IOWA STATE UNIVERSITY
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2007

Interactive and verifiable web services composition, specification reformulation and substitution

Jyotishman Pathak
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

 Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
Pathak, Jyotishman, "Interactive and verifiable web services composition, specification reformulation and substitution" (2007). *Retrospective Theses and Dissertations*. 15586.
https://lib.dr.iastate.edu/rtd/15586

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Interactive and verifiable web services composition, specification reformulation and substitution

by

Jyotishman Pathak

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Vasant Honavar, Major Professor
Samik Basu
Drena Dobbs
Shashi Gadia
Robyn Lutz
James McCalley

Iowa State University

Ames, Iowa

2007

Copyright © Jyotishman Pathak, 2007. All rights reserved.

UMI Number: 3289388

UMI Microform 3289388

Copyright 2008 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

DEDICATION

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere gratitude to my advisor Dr. Vasant Honavar for his guidance in this Ph.D. thesis work. He is, and will always be, a great source of inspiration to me. The constructive and insightful discussions that I have had with him over the past few years, has motivated me to think outside the box and develop new ideas for this work. I have always benefited from his constant encouragement, enthusiasm and zeal for research.

I would also like to specially thank Dr. Samik Basu with whom I closely collaborated for this research. Countless, and often un-scheduled, meetings with him have resulted in very important and stimulating ideas that were instrumental for my thesis. I am privileged to have him as a mentor and collaborator. Thanks also goes to Dr. Robyn Lutz who gave me insights in software engineering requirements and with whom I collaborated on many accounts, to Dr. Drena Dobbs for very important suggestions in developing the protein-protein interface database, and to Dr. James McCalley for introducing me to condition monitoring of power transformers. I also extend my warm thanks to Dr. Shashi Gadia for his support and agreeing to be a member of my committee.

I have also received a lot of support from my A.I. Research Lab group members and I am grateful for all their efforts. Thanks to Dr. Doina Caragea, Dr. Jun Zhang, Dr. Jie Bao, Neeraj Koul, and Feihong Wu—it has been a real pleasure working with all of you. I am also thankful to Dr. Yong Jiang, Yuan Li, Hieu Pham, Mohammed Alabsi, Rakesh Setty, Mahantesh Hosamani and Melissa Yahya for their help and collaboration on many occasions. Thanks also to Lanette Woodard and Linda Dutton for their assistance and guidance in many matters.

On the personal front, I am eternally grateful to my parents, Adhar and Sangeeta Pathak, for their endless love and support. I would also like to thank my brother, Ujjal Pathak, for being a very caring and understanding friend. A very special thanks also goes to my partner, Divya Ranganathan, who has been by my side in all aspects of my life, and whose unrelenting love, encouragement and perseverance has played a significant role in the completion of my thesis. Last but not the least, I thank my friends/elders Sudip and Katyayani Seal, Bhaskar Choudhury, Sonia Lall, Satyam Bhuyan, Porismita Borah, Ankit Saran, Haseena Ahmed, Arun and Kobita Barua, Madan and Jahnabimala Bhattacharya, Anantharaman Kalyanaraman, Kirthi Rajagopalan, Flavian and Athena Vasile, and others I may have forgotten, who made my stay in Ames a memorable one.

Finally, I am grateful to the National Science Foundation (grants IIS 0219699, 0540293, 0702758), the Power Systems Engineering Research Center, and the Center for Computational Intelligence Learning & Discovery at Iowa State University for funding this research work.

ABSTRACT

Recent advances in networks, information and computation grids, and WWW have resulted in the proliferation of physically distributed and autonomously developed software components and services. These developments allow us to rapidly build new value-added applications from existing ones in various domains such as e-Science, e-Business, and e-Government. Towards this end, this dissertation develops solutions for the following problems related to Web services and Service-Oriented Architectures:

1. **Web Service Composition:** The ability to compose complex Web services from a multitude of available component services is one of the most important problems in service-oriented computing paradigm. In this dissertation, we propose a new framework for modeling complex Web services based on the techniques of abstraction, composition and reformulation. The approach allows service developers to specify an abstract and possibly incomplete specification of the composite (goal) service. This specification is used to select a set of suitable component services such that their composition realizes the desired goal. In the event that such a composition is unrealizable, the cause for the failure of composition is determined and is communicated to the developer thereby enabling further reformulation of the goal specification. This process can be iterated until a feasible composition is identified or the developer decides to abort.

2. **Web Service Specification Reformulation:** In practice, often times the composite service specification provided by the service developers result in the failure of composition. Typically, handling such failure requires the developer to analyze the cause(s) of the failure and obtain an alternate composition specification that can be realized from the available services. To assist developers in such situations, we describe a technique

which given the specification of a desired composite service with a certain functional behavior, automatically identifies alternate specifications with the same functional behavior. At its core, our technique relies on analyzing data and control dependencies of the composite service and generating alternate specifications on-the-fly without violating the dependencies. We present a novel data structure to record these dependencies, and devise algorithms for populating the data structure and for obtaining the alternatives.

3. **Web Service Substitution:** The assembly of a composite service that satisfy a desired set of requirements is only the first step. Ensuring that the composite service, once assembled, can be successfully deployed presents additional challenges that need to be addressed. In particular, it is possible that one or more of the component services participating in a composition might become unavailable during deployment. Such circumstances warrant the unavailable service to be substituted by another without violating the functional and behavioral properties of the composition. To address this requirement, we introduce the notion of context-specific substitutability in Web services, where context refers to the overall functionality of the composition that is required to be maintained after replacement of its constituents. Using the context information, we investigate two variants of the substitution problem, namely environment-independent and environment-dependent, where environment refers to the constituents of a composition and show how the substitutability criteria can be relaxed within this model.

The work described above contributed to the design and implementation of MoSCoE—an open-source platform for modeling and executing complex Web services (http://www.moscoe.org).

CHAPTER 1. INTRODUCTION

This chapter provides an introduction to the main topics and background of the thesis. A brief description of our approaches and an outline of the structure of the thesis is also provided.

1.1 Service-Oriented Computing

In today's world, the ability to quickly deliver new applications is increasingly becoming imperative for business organizations. They face rapidly changing market conditions, pressure from competition and new regulations that demand compliance which drive the need for the IT infrastructure to respond aptly in support of new business models and requirements. However, since most of the enterprise and legacy applications were not designed to enable rapid adoption and adaptation of functionality, they become a bottleneck in the already intricate IT landscape of an organization for efficient and effective application development.

Service-Orientation aims to provide the underlying machinery that can potentially overcome this drawback and realize such an "on-demand" IT environment by essentially supporting three important requirements [114]: integration, virtualization and management. Integration in this context refers to the ability to seamlessly combine multiple existing, and often heterogeneous, applications and resources across organizations. Virtualization, on the other hand, is the ability to provide an uniform and consolidated access to the applications irrespective of programming language used for its implementation, the server hosting the application, the operating system on which it is running, and so on. And finally, management is the ability to provide a logical architecture for managing computing resources using managed objects and their relationships. These requirements are enabled by adopting a programming model called *Service-Oriented Computing* (SOC) [142, 144, 145] which utilizes *services* as the build-

ing blocks for fast, low-cost and efficient (distributed) application development. Services are autonomous, self-describing and platform-agnostic computational entities that can perform various functions ranging from responding to simple requests to complex enterprise processes. They allow organizations to expose multiple applications, using standard interface description languages, which can be accessed and invoked programatically over the network (Internet or intranet) using widely adopted protocols and languages. Furthermore, SOC enables the services to be published in repositories and dynamically discovered and assembled for building massively distributed, interoperable and evolvable systems. Typically, they are built in a way without preconceiving the context in which they will be used. Consequently, the provider and consumer of a particular service are loosely coupled, and often inter-organizational.

In practice, there are two basic types of services: *atomic* and *composite*. Atomic services are single network-accessible applications that can be invoked by sending a message. Upon invocation, the service performs its task and (in some cases) produces a response to the invoker. Thus, there is no ongoing interaction between the service requestor and the service. Examples of services that would fall in this category would include the ones which given the zip code of a city will output the current temperature or given the symbol of a company will provide the current stock quote. The composite services, on the other hand, comprise of multiple atomic (and/or other composite) services and require an extended interaction between the service requestor and the set of services providing a particular functionality. Many e-Commerce sites such as Amazon.com, eBay.com etc. fall into this category. For example, in order to purchase a digital camera at eBay, a user has to first search for it using different criteria, possibly read the reviews and analyze user's ratings, search for relevant accessories, and then finally provide payment and shipping information to complete the purchase.

The example services cited above are commonly referred as "services over the Web" and are mainly developed for human consumption. However, even though there is an abundance of Web-based applications primarily targeted for humans, services are also meant to be used by other applications (and possibly by other services) directly, and not only by humans. In other words, the goal of SOC is to enable pure service-to-service interactions as opposed to only

service-to-human interactions. Nevertheless, it is widely acknowledged that to enable service-to-service interactions there should be a provision for services to automatically find, select and communicate with other services, which in turn requires the services to explicitly specify their "semantics" unambiguously. The semantics of a service should capture various aspects including functional properties, behavioral (control/data-flow) properties, transactional properties, quality of service properties, and so on.

From the above discussion, it is stems that SOC model poses many challenges and research questions in different aspects including service specification, discovery, composition, execution, and management. In this dissertation, we focus mainly on the fundamental concepts pertaining *automatic composition of services*.

1.2 Motivation: Web Service Composition

One of the most widely adopted ways for realizing the SOC model into an architecture is called a *Service-Oriented Architecture* (SOA) [72, 73], which in essence is a logical way for developing distributed software system by providing services to end-user applications or to other services via published and discoverable interfaces. OASIS (the Organization for the Advancement of Structured Information Standards) defines SOA as follows:

Definition 1 (Service-Oriented Architecture [1]) *A Service-Oriented Architecture (SOA) is paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.*

In general, an SOA comprises of three different players (Figure 1.1): (*i*) the *service provider* is an entity which provides the service; (*ii*) the *service requestor* is an entity which searches for and invokes a particular in order to fulfill its goals, and finally (*iii*) a *discovery agency* is an entity which acts as a repository or a directory of services. When a provider wants to make available a particular service, it publishes information about how to invoke the service

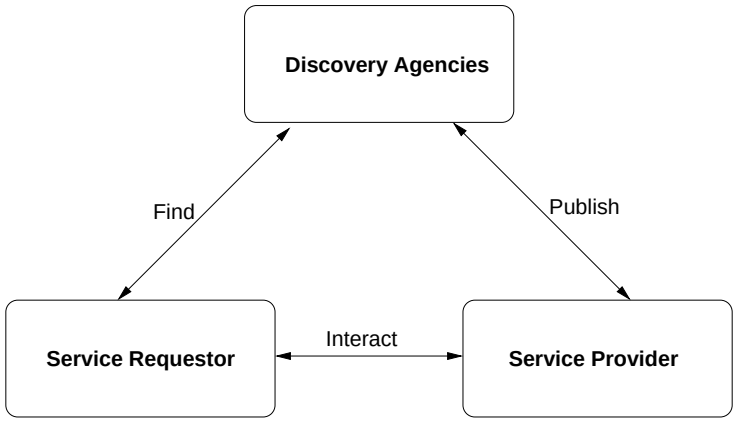


Figure 1.1 Diagram of a Service-Oriented Architecture

(e.g., URL, protocols) along with the interface description of the service itself in a discovery agency. A client who wants to use a particular service, searches for it in the repository and then interacts with the service provider directly for service invocation. Even though this framework is simplistic, it raises very interesting research issues with respect to specification (e.g., how to specify the syntax and semantics of a service unambiguously?), discovery (e.g., how to find the best service suitable for a particular job?), composition (e.g., how to assemble multiple atomic services for a particular job?), execution (e.g., how to execute services securely), and so on. As mentioned earlier, the main focus of our work is to develop techniques for automatic service composition.

