe HTTP-based APIs. Furthermore, we propose an automatic approach to identify service composition patterns from execution logs. Our technique improves the efficiency for identifying control flow structures of service composition patterns
Bibliography


16. J. Cardoso, “Workflow Quality of Service Management Using Data Mining Techniques”,
Proceedings of 3rd International IEEE Conference on Intelligent Systems, London,
September 2006, pp. 479-482.


Language Version 2.0”, W3C Recommendation, 2007

19. E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. “Web Services Description
Language (WSDL) 1.1”, http://www.w3.org/TR/wsdl, last accessed on September 9,
2010

Near-Miss Clones”, Proceedings of the Conference of the Centre For Advanced Studies
on Collaborative Research, Markham, Ontario, Canada, October 4-7, 2004, pp. 1-12.

21. T. C. Craven, “HTML Tags as Extraction Cues for Web Page Description Construction”,
Informing Science: International Journal of an Emerging Transdiscipline, Vol. 6, 2003,
pp. 1-12.


23. DAML-S Coalition, “DAML-S: Web Service Description for the Semantic Web”,


27. Document Object Model (DOM), http://www.w3.org/DOM/, last accessed on September 9, 2010


34. Flickr API, http://www.flickr.com/services/api/, last accessed on September 9, 2010


55. JOpera, http://www.jopera.org/, last accessed on September 9, 2010


http://ca.com/files/IndustryAnalystReports/technology_audit_ca_wily_application.pdf, last accessed on September 9, 2010


73. National Digital Library of Theses and Dissertations
http://www.dlib.org/dlib/september96/theses/09fox.html, last accessed on September 9, 2010


77. PHP, http://php.net/, last accessed on September 9, 2010


97. SOAP Version 1.2, http://www.w3.org/TR/soap12-part1/, last accessed on September 9, 2010

98. SourceLabs’ Byron Sebastian joins Heroku as CEO


99. Stanford Log-Linear Part-Of-Speech Tagger,

http://nlp.stanford.edu/software/tagger.shtml, last accessed on September 9, 2010


104. The Stanford-IBM Manager of Multiple Information Sources.
http://infolab.stanford.edu/tsimmis/, last accessed on September 9, 2010


110. Web Application Description Language (WADL)
http://labs.oracle.com/techrep/2006/abstract-153.html, last accessed on September 9, 2010

111. WebSPHINX: http://www.cs.cmu.edu/~rcm/websphinx/, last accessed on September 9, 2010