Note that a SOA is not tied to a specific implementation technology. It may be implemented using a wide-range of technologies including Web Services [10], RPC [41], DCOM [77], REST [75] or CORBA [42]. In our work, we only consider Web Services-based SOA, which we describe in the following.

1.2.1 What are Web Services?

According to the W3C (World Wide Web Consortium) Web Services Architecture [44], Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. More formally, the architecture defines Web services as follows:

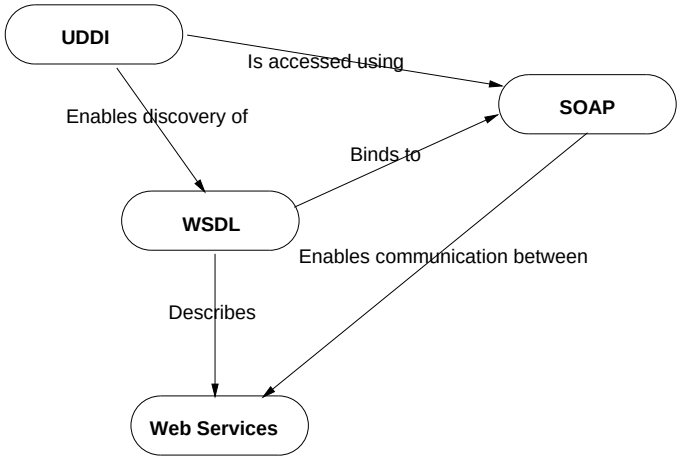


Figure 1.2 Relationship Between Standard Web Service Specifications [72]

Definition 2 (Web Service [44]) *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [62]). Other systems interact with the Web service in a manner prescribed by its description using SOAP [90] messages, typically conveyed using HTTP with an XML [51] serialization in conjunction with other Web-related standards.*

This definition, represented pictorially in Figure 1.2, outlines two fundamental requirements of Web services:

- they communicate by exchanging data formatted as XML documents using SOAP over Internet protocols (such as HTTP);
- they provide a service description that, at minimum, consists of a WSDL document.

where, SOAP provides a standard, extensible, and composable framework for packaging and exchanging XML messages and WSDL describes Web services starting with the messages that are exchanged between the service requester and provider.[1] Thus, such a description describes a service in terms of functionalities that it exports which can be invoked by input/output messages. However, in many cases a simple description of inputs and outputs is not enough

[1] We provide more details on these technologies in Chapter 2

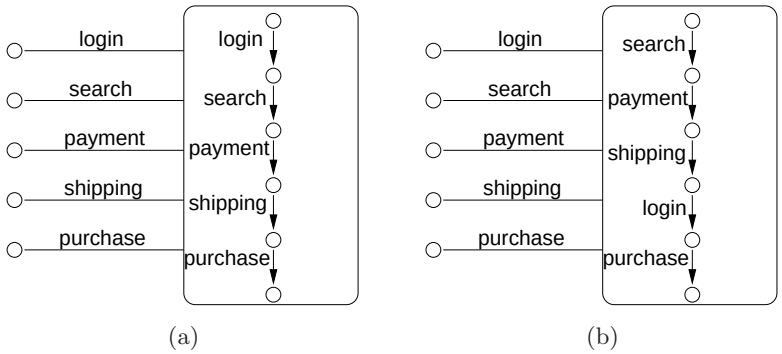


Figure 1.3 Input/Output & Behavioral Descriptions of (a) `e-Com` and (b) `e-Com'` services

because it does not represent the actual "behavior" of a service [26, 29, 101]. For instance, consider our previous example (Section 1.2) where a user wants to interact with an e-Commerce service to purchase items. The input/output interface of such a service, `e-Com`, can be represented in Figure 1.2.1(a) which essentially allows a client to search for items of interest and purchase them by providing payment and shipping information.[2] In addition, it requires the client to login (i.e., authenticate) first before using the search functionality. A similar service `e-Com'`, with same input/output interface, is shown in Figure 1.2.1(b) which provides the same functionality, but requires the client to login only when it is ready to make a purchase. That is, the behavioral model of `e-Com'` is different from that of `e-Com`, even though they have the same input/output interface, which in turn implies that the behavioral model of the clients that interact with these two services will also be different. However, such differences cannot be captured by description languages such as WSDL. This has warranted the development of much more expressive languages such as WS-BPEL [16] and WS-CDL [107] which allow to explicitly represent the services in terms of exact sequence of operations that they support. In the Web services domain, such descriptions are commonly referred to as *conversations* and can be represented using transition systems [100]. We provide more details on this topic in Chapter 3.

[2]This example and the figure has been adopted from [29].

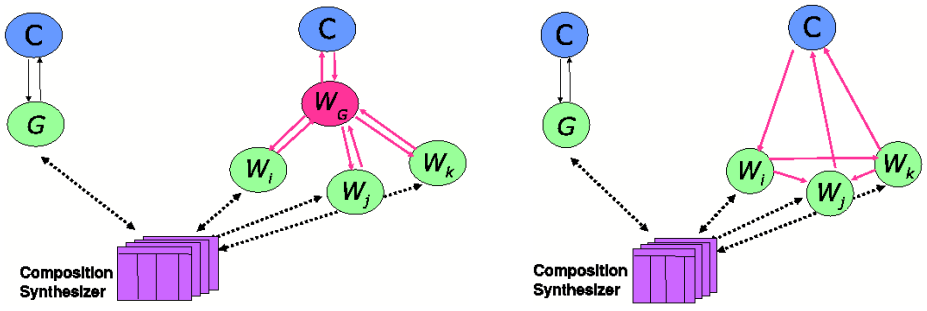


Figure 1.4 Two different types of composition [32]: (a) Orchestration-based (b) Choreography-based

1.2.2 What is Web Service Composition?

As outlined above, research in Web services, and SOC in general, span across multiple areas. One of the areas that has received a lot of attention from both academia and industry in the recent past is *Web Service Composition* which is the ability to aggregate multiple services into a single composite service that would provide a certain functionality, which otherwise cannot be provided by a single service. The motivation for service composition is based on the requirement for developing "value-added" services and applications by selecting and integrate pre-existing services. Such an approach has tremendous benefits in terms of reducing the cost and effort for building newer services from scratch, thereby promoting rapid application development. Additionally, the resulting composite services may be used as basic services in further service compositions.

In practice, there are two different (and competing) notions of modeling Web service compositions [32]: orchestration (Figure 1.4(a)) and choreography (Figure 1.4(b)). Orchestration describes how multiple services can interact by exchanging messages including the business logic and execution order of the interactions from the perspective of a single endpoint (i.e., the orchestrator). It refers to an executable process that may result in a persistent, multi-step interaction model where the interactions are always controlled from the point of view of a single entity involved in the process. Choreography, on the other hand, provides a global

view of message exchanges and interactions that occur between multiple process endpoints, rather than a single process that is executed by a party. Thus, choreography is more akin to a peer-to-peer (P2P) architecture and offers a means by which the rules of participation for collaboration are clearly defined and agreed upon. Even though there exists competing standards for both the models of composition, namely WS-BPEL [16] for orchestration and WS-CDL [107] for choreography, it is widely accepted that both orchestration and choreography can (and should) co-exist within one single environment.

Assuming that either model of composition is chosen, in general, there are two main ways for realizing a feasible composition: manually and (semi-) automatically. Manual composition mostly takes place statically during the design-time when the architecture and the design of the software system is being planned. Here, the services to be used are selected, integrated together, and finally deployed. As obvious, such a process will become cumbersome and error-prone, specially when modeling complex software systems comprising of hundreds, if not thousands, of services. Furthermore, the composite system will work fine as long as there is no (unexpected) change in the component services that take part in the composition. But, in certain cases the entire composite service might fail to execute if one or more of the component services become unavailable/inaccessible. On the other hand, (semi-) automatic composition tries to address some of these drawbacks by providing techniques, which given the specification of a composite (or *goal*) service, will automatically select and integrate a set of component services that realizes the goal. Depending on the technique used, the specification can be provided as a transition system (see Section 2.2.1), a logic formula (see Section 2.2.2), an UML diagram (see Section 2.2.3), and so on. A positive aspect of such techniques is that it reduces the cognitive burden on the service developer by essentially removing the requirement to manually discover and assemble component services. However, in spite of its advantages, (semi-) automatic composition is not widely pursued in the industry in part due to added complexity and overhead, lack of robust tooling support and development environments, and scalability and efficiency problems. We discuss some of these issues and challenges and outline how we address them in this thesis in the proceeding sections.

1.2.3 Research Questions and Challenges

As illustrated in the previous sections, Web service composition, and SOC in general, raises many challenging issues. These include:

- *Service Specifications:* A very important requirement to develop effective techniques for service composition is to build description languages that are expressive enough to capture various functional and behavioral properties of services. It should be intuitive, yet unambiguous, in representing the service semantics. More specifically, the issues that need to be addressed in this context include: (*i*) how easy it is for a service developer to comprehend the language and model composite (goal) service specifications? (*ii*) how should the composition engine interpret the developer's requests? (*iii*) how to verify that the language is (logically) correct? (*iv*) what is the formal basis for representing a composition of services from the goal service specification?

- *Service Discovery:* Assuming that the composition engine is able to interpret a service developer's request for building a composition, the next very important step in this process is to find suitable candidate services that can provide (in-part or entirety) the desired functionality. In practice, there might be hundreds, if not thousands, of candidate services that the composition engine will have to analyze which can lead to an exponential blowup. Furthermore, in addition to functional, the service developer might also provide non-functional requirements (e.g., Quality of Service) as part of the composition request. Consequently, the service composition engine should be capable of handling such requests. More specifically, the issues that need to be addressed in this context include: (*i*) how can service providers advertise their services that would enable efficient discovery? (*ii*) how to do service matchmaking based on functional (input/output) as well as behavioral properties? (*iii*) how to disambiguate between services that provide the same functionality, but have different specifications or vice-versa? (*iv*) how to optimize the search?

- *Verification and Validation of Composition:* Given the specification of a goal service,

once a feasible composition is determined, it is imperative to validate and verify that the composition satisfies all the desired requirements. Such a verification step can be done statically (before execution) and/or dynamically (during execution). Static verification can be usually done using model-checking techniques [70] whereas dynamic verification requires empirical test-runs. More specifically, the issues that need to be addressed in this context include: (*i*) how to model the properties (both functional and behavioral) that we want to verify? (*ii*) how to develop new or extend existing techniques in formal methods-based verification for service composition? (*iii*) how to build test-cases for dynamic verification?

- *Service Specification Reformulation:* During the process of building a feasible composition, it might happen that certain requirements of the goal service cannot be met by any of the available component services, thereby resulting in the failure of composition. Typically, in such a situation, the service developer has to manually modify the goal service specification and repeat the composition procedure. As expected, this process can be cumbersome and non-trivial for complex specifications. Instead a more practical solution will try to automatically correct the failures without any guidance from the developer and without changing the 'overall' functionality of the desired composition. This reformulation of the goal service can lead to situations in which it can be realized by suitably composing a set of component services. More specifically, the issues that need to be addressed in this context include: (*i*) how to formally characterize the problem of reformulation of the goal service specification during composition? (*ii*) how to represent the functional and behavioral properties of a service that can be used for reformulation? (*iii*) how to plug-in specification reformulation as part of the composition procedure, that is, how to enable composition and reformulation simultaneously?

- *Analysis for Compatibility and Replaceability:* Assuming that a feasible composition W_G has been obtained, in many cases it might happen that one of the candidate services W_i becomes unavailable either because the service provider for W_i chooses not to offer it any more or updates it (e.g., by adding/removing some of W_i's features). In such

circumstances, W_i has to be substituted by an 'equivalent' service $W_i^{'}$, which provides the exact same functionality. Similar to verification, here also substitution can be carried out statically and/or dynamically. More specifically, the issues that need to be addressed in this context include: (*i*) how to find a service that is equivalent to another? (*ii*) how to ensure that the replacement service is compatible with the overall functionality of the composition? (*iii*) how to build approaches for dynamic substitution?

- *Execution Management and Monitoring:* As with any distributed system, management and monitoring of service execution is a big research challenge. This challenge becomes even more formidable when the goal is to enable autonomic [131] capabilities that would possibly allow management related problems to resolve with minimal (and in ideal cases, no) human intervention. More specifically, the issues that need to be addressed in this context include: (*i*) how to enable services to self-configure and optimize themselves depending on the operating conditions? (*ii*) how to enable services to self-heal by discovering, diagnosing and reacting to disruptions? (*iii*) how to enable services to self-protect by anticipating and detecting hostile behaviors (e.g., denial-of-service attacks)?

- *Tooling Support:* Last, but not the least, an important challenge for wide-scale adoption of service composition techniques is to provide robust and intuitive tooling support. More specifically, the issues that need to be addressed in this context include: (*i*) how to build user-friendly interfaces for service modeling by leveraging approaches from HCI (Human Computer Interaction)? (*ii*) how to build tools that are efficient in terms of resource usage (e.g., tools that can be used in mobile computing devices such as PDA)?

1.3 Goals and Main Results

Motivated by the challenges and research questions as illustrated in the previous section (Section 1.2.3), we propose a formal and comprehensive approach and an end-to-end framework for Web service composition which simultaneously addresses the following issues:

- *Modeling Complex Web Services using Abstraction, Composition and Reformulation:* We

proposed an interactive and verifiable framework Modeling Web Service Composition and Execution (MoSCoE). This framework provides the architectural foundation for incremental development of composite services based on three basic principles: *abstraction, composition* and *reformulation*. By abstraction, we refer to the ability of MoSCoE that allows the users (i.e., service developers) to specify an abstract and possibly incomplete specification of the (goal) service. This specification is used to select a set of suitable component services such that their composition realizes the desired goal in terms of both functional and non-functional requirements. In the event that such a composition is unrealizable, the cause for the failure of composition is determined and is communicated to the user thereby enabling further reformulation of the goal specification. This process can be iterated until a feasible composition is identified or the user decides to abort.

- *Web Service Specification Reformulation:* We propose an approach for enabling Web service composition via automatic *reformulation* of the desired (or goal) service specifications in the event when the service composition algorithms fail to realize the goal service whenever the available component services cannot be used to "mimic" the structure of the goal service, even if the overall functionality of the goal service can be realized by an alternative formulation of the goal specification. In particular, we model services in our technique using labeled transition systems (LTS) and describe an efficient data structure and algorithms for analyzing data and control flow dependencies implicit in a user-supplied goal LTS specification to automatically generate alternate LTS specifications that capture the same overall functionality without violating the data and control dependencies implicit in the original goal LTS, and determine whether any of the alternatives can lead to a feasible composition. The result is a significant reduction in the need for the tedious manual intervention (by the service developers) in reformulating specifications by limiting such interventions to settings where both the original goal LTS as well as its alternatives cannot be realized using the available component services.

- *Context-Specific Web Service Substitution:* We propose a general technique for context-specific Web service substitution, where *context* refers to the overall functionality of the

composition that must be preserved after the substitution. In particular, we introduce two variants of the context-specific service substitutability problem that are based on weaker and flexible requirements compared to existing techniques. Our solution makes it possible to safely replace a service W_i with $W_i^{'}$ within the context of a given composition, even though $W_i^{'}$ may not meet the stronger requirement of being functionally or behaviorally equivalent to W_i. More precisely, we represent a composition (denoted by $||$) of two services W_1 and W_2 that realizes a specific functionality or property (denoted by φ and expressed in temporal logic [69]) by $W_1 \,||\, W_2 \models \Phi$. In the event W_1 becomes unavailable, the technique identifies candidates ($W_1^{'}$) that can be used as replacement for W_1 in the *environment* W_2 and *property* Φ.

- *Service Interoperability:* We provide an approach for ontology-based service discovery and composition. Web services, in general, are autonomously developed and maintained software entities. Consequently, it is unrealistic to expect syntactic and semantic consistency across independently developed service libraries that are analyzed for for composition, reformulation and substitution. Towards this end, realizing the vision of the Semantic Web, i.e., supporting seamless access and use of information sources and services on the Web, we build on recent developments in ontology-based solutions on information integration to develop principled solutions to addressing the semantic interoperability problem in service-oriented computing. Specifically, we introduce ontology-extended components and mappings between ontologies to facilitate discovery and composition of semantically heterogeneous component services.

- *Open-Source Implementation and an Empirical Study:* We provide an implementation of the proposed techniques in the MoSCoE prototype. We adopt emerging Web services standards including WSDL and BPEL. Additionally, we provide case studies demonstrating the applicability of the tool. The implementation is made open-source (under GNU Public License) and can be accessed at http://www.moscoe.org.

The results of this thesis have been published in international journals, books, and conference and workshop proceedings [146, 148, 149, 150, 151, 152, 153, 154, 155, 156, 158].

1.4 Thesis Outline

The rest of the dissertation is organized as follows:

- Chapter 2: A brief introduction to various Web services standards together with a representative set of existing work in Web services composition, adaptation and substitution is presented.

- Chapter 3: A general framework for representing the behavioral descriptions of Web services as labeled transition systems is given. We show how such a representation can be used to model composition of services as well as finding out equivalence services. We also show how current Web service description languages (e.g., WS-BPEL) can be translated to labeled transition systems.

- Chapter 4: The problem of composing Web services is formally defined and an approach for building composite services is proposed. We show that our approach can be applied to model complex services that satisfy both functional and non-functional (e.g., Quality of Service) requirements. We also provide soundness and completeness guarantees of the proposed algorithms.

- Chapter 5: The problem of Web service specification reformulation is introduced and a solution is proposed. We illustrate how our technique can be applied to automatically correct failure(s) in the composition process without any guidance from the developer and without changing the 'overall' functionality of the desired composition.

- Chapter 6: An approach for Web service substitution is proposed. At its core, the technique considers the 'context' (i.e., the overall functionality) of the composition and analyzes substitution in a manner that preserves the context after the substitution is carried out. We also demonstrate that our technique is sound and complete and relaxes

the stronger requirement of functional/behavioral equivalence between services proposed in the existing work.

- Chapter 7: The techniques for Web service composition are extended to yield an approach for discovering and composition of Web services. We introduce the notion of ontology-extended components and mappings between ontologies to facilitate interoperability between multiple, semantically heterogeneous services.

- Chapter 8: A system called MoSCoE (Modeling Web Service Composition and Execution, http://www.moscoe.org) for interactive Web service composition is designed. We describe the architectural and implementation details of MoSCoE as well as illustrate its usability using case studies.

- Chapter 9: We conclude with a summary, a list of contributions that this dissertation makes and several directions for future work.

CHAPTER 2. RELATED WORK

This chapter surveys a representative set of existing literature that is related to the work presented in this thesis. The chapter is divided into three main sections. The first section focuses on various Web services standards and technologies that are at present already in place or developed by the services computing community. The second section provides a literature review of state-of-the-art in automatic Web service composition. Research areas that are closely related to and are complementary to Web service composition, namely Web service substitution and adaptation, are surveyed in the third section.

2.1 Web Services: Standards and Related Technologies

We provided the definition of Web services as outlined by the Web Services Architecture [44] in Section 1.2.1, which we re-state for the sake of readability: *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [62]). Other systems interact with the Web service in a manner prescribed by its description using SOAP [90] messages, typically conveyed using HTTP with an XML [51] serialization in conjunction with other Web-related standards.* From this definition, it can be inferred that at the *conceptual* level, Web services are Web-accessible software system that provide certain functionality which can be invoked and respond to message interactions based on an XML messaging standard. Each service can be identified uniquely by an URI (Uniform Resource Identifier) and provide an interface to methods which can be executed via a message handler. The handler implements the logic for processing the messages (i.e., instructions) detailing what data should be passed on to what method for execution. On the other hand, at the *physical* level, Web services are

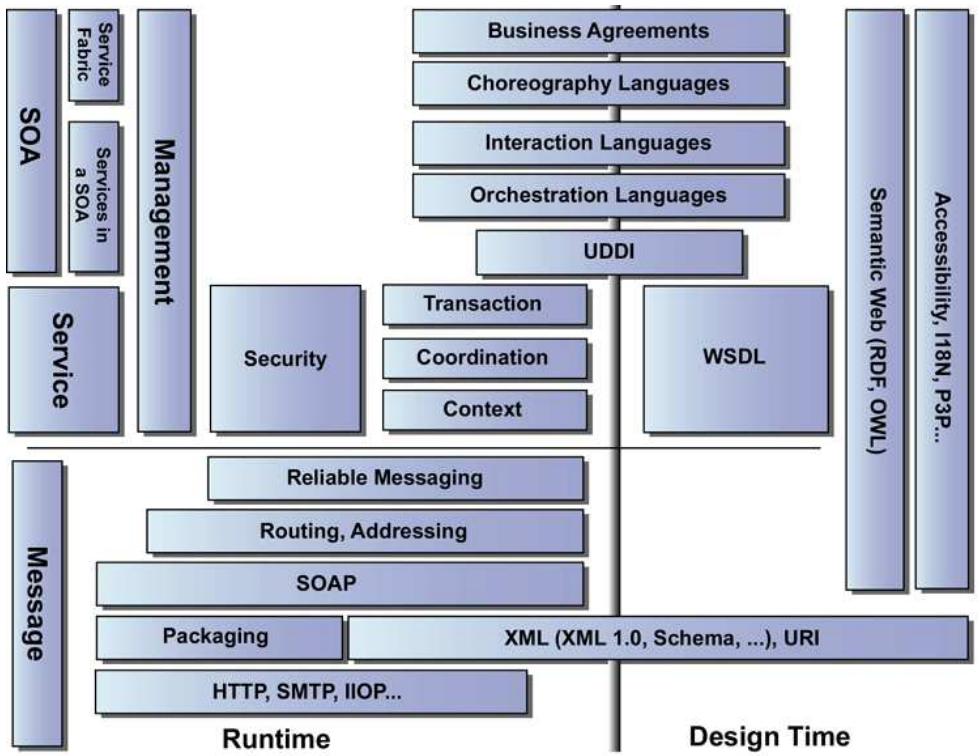


Figure 2.1 Web Services Protocol Stack

built from a stack of emerging standards and protocols (Figure 2.1 & Table 2.1). Although an elaborate discussion of all the specifications is beyond the scope of this thesis, we briefly describe some of the existing standards which are pertinent within our context of Web service composition.[1] These include:

- *Web Services Description Language:* The Web Services Description Language (WSDL) [62] is an XML-based language submitted to W3C for recommendation and is used to describe network services. The language represents services as network endpoints (or ports) and provides a model describing the communication between multiple services. The model introduces the notion of messages which are abstract representation of data being exchanged, and port types that are an abstract collection of various operations supported by a particular service. The data format specifications for a particular port type along with a concrete protocol constitutes a reusable binding, where the messages

[1] Interested readers can find more information about the entire spectrum of Web services protocols in [143, 177, 189, 190].

Layers	Sub-Layers	Standards
Business Domain	Business Domain Specific Extensions	Various
Management	Distributed Management	WSDM [54], WS-Manageability [165]
	Provisioning	WS-Provisioning [192]
Security	Security	WS-Security [132]
	Security Policy	WS-SecurityPolicy [66]
	Secure Conversation	WS-SecureConversation [14]
	Trusted Messsage	WS-Trust [15]
	Federated Identity	WS-Federation [118]
Portal	Portal & Presentation	WSRP [185]
Transactions	Asynchronous Services	ASAP [80]
	Transaction	WS-AtomicTransactions [56], WS-Coordination [57]
	Orchestration	BPEL4WS [16], WS-CDL [107]
Messaging	Events & Notifications	WS-Eventing [48], WS-Notification [86]
	Multiple Message Sessions	WS-Enumeration [7], WS-Transfer [8]
	Addressing	WS-Addressing [49], WS-MessageDelivery [106]
	Reliable Messaging	WS-ReliableMessaging [38], WS-Reliability [104]
	Message Packaging	SOAP [90], MTOM [89]
Metadata	Publication & Discovery	UDDI [53], WSIL [21]
	Policy	WS-Policy [19], WS-PolicyAssertions [50]
	Message Description	WSDL [62]
	Metadata Retrieval	WS-MetadataExchange [20]

Table 2.1 Web Services Specifications

and operations are bound to the protocol and the format. A client program interacting with the Web service can read its WSDL description to determine what operations can be invoked. During the invocation process, the client can send SOAP [90] messages over various Internet protocols such as HTTP.

- *Web Services Business Process Execution Language:* The Business Process Execution Language (BPEL) [16] is an XML-based executable language for modeling business processes, which in general, manifest as Web services defined using WSDL. BPEL is an orchestration language (Figure 1.4(a)), and hence focuses on the view of one participant (i.e., central control of behavior), where the participants are represented using a state transition model. The model exposes a set of publicly observable behaviors which include when to send/receive messages, when to compensate for failed transactions, when to execute functional operations (e.g., retrieving data from a file), and so. In addition to this messaging facility, BPEL allows writing expression and queries in multiple languages

(such as XPath [63]) and supports structured-programming constructs including execution of commands in sequence (e.g., `if-then-else`, `while`) and parallel (e.g., `flow`). Refer to Section 3.2.2 for more details on BPEL.

- *Web Services Choreography Description Language:* The Web Services Choreography Description Language (WS-CDL) [107] is an XML-based language that can be used to describe the common and collaborative observable behavior of multiple services that need to interact in order to achieve some goal. As opposed to BPEL, WS-CDL describes this behavior from a global or neutral perspective rather than from the perspective of any one party. Such an interaction typically happens through some common understanding between the participating services or by a declaration of interest in the progress of one service by another. The global model ensures that no single party adopts a biased view towards other services. Instead WS-CDL adopts a collaborative observable behavior of all the services such that on one service can exert any control over any other service. Consequently, each service carries out its functionality in a distributed fashion and has a distinct relationship with its peers.

2.2 Web Service Composition

In this section, we consider again the work presented in the previous section which is targeted towards the problem of Web service composition. In particular, we focus on techniques that leverage existing approaches in different domains such as artificial intelligence, software engineering, human-computer interaction, and so on.

2.2.1 Techniques based on Formal Methods

Formal Methods [47] is an area of study that provides a language for describing a software artifact (e.g., specifications, design, source code) such that formal proofs are possible, in principle, about properties of the artifact so expressed. In the context of Web service composition, typically the property proved is that an implementation is functionally correct, that is, it fulfills a particular specification. In the recent past, many research efforts for service composition

have adopted formal methods techniques to leverage its mathematically-precise foundation for providing theoretically sound and correct formalisms. We discuss few of those approaches in the following paragraphs.

Pistore et al. [162, 163] represent Web services using transition systems [100] that communicate via exchanging messages. Their approach relies on planning via symbolic model checking techniques to determine a parallel composition of all the available services, and then generate a controller to control the composed services such that it satisfies the user-specified requirements. Informally, if $W = \{W_1, W_2, \ldots W_n\}$ is the set of available services, ρ is the service developer-specified requirement (i.e., ρ describes the goal service specification), and $||$ is the composition operator, the aim is to determine a "controller" W_c, such that: $W_c \rhd (W_1 || \ldots || W_n) \models \rho$. The goal specification is described using a temporal logic-based language called EAGLE, whereas the transition systems representing the component services are generated either from OWL-S [188] or BPEL [164]. Furthermore, the transitions systems in their approach are non-deterministic with partial observability (i.e., only partial information is available at any given state). Consequently, due to the incomplete knowledge on the initial states and on the outcome of the actions, at each execution step, each service could be in a set of states that are equally plausible given the initial knowledge and observable behavior.

Berardi et al. [30, 31, 32] also provide a formal framework where services are represented using transition systems. The approach assumes that the services exchange messages according to a pre-defined communication topology (referred to as the linkage structure), which is expressed as a set of channels. Two inputs are given to the composition synthesis system: (*i*) a desired goal service behavior (i.e., the set of all possible conversations) specified as a labeled transition system, and (*ii*) the composition environment which comprises of the linkage structure, the set of component services and the messages exchanged between them. The output of the synthesis is a mediator (also represented as a labeled transition system) which orchestrates the execution of the component services such that their conversations are compliant with the goal service specification. The authors encode the composition problem as a Propositional Dynamic Logic (PDL) [93] formula Φ and reduce the generation of the mediator to the satisfi-

ability of Φ. Furthermore, formal proofs are provided stating that a composition exists if and only if Φ is satisfied and that the model of Φ exactly represents the composite service.

Hamadi and Benatallah [92] apply a petri net-based algebra to model the control flow and capture semantics of complex Web service compositions. Their framework provides various structured-programming constructs such as sequence, alternative, iterative and arbitrary, and the authors show how these constructs can be used to determine and verify a composition. However, [92] does not provide an approach for manual or (semi-) automatic service composition. SELF-SERVE [27, 28] built on this work to provide the ability for dynamically composing and executing Web services represented as state charts. One of the key features of SELF-SERVE is to adopt a peer-to-peer (P2P) computing environment for executing the (composite) services, which in practice has multiple advantages (in terms of scalability, fault-tolerance etc.) compared to centralized architectures. Similar to [32], here also a linkage structure between the peer services is created which in turn is leveraged by the composition algorithm for generating message routing tables, and in essence, creating the P2P conversation model.

Bultan et al. [55, 35, 78, 79] propose techniques for analyzing conversations of composite Web services for both synchronous and asynchronous communication models. Synchronous communication happens when a message sent by a service is received immediately by the recipient service, whereas in asynchronous communication the message may not be received immediately (i.e., the message may be queued). As expected, analysis within an asynchronous messaging is much more difficult due to the added complexity of the message queues. In particular, the authors represent services using finite state machines [100] augmented with FIFO (First-In First-Out) message queues and develop methods for synchronizability and realizability analysis, where synchronizability analysis determines whether a composite service generates the same set of conversations under synchronous and asynchronous communication models, and realizability analysis ascertains whether a given conversation protocol (e.g., the goal service specification), modeled as a finite state machine, can be realized by a feasible composition of component services which communicate asynchronously. Note that even though this work is not directly related to developing algorithms for service composition, it addresses

a very important problem of verifying the correctness of composition.

Gwen Salaün et al. [176] apply Process Algebra (PA) [99] to model Web services in at least two different ways: (i) at *design time*, PA can be used to describe an abstract specification of the system to be developed, which can be validated and used as a reference for implementation; (ii) by applying *reverse engineering*, existing Web service interface descriptions can be translated to PAs. Specifically, this work adopted Calculus of Communicating Systems (CCS) [129] as the PA and demonstrated techniques for translating BPEL [16] processes into CCS, which can then be verified to reason about properties specified in temporal logic.

Ferrara [74] advocates a thought similar to [176], that is, using Process Algebra (PA) for modeling Web services to: (i) establish whether a service can substitute another service(s) in a composition; (ii) develop Web services by adopting hierarchical-refinement techniques [110, 112] that allow to begin with an abstract description of a process which can be refined iteratively; (iii) analyze and find redundant services in a community; and (iv) verify desirable properties specified in a temporal logic-based language. In particular, this works focuses on providing a two-way mapping between BPEL/WSDL and Language of Temporal Ordering Specifications (LOTOS) [68], one of the most expressive process algebra. The advantage of using LOTOS (e.g., opposed to using CCS as proposed in [176]) is that it allows addressing issues related to exchange of data during Web service interactions and dynamic service compositions. As a result, one can verify services that deals with messages and with messages, where the properties of the services depend on the values (e.g., for a set of input values, some properties are satisfied). Additionally, LOTOS allows defining abstract data types and operations on them, which correspond to the data type definitions in BPEL/WSDL specified using XML Schema. Consequently, if a service is modeled using LOTOS (and is later mapped into BPEL/WSDL), the data types can be checked and verified for desirable properties.

2.2.2 Techniques based on AI Planning

Planning [175], in general, can be regarded as an area of study that is concerned with automatic generation of plans that will be able to solve a problem within a particular domain.

Typically, a plan consists of sequence of actions, such that given an initial state or a condition, a planner will suitably select a set of actions which, when executed according to the generated plan, will satisfy certain goal conditions. In the context of Web services, a planning domain can be represented by a sextuplet $(W, S, A, \longrightarrow, s_0, s_G)$, where W is the set of available Web services, S is the set of all possible states of these services (world), A is the set of actions/functions provided by the services that the planner can perform in attempting to change the state from one to another in the world, $\longrightarrow \subseteq S \times A \times S$ is the set of state transitions which denote the precondition and effects for execution of each action, and finally $s_0 \in S$ and $s_G \in S$ are the initial and goal states, respectively, specified in the requirement of the Web service requesters to indicate that the plan initiates its execution starting from state s_0 and terminates at state s_G. Given this domain, many approaches have been proposed by applying a variety of planning techniques that will generate a plan for realizing the goal requirements.

Planning Domain Definition Language (PDDL) [83] is one of the very widely known description languages in the planning domain and has influenced the development of Web service description languages such as OWL-S [121] (Web Ontology Language for Services). McDermott [123] extend PDDL by introducing the notion of "value of the action", essentially representing certain information that is created or learned as a consequence of executing a particular action. The main intention of introducing this extension was to have the ability to capture the information and the content of messages that are exchanged between the services. The work demonstrates how this extended language can be used with estimated regression planners to create conditional plans that achieve the desired goal. In particular, given a goal and an initial situation, the technique does a situation-space search for a sequence of steps that achieve the goal. A search state is a series of feasible steps starting in the initial situation and ending in a situation which is potentially close enough to satisfy the goal. To make the search efficient, the approach also proposes certain heuristics.

Medjahed et al. [126, 127] apply a rule-based planning technique for finding feasible compositions and introduced a declarative language for describing the goal requirements. The core of the approach comprised of developing composability rules that consider and analyze syn-

tactic and semantic properties of the Web services to devise a plan. Such rules, for example, might specify that two Web services W_1 and W_2 are composable only if the output messages of W_1 are compatible with the input messages of W_2. The composition model comprises of four different steps: (*i*) during the specification phase, the requirements of the composition are specified using an XML-based language (developed by the authors) called Composite Service Specification Language (CSSL) which adopts an ontology-based model suitable for describing semantics-enabled Web services [124]; (*ii*) once the goal service specification is provided, multiple plans are generated by a matchmaking algorithm leveraging a set of pre-defined composability rules; (*iii*) at least one feasible plan is selected based on additional non-functional constraints, and (*iv*) finally an executable code is generated in WSFL [113].

Sirin et al. [179] adopt Hierarchical Task Network (HTN) planning [134] for automated composition of semantic Web services. The main motivation for using HTN was because the concept of task decomposition in HTN planning is very similar to the concept of composite process decomposition in OWL-S process ontology [121]. The authors provide an algorithm for translating OWL-S service descriptions into SHOP2 [134] (the HTN planner used) domain and device a planning procedure for generating feasible composition plans. In addition, [179] also proves the correctness of their approach by showing the correspondence to the situation calculus semantics of OWL-S.

SEMAPLAN [6] attempts to leverage traditional AI planning and information retrieval techniques for building a semi-automated service composition tool. The technique relies on domain-dependent/independent ontologies [88] for calculating semantic similarity scores between the concepts/terms in service descriptions, and applies this score to guide the searching process of the planning algorithm. The planning algorithm is based on a cost-based heuristic which leverages the semantic scores and is built on the Planner4J framework [181]. The experimental results demonstrate that SEMAPLAN performs superior compared to the traditional planning based techniques.

Similar to [6], Agarwal et al. [5] also combine traditional AI planning techniques with semantics-based approaches to build an end-to-end solution for service composition. The ser-

vices in this approach are represented using OWL-S descriptions and the composition is divide into two parts: logical and physical. During logical planning, a planner is used to create composition plans based on service 'types' according to the desired functional requirements. If one or more composition plans can be obtained, then during the physical composition phase service 'instances' are selected based on non-functional requirements (e.g., Quality of Service) to instantiate the plans for deployment. The authors demonstrate the such a separation leads to scalability by providing the ability to handle different goals, different data, different rates of change of data at each planning stage, and different means to optimize them.

2.2.3 Techniques based on Model-Driven Architectures

Model-Driven Architecture (MDA) [108] is a software design approach that promotes a systematic use of "models" as primary engineering artifacts throughout the software development lifecycle. The main objective of MDA is to separate design (i.e., the development of the model) from the architecture. Depending on the discipline, different types of modeling languages can be used to express information about a system that is defined by a consistent set of rules. The rules, in essence, provide a way for interpreting the meaning of the components in a particular model. Furthermore, depending on the approach adopted, sometimes the model is developed with certain level of details and the executable code (corresponding to the model) is generated separately, and sometimes the entire code is generated completely (or in-part) from the model itself. One of the modeling languages which has become the de-facto industry standard for MDA is Unified Modeling Language (UML) [76], which is a general-purpose language that allows creation of abstract/concrete models of a system using a graphical notation.

Orriëns et al. [139] propose an approach for development and management of dynamic service composition. Their main idea was to make the fundamental composition logic agnostic to particular composition specifications (such as BPEL) in order to raise the level of abstraction. This will in turn enable rapid development and delivery of service compositions based on proven and test models for software-development life cycle. In particular, the authors use UML as the modeling language and demonstrate how it can be used to steer the composition process and

finally mapped to executable languages such as BPEL.

Similar [139], Grønmo et al. [87, 180] also propose an approach for Web service composition using UML. However, an unique aspect of their approach is the ability of translating WSDL descriptions into UML models. Consequently, existing services can be modeled within the UML environment for building compositions, which in turn can be translated into executable BPEL specifications. Thus, UML acts as a common integration platform. Additionally, the authors also provide an open-source implementation of their tool.

Manolescu et al. [119] present a high-level language and methodology for designing and deploying Web applications using Web services. In particular, the authors extend WebML [59] to support message-exchange patterns present in WSDL and use the WebML hypertext model for describing Web interactions and defining specific concepts in the model to represent Web service calls. Consequently, the Web service invocation is captured by a visual language representing the relationships between the invocations and the input/output messages.

Gannod et al. [81, 186] develop an approach for construction of OWL-S [121] specifications using model-driven techniques. The authors propose a 2-stage approach, where in the first stage UML is to generate an OWL-S description of a Web service by mapping UML Activity Diagrams to OWL [17]. In the second stage, constructs are provided for mapping the concepts in OWL-S description to concepts in the WSDL file of a concrete service for realization. Specifically, the profile and process constructs of OWL-S description and a set of existing WSDL files are used to generate the OWL-S grounding construct. This work was later extended in [187] using Object Constraint Language (OCL) [2] for modeling structured-programming control constructs.

2.2.4 Techniques based on Graph Theory

Graph Theory [37] is the area of study in computer science which analyzes mathematical structures, called graphs, that are used to model pairwise relationships between objects from a certain collection. Informally, a graph comprises of a set of vertices and a set of edges that connect pair of vertices, such that edges may be directed or undirected. In the context of Web

services, graphs have been used to model control and data flow dependencies between various functions provided by a service.

Lang and Su [116] formalize the problem of Web service composition as an AND/OR graph [175] search problem, where the graph essentially represents the input/output dependencies between the service-functions. Given a request for building a composite service, the technique identifies component services that can satisfy the request and dynamically constructs an AND/OR graph to capture the data dependencies among the Web services that can be used to realize the composite service specification. The graph is modified based on the information provided in the service request and the search algorithm is used to search the modified AND/OR graph to find alternate composition templates repeatedly until the service developer approves one. After a template is selected, the system then attempts to bind the template's service operations to registered services to generate a WSFL [113] specification.

Hashemian and Mavaddat [96] propose an approach for automatic Web service composition by combining techniques based on interface automata and graph theory. In particular, the authors model Web services using interface automata [65], which expose the inputs and outputs of a component along with a temporal ordering of actions it performs, and represent the data dependencies between the component services using a dependency graph, where the nodes of the graph correspond to the inputs and outputs of the Web services and the edges represent the associated Web services themselves. The composition comprises of two stages, where in the first stage, suitable services are discovered that can potentially participate in the composition, and in the next stage a dependency graph is created using the discovered services. This technique was later extended in [97] to model composition of stateless Web services and shown using process algebra that the composition generated is correct.

Oh et al. [138] leverage the A* search algorithm [175] to develop a novel technique called BF* (BF-Star) for sequential composition of services. In particular, the authors introduce the notion of "joint-matching" which represents the data dependencies required to invoke various services, such that the invocation process corresponds to the desired conversation model of the composite service. The technique uses an efficient data structure called bloom filter [40] to

identify the data dependencies and then applies BF* to construct an end-to-end joint matching of Web services.

Gekas and Fasli [82] develop a service composition registry as a hyperlinked graph network of scalable size, and dynamically analyze its structure to derive useful heuristics to guide the composition process. Similar to above mentioned approaches, here also the composition process is modeled into a graph search problem where the search space, represented as a hyperlinked graph, consists of all the potential Web service operations that can be part of a feasible composition. In order to make the search efficient, heuristics are applied that essentially measure the "semantic goodness" of a service in terms of how the service can provide the desired functionality. The authors also demonstrate that their approach can scale with the increasing size of the composition registry.

2.3 Additional Research Areas Related to Web Service Composition

In the following, we discuss existing literature in Web service substitution and adaptation, areas that are highly relevant to service composition, and hence have become active research topics.

2.3.1 Web Service Substitution

Given a composition model, the problem of Web service substitution concerns with analyzing whether a particular service can be replaced with another without violating the desired requirements. A simpler way of carrying out such an analysis between the replaced and the replacement services is to determine whether they have the exact same interface definition (e.g., "types" of messages exchanged) or not. However, this is not often sufficient since the interface definitions fail to capture the exact behavior (i.e., the conversation model) of the services. Consequently, most of the existing techniques in Web service substitution have focused substitutability based on structural and behavioral representations.

Bordeaux et al. [45] introduce three different notions of compatibility of Web services (namely, observation indistinguishability, unspecified receptions, and deadlock freeness) and

use them as basis to define context- dependent and independent substitution of Web services (that are modeled as labeled transition systems). Two services, S_1 and S_2 are said to be observation indistinguishable if both can send and receive messages simultaneously for all possible interactions. On the other hand, if there exists at least one interaction where a particular message sent by the sender (say S_1) cannot be received by the recipient (say S_2), then S_1 and S_2 have unspecified receptions. And finally, if none of the interactions between S_1 and S_2 lead to a deadlock, then S_1 and S_2 are said to be deadlock-free. Based on these ideas of service compatibility, substitution with respect to a "context" is defined, where context refers to a particular application or a functionality. This work was later extended by Liu et al. [117] to handle non-determinism in service behavior.

Mecella et al. [125] propose a formal model for substitutability of Web services. The authors use state machines for representing the behavioral description of services and analyze computational traces (i.e., the sequence of events) of services for determining substitutability. In particular, a service S participating in a composition C can be replaced by another service S' if the sequence of executions of S and S' with respect to C are equivalent (i.e., S and S' are trace equivalent). A typical trace would comprise of various input (output) messages received (sent) by a service along with various atomic actions executed as a consequence of the message interaction (with other services).

Benatallah et al. [25] introduce multiple operators for analyzing protocol compatibility and similarity in Web services. In particular, the authors characterize two different types of protocol compatibility, fully compatible and partially compatible, where the former corresponds to the situation when any conversation generated by a protocol P_1 can be understood by another protocol P_2, and the latter corresponds to the situation when such an understanding can be established for at least one conversation between the protocols. Using these notions of compatibility, analysis is done to determine whether two services exhibit the same behavior or if one can be used instead of another when interacting with a client. Four different classes of protocol replaceability are described (equivalence, subsumption, replaceability w.r.t. to client protocol and replaceability w.r.t to the interaction role) and algorithms are presented

for analysis corresponding to each class.

Beyer et al. [36] provide three different languages (namely, signature interfaces, consistency interfaces and protocol interfaces) for specifying Web service interfaces, and consider subsumption equivalence and subsumption ordering to ascertain replaceability of services. The interface languages demonstrate how different aspects of a service behavior can be captured in an increasingly complex manner and presented within a formal model. A signature interface simply specifies various methods of a service that can be invoked by a client, consistency interface specifies various propositional conditions on the method calls and output values that may result during a conversation, and finally temporal obligations on the ordering of method calls are modeled by protocol interfaces. The authors provide algorithms for analyzing compatibility and replaceability between signature interfaces using simple type checking, between consistency interfaces by solving propositional constraints, and between protocol interfaces by model checking temporal safety constraints.

Martens et al. [120] devise an approach for determining behavioral and syntactical compatibility between Web services that are modeled as petri nets. Similar to above mentioned approaches, here also the authors adopt trace equivalence and reachability analysis to determine similarity between two petri net models. However, their approach also ensures that there are no deadlocks between the compatible processes.

2.3.2 Web Service Adaptation

Adaptation by definition implies to "something" that is changed or changes so as to become suitable to a new or special application or situation. In the context of Web services, adaptation refers to two basic ideas: static adaptation and dynamic adaptation. The problem of static adaptation is concerned with analyzing techniques that are typically applied during a service composition process to (*i*) automatically build adapters to enable service mediation, and (*ii*) modify the specification of the goal (or desired) service for generation of a feasible composition. On the other hand, the problem of dynamic adaptation is concerned with developing methods that allow the execution model of a composite service to be modulated depending on run-time

conditions (e.g., Quality of Service requirements). As can be noticed, techniques developed for both the notions of adaptation complement each other and can be applied in unison for building robust service composition frameworks.

Harney and Doshi [94] introduce adaptation of Web services using value of changed information. Their technique relies on formulating queries for probing the status and dynamic properties (e.g., cost) of a service that change overtime to determine whether the service should be replaced in a particular composition. These queries are performed only after it is identified that the benefits of gathering the revised information is more than the cost involved for querying. This work was later extended in [95] to make the querying process more efficient by eliminating candidate services to be queried based on analysis of their information expiry timespan.

Chafle et al. [60] propose a framework for adaptive Web service composition and execution. Their approach was based on a staged solution that allowed adaptation by generation and deployment of multiple workflows at different stages based on feedback mechanisms and ranking functions. The monitoring infrastructure developed in [60] constantly monitors all the services taking part in a composition and sends information about QoS changes and run-time failures for appropriate adaptation. This work was also later extended in [61] by leveraging the work done in [94] to selectively monitor the services of "interest". The authors also provide experimental results demonstrating the robustness and scalability of their technique.

Kwan et al. [111] devise a proxy-based approach to context-aware adaptation of services. The main idea of this work was to leverage contextual information (e.g., memory, bandwidth) of the clients (e.g., mobile phones, PDAs) which are executing a particular service and adapt according to the constraints. The authors develop technique that estimate the resource usage required to execute a service instance in a particular device, and then adapt the execution parameters based on constraints of the executing environment. Although this work focused mainly on execution of mobile code, it can be extended to a traditional Web-based services setting.

Nezhad et al. [135] develop an approach for identification and resolution of mismatches

between service interfaces and protocols, and for generating an adapter specification. Their technique generates a "mismatch tree" comprising of mismatches of various types (e.g., signature, extra/missing messages) that requires inputs from the service developers for their resolution and assists the developers in generating an adapter to resolve the incompatibilities. Brogi and Popescu [52] also propose a similar approach to generation of BPEL adapters. However, the approach can only perform adaptation when there are no mismatches between the service interfaces and the interactions between the services are deadlock-free.

Sinha et al. [178] develop an on-the-fly approach for adapter generation using model checking where the protocols and the sequence of events between the protocols are represented using kripke structures and temporal logic, respectively. Their work provides a tableau-based algorithm for identifying and automatically generating an adapter (if it exists) by taking into consideration various types of mismatches. In addition, the technique ensures that the adapter generated for protocol communication satisfies fairness constraints, i.e., the constraints as specified in the temporal logic representation of the desired behavior.

2.4 Discussion

One of the key aspects of Service-Oriented Architectures (SOAs) is the ability to rapidly build new applications and services by assembling the existing ones. Developing techniques and approaches to facilitate such an assembly process automatically has been widely researched in both academia and industry. However, most of the existing approaches ignore or oversimplify multiple aspects and characteristics that are specific to Web services and SOA, and are vital for the success of service-oriented computing paradigm, in general. Some of the important aspects include representation of functional and behavioral properties of services, ability to handle failure of composition, service adaptation during composition, analysis of service substitution, and handling semantic heterogeneity in service specifications. Without addressing these issues in an uniform manner, the present techniques and tools can only operate in a restricted setting, and hence cannot be applied to a wide-range of realistic problems and application domains.

In our research work, we aim to provide an uniform framework that is capable of addressing

various problems, and in particular the aforementioned aspects, pertaining to Web services and SOA. Similar to existing approaches, our goal is also to develop techniques and tools that allow a service developer to model complex composite services that satisfy the desired requirements. However, we leverage and extend the current formalisms and techniques to develop novel algorithms and analysis methods for building an end-to-end platform to model and execute complex Web services.

In particular, we use labeled transition systems (LTSs) to represent the functional and behavioral properties of Web services. This choice is motivated by two main reasons: firstly, LTSs are a simple, yet intuitive, formal model for representing communicating systems such as Web services, and secondly, the LTS semantics are very similar to the semantics of the existing service specification languages such as BPEL. However, we extend the conventional notion of an LTS to represent infinite-domain variables and guards (or conditions) based on such variables.

Based on the LTS formalism, we propose an interactive technique for composing Web services. An important aspect of our work is the ability to determine the failure of a composition (if any) and appropriately notify the service developer. We claim and demonstrate that such a technique facilitates and helps the service developers in modeling complex services. However, manual inspection of failures and taking corrective measures to achieve a feasible composition is a time-consuming and cumbersome process, even while modeling simpler services. To address this need, we analyze data and control flow requirements in the service specifications and devise algorithms for automatically identifying alternate composition models that satisfy those requirements with minimal supervision from the service developers. Furthermore, since in many cases the component services participating in a composition process might become unavailable, there is always a need to replace such services with alternate ones in a transparent manner. A particular aspect of such replacement techniques is the ability to provide a guarantee that none of the desired requirements of the composition are violated. We have investigated such a technique and proposed a sound and complete approach for Web service substitution that conforms to the composition requirements.

CHAPTER 3. WEB SERVICES AND LABELED TRANSITION SYSTEMS

This chapter provides a general framework for representing behavioral descriptions of Web services as labeled transition systems (LTSs). The chapter is divided into three sections. The first section introduces formalisms related to LTSs and ideas pertaining to equivalence between LTSs and composition of multiple LTSs. The second section focuses on the mapping and translation of existing service description and modeling languages to LTS representations. The third section concludes the chapter with a discussion.

3.1 Representing Web Services as Labeled Transition Systems

Web services, in essence, are software systems that can be invoked by a client over a network using standard Internet protocols to realize a desired task. Clients can take the form of a human being or another service itself, and in practice, multiple different clients might interact with the service simultaneously by executing multiple instances of the service implementation. During the interaction of the client and the service, various actions of the service will be directly executed, with the possibility of certain actions being delegated to other services. Typically, the interaction occur by the exchange of messages between the interacting parties. According to the W3C recommendation [98], a *message exchange pattern* (MEP) is a template that establishes a pattern for the exchange of messages between two communicating parties. A MEP identifies a common grouping of related messages. The MEPs are defined based on the client (i.e., the service requestor) and service provider, and are named based on message characteristics in the service provider, for the sake of clarity. The concept of MEPs is still evolving, and the number of patterns are potentially unlimited. However, the two basic

forms of MEPs that are widely used and can be applied to construct most of the other patterns are:

- *In-only* ("*fire & forget*"): The client sends a message to the service provider and does not expect any related message.
- *In-Out* ("*request-response*"): The client sends a message to the service provider and expects a message.

where, the MEP names can be understood by replacing the *in* with request and *out* with response. Thus, depending on the MEP, the client can choose the execution of some action (by sending a message) and wait for the execution to finish and return of some information (in the case of *In-Out*). Based on the outcome of the execution (and whenever possible, on the returned information), the client might choose another action to invoke or terminate the interaction with the service indicating that all the desired task requirements of the client have been fulfilled by the service. The service (instance), on the other hand, after executing the invoked action, is either ready to execute new actions or is no more in a position to accept messages from the client, and hence execute new actions. However, in principle, a particular service instance might have to interact with a client infinitely. In such situations, termination of the service instance is not carried out, that is, the service is always able to accept messages from the client and execute actions.

We claim and discuss in the remainder of this chapter that such an interaction pattern between the client and the service representing their behavioral descriptions can be adequately modeled using labeled transition systems. We begin the discussion with an illustrative example.

3.1.1 Labeled Transition Systems

Example 1 *A client wants to search for a particular book in an online store, and if available, is willing to purchase it. Hence, it decides to interact with an available service,* `e-Buy`*, and activates an instance of the service. The service provides two functions to the client:* (*i*) `SearchBook` *for searching books, and* (*ii*) `PrchBook` *for purchasing books (if available). Assuming that the client decides to search for a particular book, it invokes* `SearchBook` *by first*

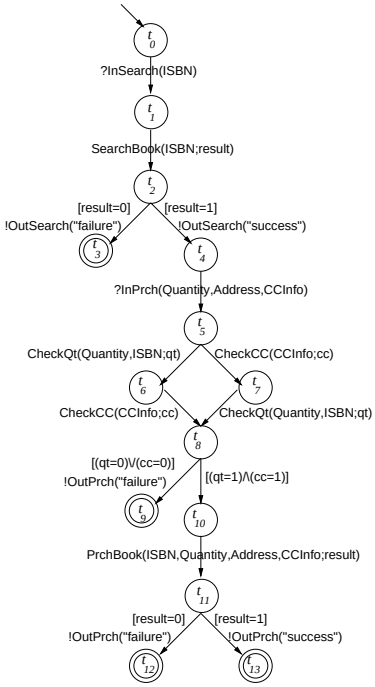


Figure 3.1 Labeled Transition System representation of `e-Buy` service

providing the ISBN of the book as input, and then waiting for `e-Buy` *to finish execution of the* `SearchBook` *function. Depending on whether the requested book is available or not, a "success" or "failure" message is sent to the client. If a "failure" message is sent, the service instance terminates. On the other hand, if a "success" message is sent, the service offers the client to make a purchase by providing the quantity, shipment address and credit card information. Assuming that the client is interested in making a purchase and provides such information,* `e-Buy` *will first check whether the desired quantity (credit card) is available (valid) or not, and then execute the* `PrchBook` *function and send an appropriate message to the client depending on the success or failure of the entire operation.*

Figure 3.1 shows the behavioral representation of the `e-Buy` service using a labeled transition system, which we define as follows:

Definition 3 (Labeled Transition System [100]) *A labeled transition system (LTS) is a tuple* $(S, \longrightarrow, s0, S^F)$ *where* S *is a set of states represented by terms,* $s0 \in S$ *is the start state,* $S^F \subseteq S$ *is the set of final states and* $\longrightarrow$ *is the set of transition relations of the form* $s \xrightarrow{\gamma,\alpha} t$

where:

1. *an action α such that*

   (a) $vars(\alpha) \subseteq vars(s)$ *if* α *is an output action*

   (b) $vars(\alpha) \cap vars(s) = \emptyset$ *if* α *is an input action*

   (c) $ivars(\alpha) \subseteq vars(s) \wedge ovars(\alpha) \cap vars(s) = \emptyset$ *if* α *is an atomic action*

2. *a guard* γ *such that* $vars(\gamma) \subseteq vars(s)$*, and*

3. $vars(t) \subseteq vars(s) \cup vars(\alpha)$.

where, (i) guards, denoted by γ, are predicates over other predicates and expressions; (ii) variables in a term t are represented by a set $vars(t)$; (iii) substitutions, denoted by σ, map variables to expressions. A substitution of variable v to expression e is denoted by $[e/v]$. A term t under the substitution σ is denoted by $t\sigma$; and finally (iv) action is a term that takes one of the following forms:

- `?msgHeader(msgSet)`: input action. Variables of the input action are in `msgSet`, i.e. $vars($`?msgHeader(msgSet)`$) =$ `msgSet`.
- `!msgHeader(msgSet)`: output action. Variables of the output action are also in `msgSet`, $vars($`!msgHeader(msgSet)`$) =$ `msgSet`.
- τ: an internal or unobservable action of a composition. Two entities synchronize on input and output action with the same message header to generate such an action.
- `funcName(I; O)`: atomic action with input parameters `I` and return valuation `O`. We say that $ivars($`funcName(I;O)`$) =$ `I`, $ovars($`funcName(I;O)`$) = \{$`O`$\}$ and $vars($`funcName(I;O)`$)$ $=$ `I` $\cup \{$`O`$\}$.

In addition (similar to atomic functions), we will also refer to the variables needed in an operation as *ivars* and variables obtained from an operation as *ovars*. Therefore, $ivars($`?msgHeader(msgSet)`$) = \emptyset$ as input operations obtain their messages from an external entity and

$ovars$(`?msgHeader(msgSet)`) = `msgSet` as variables from input operations can be used by the service executing the input operation. Proceeding further, $ivars$(`!msgHeader(msgSet)`) = $ovars$(`!msgHeader(msgSet)`) = $\emptyset$ as output operation variables are generated by the service from $ovars$ of input operations and/or atomic functions. Finally, $ivars$(`guard`) = $vars$(`guard`) and $ovars$(`guard`) = $\emptyset$.

For instance, Figure 3.1 shows the LTS representation of the `e-Buy` service described in Example 1. Here, the transition from state t_0 to t_1 is annotated with an input action `?InSearch(ISBN)`, where `InSearch` is the message header and `ISBN` is the variable in the input message. This action corresponds to an instance of the `e-Buy` service receiving an input message from the client. The transition from state t_1 to t_2 is annotated by an atomic action `SearchBook(ISBN;result)`, which corresponds to a function provided by the service, and where the argument(s) preceding ";" is(are) the input(s) to the function and the argument proceeding is the output of the function. In our case, `SearchBook(ISBN;result)` takes `ISBN` of the book as the input, searches the repository for the book, and generates an output `result` indicating whether the book is available or not. If the book is not available (denoted by [`result=0`]), a *failure* message is sent to the client, as shown by the transition from state t_2 to t_3, indicating the termination of execution of the service instance since t_3 is a final state (denoted by double circles). The output action corresponding to transmission of a message to the client is denoted by `!OutSearch("failure")`. On the other hand, if the requested book is available (denoted by [`result=1`]), the execution continues further and the client can make a purchase by providing information about shipment address and payment, and so on. The pre-conditions such as [`result=0`] are guards in the LTS representation and correspond to constraints between the variables. Note that the absence of a guard on a transition implies that the guard is `true` (i.e., always enabled).

Semantics of Labeled Transition Systems. The semantics of an LTS is given with respect to substitutions of variables present in the system. A state represented by the term s is interpreted under substitution σ ($s\sigma$). A transition $s \xrightarrow{\gamma,\alpha} t$, under *late semantics*, is said to be *enabled* from $s\sigma$ if $\gamma\sigma = tt$. The transition under substitution σ is denoted by $s\sigma \xrightarrow{\alpha\sigma} t\sigma$.

Such *late semantics* form a natural interpretation of LTSs by capturing the substitutions of input-variables at the destination state of a transition. For instance, consider an input transition of the form $s \xrightarrow{?\mathtt{m}(\vec{x})} t$. From the definition of LTS, $\vec{x} \cap vars(s) = \emptyset$. A consequence of late semantics is that if t contains elements in $\vec{x}$, their valuations are left to be interpreted by the guards in the subsequent transitions.

3.1.2 Equivalence of Labeled Transition Systems

Within a services computing environment, typically there exists multiple services which provide the same functionality and have the same behavioral description. Consequently, it might be of interest to a client to substitute an existing service S, with which it is interacting, with an alternate service S' depending on fulfillment of extra-functional requirements (e.g., it might be economically viable to replace S with S' if both provide the same functionality and have the same conversational or interaction model, but S' is cheaper than S to use).

To determine such "similarity" between services, we introduce two variants of *equivalence of LTSs*: *strong equivalence* (or bisimulation) and *weak equivalence* (or simulation), which identify equivalent LTSs in the presence of guarded transitions with input/output actions, atomic actions and unobservable actions τ. We define both the variants in the following.

Definition 4 (Strong Equivalence) *Given an LTS* $= (S, \longrightarrow, s0, S^F)$*, the strong equivalence (or bisimulation) relation with respect to substitution* θ*, denoted by* $\approx^\theta$*, is a subset of* $S \times S$ *such that:*

$$s_1 \approx^\theta s_2 \;\Rightarrow\; (\forall(s_1\theta \xrightarrow{\alpha_1\theta} t_1\theta) : \exists(s_2\theta \xrightarrow{\alpha_2\theta} t_2\theta : \forall\sigma : (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \wedge t_1 \approx^{\theta\sigma} t_2) \wedge\; s_2 \approx^\theta s_1)$$

Definition 5 (Weak Equivalence) *Given an LTS* $= (S, \longrightarrow, s0, S^F)$*, the weak equivalence (or simulation) relation with respect to substitution* θ*, denoted by* $\sim^\theta$*, is a subset of* $S \times S$ *such that:*

$$s_1 \sim^\theta s_2 \;\Rightarrow\; (\forall(s_1\theta \xrightarrow{\alpha_1\theta} t_1\theta) : \exists(s_2\theta \xrightarrow{\alpha_2\theta} t_2\theta : \forall\sigma : (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \wedge t_1 \sim^{\theta\sigma} t_2))$$

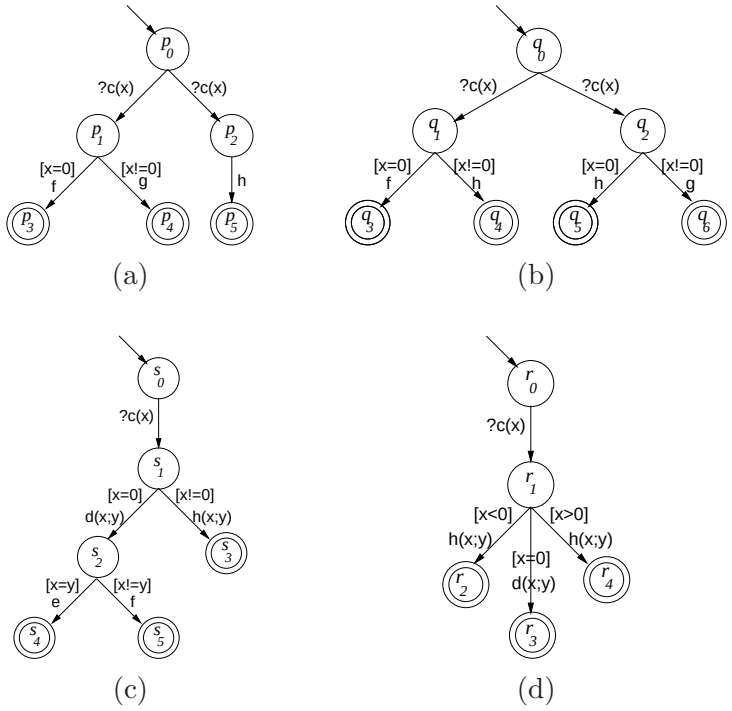


Figure 3.2 Example Labeled Transition Systems: (a) LTS_1 (b) LTS_2 (c) LTS_3 (d) LTS_4

In the above definitions, $s_2\theta \xrightarrow{\alpha_2\theta} t_2\theta$ denotes transitive closure of transitions over τ transitions, i.e., a transition may contain zero or more τ transitions preceding and following action α_2. Furthermore, α can be an ϵ or empty transition. Two states are said to be equivalent with respect to bisimulation (simulation), under the substitution θ, if they are related by the *largest* bisimilarity (similarity) relation $\approx^{\theta}$ ($\sim^{\theta}$). Two LTSs are said to be bisimulation (simulation) equivalent if and only if their start states are bisimilar (similar).

For example, consider checking the bisimilarity of states p_0 and q_0 in the the LTSs given in Figures 3.2(a) & 3.2(b), respectively. The state p_1 is bisimilar to q_1 when $x = 0$, and is bisimilar to q_2 when $x \neq 0$. Similarly, p_2 is bisimilar to q_1 when $x \neq 0$, and is bisimilar to q_2 when $x = 0$. However, p_0 and q_0 are not bisimilar as the input action $?c(x)$ from p_0 to p_1, if matched with input action $?c(x)$ from q_0 to q_1, demands that p_1 and q_1 are bisimilar for all possible valuations of x (i.e., for both $x = 0$ and $x \neq 0$). On the other hand, states s_0 and r_0

in the the LTSs given in Figures 3.2(c) & 3.2(d), respectively, are equivalent with respect to the simulation relation, since s_0 is simulated by r_0 and s_1 is simulated by t_1 for all possible valuations of x.

3.1.3 Composition of Labeled Transition Systems

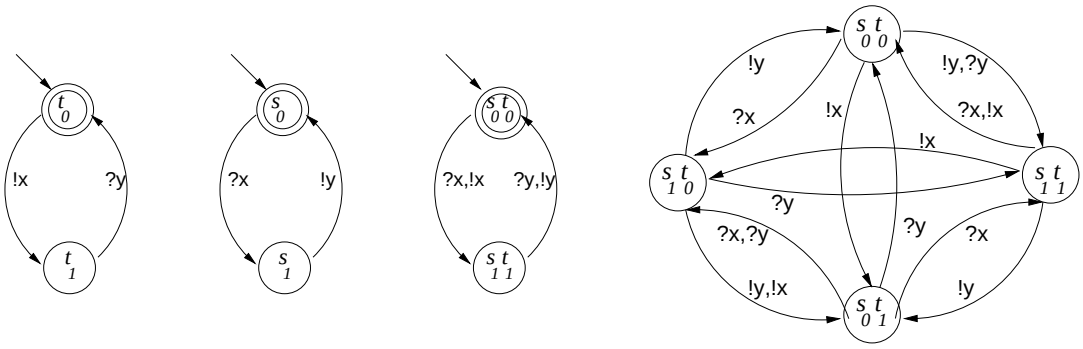
As mentioned earlier, a client can take the form of either an human agent or another service itself. Irrespective of the client representation, the interaction between the client and the service takes place primarily by exchange of messages. For example, the client might send a message to a service requesting invocation of a particular atomic action and expect to receive to an appropriate message from the service as an outcome of the invocation. Such a conversation model between interacting services is described using the notion of a *composition*, which models the fact that both the interacting parties may evolve independently as a consequence of the conversation process and communicate via exchange of messages.

Definition 6 (Composition) *Given two labeled transition systems* $LTS_1 = (S_1, \longrightarrow_1, s0_1, S_1^F)$ *and* $LTS_2 = (S_2, \longrightarrow_2, s0_2, S_2^F)$*, their composition, under the restriction set* L*, is denoted by* $(LTS_1 \parallel LTS_2)\backslash L = (S_{12}, \longrightarrow_{12}, s0_{12}, S_{12}^F)$ *where* $S_{12} \subseteq S_1 \times S_2$*,* $s0_{12} = (s0_1, s0_2)$*,* $S_{12}^F = \{(s_1, s_2) \mid s_1 \in S_1^F \wedge s_2 \in S_2^F\}$ *and* $\longrightarrow_{12}$ *relation is of the form:*

1. $s \xrightarrow{g_1, ?m(\vec{x})} s' \wedge\ t \xrightarrow{g_2, !m(\vec{x})} t' \wedge\ m \in L \ \Rightarrow (s,t) \xrightarrow{g_1 \wedge g_2, \tau} (s', t')$,
2. $s \xrightarrow{g_1, \alpha} s' \wedge header(\alpha) \notin L \ \Rightarrow (s,t) \xrightarrow{g_1, \alpha} (s', t)$, *and*
3. $t \xrightarrow{g_2, \alpha} t' \wedge header(\alpha) \notin L \ \Rightarrow (s,t) \xrightarrow{g_2, \alpha} (s, t')$.

In the above, restriction set L includes the message headers on which the participating LTSs must synchronize and generate a τ action. We use $header(\alpha)$ to return the message header of input and output actions; for atomic actions and τ-actions it returns a constant which is never present in L.

For example, Figure 3.3(c) shows the composition LTS_c of LTS_5 and LTS_6 (Figures 3.3(a) and 3.3(b), respectively), where $L = \{x, y\}$. On the other hand, if L is `null`, i.e., the restriction



(a) (b) (c) (d)

Figure 3.3 Composition of Labeled Transition Systems: (a) LTS_5 (b) LTS_6 (c) LTS_c (d) $LTS_{c'}$

set is empty, then the composition LTS_5 and LTS_6 can be represented by $LTS_{c'}$ as shown in Figure 3.3(d).

We discuss more on composition of LTSs in Chapter 4.

3.2 Transforming Web Service Descriptions to Labeled Transition Systems

Labeled Transition Systems, even though are adequate enough to represent the behavioral description of Web services, are not widely used in the services computing domain because modeling LTSs for complex Web services is a time consuming, cumbersome and error-prone process. Furthermore, the tool support for developing LTS descriptions is inadequate. Consequently, in our framework we allow service developers to model services using state machines [64] and Business Process Execution Language (BPEL) [16], which are widely used service development languages with robust open-source and commercial tooling support. The service descriptions modeled in these languages can be mapped to LTS representations (Sections 3.2.1 & 3.2.1) and are automatically automatically translated by our system (see Chapter 8 for more details).

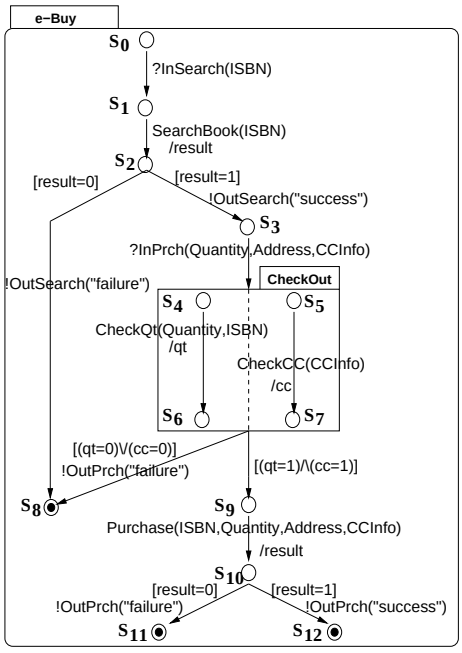


Figure 3.4 State Machine representation of the `e-Buy` service

3.2.1 Mapping State Machines to Labeled Transition Systems

A state machine [64] is a model of behavior composed of a set of states $(s_1, s_2, \cdots)$ representing an abstraction of the system configuration, and inter-state transitions $(s_1 \longrightarrow s_2)$ denoting the conditions under which the system evolves from one state to the next. The states can be either *composite* (or-/and- states) or *atomic*. A composite state is an "or-state" if there exists multiple transitions originating from the state such that any one of these transitions can be executed; whereas the composite is an "and-state" if all the transitions can be executed simultaneously.

Each transition, `source` $\stackrel{ev[g]/e}{\longrightarrow}$ `destination`, is annotated with *action* labels consisting of an *event* (ev), *guard* (g), and *effect* (e). In the context of Web services, the events correspond to various functions (i.e., atomic actions) that a service provides; the guards refer to pre-conditions of those functions; and effects correspond to post-conditions of the transition-functions, in essence denoting the possible assignment of values to variables after the function is executed. A `true` guard and ϵ (empty) effect denote the absence of pre-/post-conditions, respectively. For example, Figure 3.4 shows the state machine representation of the `e-Buy`

service described in Example 1. Here, for the transition $s_1 \rightarrow s_2$, the *event* corresponds to the function `SearchBook(ISBN)`, the *guard* is assumed to be `true`, and the *effect* refers to assigning some value to the variable `result`. We assume that the control flow of the service represented in a state machine is dependent entirely on the guards augmenting the corresponding transitions. That is, if there exists multiple transitions originating from a particular state, then the branching behavior is based on how the guards are analyzed. For instance, in Figure 3.4, there are two transitions originating from state s_2 namely, $s_2 \overset{[result=0]}{\longrightarrow} s_8$ and $s_2 \overset{[result=1]}{\longrightarrow} s_3$. Thus, the transition from state s_2 to s_8 is executed only when $[result = 0]$, whereas the transition from s_2 to s_3 is executed when $[result = 1]$. Furthermore, we assume that if there is no correlation between the order in which the events (or functions) are executed (i.e., the functions can be invoked in parallel since there is no dependency between them), they are represented using "and-composite" states in the state machine model.

In our context, a state machine representation can be translated to its corresponding LTS form as follows: (a) an LTS-state corresponding to an "and-state" is determined by all the active atomic states, (b) an LTS-state for an "or-state" corresponds to one of the possible active states, (c) states outside the scope of any "and-/or-composition" are also states in the LTS, and finally, (d) initial and end states along with their transitive closures over event-free transitions are start and final states, respectively, of the LTS. Figure 3.1 shows the LTS representation of the `e-Buy` state machine shown in Figure 3.4.

3.2.2 Mapping BPEL to Labeled Transition Systems

We introduced Business Process Execution Language (BPEL) in Section 2.1. It is one of the most widely used languages for modeling Web services and their compositions since it offers an uniform semantics for creation of complex processes by integrating different activities that can, for example, perform service invocations, process data, throw exceptions, or terminate execution. These activities may be nested within structured activities that define how they may be run, such as in sequence, or in parallel, or depending on certain conditions.

Central to the theme of BPEL is a *process* which is used to model the behavioral description

of a composition of a set of services or even a single service. The process either receives (sends) messages from (to) various interacting entities, which are called *partners*. Thus, a partner is either a service the process invokes (*invoked partners*) as an integral part of its algorithm, or those that invoke the process (*client partners*). As obvious, when modeling a composition, the process is going to interact and invoke multiple partners. For the sake of simplicity, in the remaining discussion, we will only consider BPEL processes that interact with two partners: a client partner and an invoked partner.

In an abstract sense, a BPEL process is a flow-chart representation of an algorithm. Each step in the process is called an *activity*. Some of the basic activities in BPEL include:

- <`receive`>: this activity corresponds to reception from a service due to an invocation (and subsequent execution) of an operation in its interface.
- <`reply`>: this activity corresponds to generation of a response due to invocation of an input-output operation in the service interface.
- <`invoke`>: this activity corresponds to invoking an operation in the service interface.
- <`wait`>: this activity corresponds to waiting period for a fixed amount of time.
- <`assign`>: this activity corresponds to copying data from one place (or variable, or message) to another.
- <`throw`>: this activity indicates the occurrence of a problem.
- <`terminate`>: this activity corresponds to terminating a service instance.
- <`empty`>: this activity corresponds to doing nothing.

These primitive activities can be combined to form more complex ones akin to constructs in structured-programming. Some examples include the ability to define an ordered sequence of steps (<`sequence`>), the ability to have branches using "case-statements" (<`switch`>), the ability to define a loop (<`while`>), the ability to execute one of several alternative paths (<`pick`>), and finally the ability to indicate that a collection of steps should be executed in

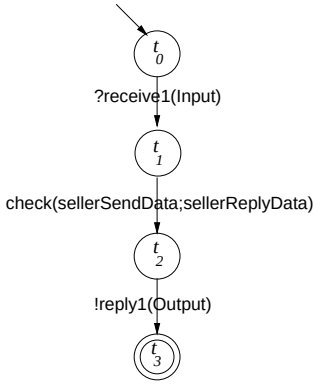


Figure 3.5 Labeled Transition System representation of `e-Auction` service

parallel (<`flow`>). In addition to above, BPEL provides a range of complex activities for message correlation, fault handling, and compensation. For our purpose, we will only consider some of the primitive activities along with their complex variants.

As an example, consider a simple service named `e-Auction` which allows clients to query for the price of an item that has been put on auction. The BPEL process describing the behavioral model and the WSDL file describing the interface of `e-Auction` are shown in Appendices A and B, respectively. The BPEL process essentially says that the execution of an instance of `e-Auction` service is (*i*) able to receive a message from an external client containing the name of the item whose price the client is interested in knowing, (*ii*) upon reception of the message, the data (i.e., item name) from the message is assigned/copied to another (local) message, (*iii*) this (local) message is used to invoke the `check` operation of the `e-Auction` service for checking the price, (*iv*) the output information (containing the price of the item) provided by `check` is then assigned to another (local) message, (*v*) which is finally sent back to the client as a reply. The corresponding LTS representation of this BPEL process model is shown in Figure 3.5. As can be noticed, the <`receive`> and <`reply`> activities in the BPEL process correspond to input (*?receive1(Input)*) and output (*!reply1(Output)*) actions in the LTS, respectively. The <`assign`> activity corresponds to assigning the variables and their values from the transition label to the destination state of the transition (not shown in Figure 3.5). And finally, the <`invoke`> activity corresponds to the atomic action *check(sellerSendData;sellerReplyData)* in the LTS representation. All these activities in the BPEL are part of the <`sequence`> activity.

We have developed a tool which does this translation from BPEL files to their corresponding LTS representation. Refer to Chapter 8 for more details.

3.3 Discussion

In this chapter, we have presented a general framework for formally representing the behavioral model of a service. In particular, we used labeled transition systems (LTSs) for the behavioral representation which comprises of a finite number states and transitions between the states that are annotated with actions or guards (i.e., the pre-conditions on the actions). LTSs have been used and researched widely in various communities such as software engineering, distributed systems and networks, and are adequate to model Web services behavior since they (*i*) provide a natural way to represent message exchange patterns in terms of input and output actions, (*ii*) provide an abstract representation of a concrete service realization, and (*iii*) possess well-defined formal semantics. We also introduced two key ideas, that are in particular relevant to this thesis, namely, determining equivalence between LTSs and composition of multiple LTSs. Equivalence of LTSs is particularly important in a highly dynamic service-computing environment because existing services can become unavailable/obsolete or newer services might become available quite frequently, thereby requiring the client the replace an existing service with another "functionally-equivalent" service. On the other hand, composition of LTSs is also a vital problem since in many cases the desired functionality cannot be provided by a single service, but possibly by integrating multiple services in a suitable way. We discuss more on composition and equivalence of LTSs in Chapters 4 and 6, respectively. In addition to the above, we introduced techniques and developed tools for mapping existing service description and modeling languages to the LTS representation. In particular, we addressed translation of graphical languages such as state machines and XML-based languages such as BPEL to LTSs. This will in turn pave the way for using existing services specified in those languages in our framework (see Chapter 8).

However, there are a few limitations of our approach. For instance, when translating BPEL specifications to LTS representations, we consider only the primitive and few complex activ-

ities. Our current implementation cannot handle, for example, correlation between messages or compensation to failures—aspects that are important and relevant in service modeling. Similarly, one can argue that translation of state machines to LTSs is too limited in terms of handling complex structure-programming constructs such as exception or failure handling. Although a part of this criticism can be attributed to the semantics of state machines (i.e., state machines formally do not have notion of "exception handling"), we believe that further research is required for realizing important benefits of model-driven architectures in service oriented computing environments. Furthermore, in the current setting, we represented the behavioral model of Web services using LTSs, which are discrete-event systems. In other words, every event, whether receiving a message or invoking an atomic action, occurs at a discrete time step. However, in many cases, the actions present in a service may be related by continuous temporal intervals (e.g, execution of action `a` occurred during action `b`), which cannot be represented by LTS. Towards this end, investigation of temporal algebra-based representations [9] and their applicability within our setting is needed.

CHAPTER 4. WEB SERVICE COMPOSITION

This chapter introduces the problem of Web service composition and describes in details our approach to address the problem. The chapter is divided into four sections. The first section provides background to and describes the problem of Web service composition. An illustrative example is described in the second section to explain the salient features of our proposal, which is discussed in the third section. The fourth section concludes the chapter with a discussion.

4.1 Introduction and Problem Description

Recent advances in networks, information and computation grids, and WWW have resulted in the proliferation of physically distributed and autonomously developed software components and services. These developments allow us to rapidly build new value-added applications from existing ones in various domains such as e-Science, e-Business, and e-Government. However, often the process of integrating applications becomes tedious and time consuming because individual software entities are not developed using standard frameworks or component models. This results in significant re-designing and re-coupling of existing software leading to substantial loss in productivity.

In this context, developing approaches for (semi-) automatic composition of Web services has emerged as an active area of research in both academia and industry. As introduced earlier in Section 1.2.2, there are two basic ways of modeling composite services: *orchestration*-based (Figure 1.4(a)), wherein message exchanges between the services participating in a composition is observed from a single point-of-view, and *choreography*-based (Figure 1.4(b)), wherein the message exchanges are observed from a global perspective. Many recent efforts (see Chapter

2 for a brief literature review and [67, 103, 128, 137, 170, 184] for surveys), that leverage techniques based on AI planning, logic programming, and formal methods have focused on different aspects of Web service composition ranging from service discovery to service specification and deployment of composite services. However, despite the progress, the current state of the art in service composition has several limitations:

- *Complexity of Modeling Composite Services:* For specifying functional requirements, the current techniques for service composition require the service developer to provide a specification of the desired behavior of the composite service (goal) in its entirety. Consequently, the developer has to deal with the cognitive burden of handling the entire composition graph (comprising appropriate data and control flows) which becomes hard to manage with the increasing complexity of the goal service. Instead, it will be more practical to allow developers to begin with an abstract, and possibly incomplete, specification that can be incrementally modified and updated until a feasible composition is realized.

- *Inability to Analyze Failure of Composition:* The existing techniques for service composition adopt a 'single-step request-response' paradigm for modeling composite services. That is, if the goal specification provided by the service developer cannot be realized by the composition analyzer (using the set of available component services), the entire process fails. As opposed to this, there is a requirement for developing approaches that will help identify the cause(s) for failure of composition and guide the developer in applying that information for appropriate reformulation of the goal specification in an iterative manner. This requirement is of particular importance in light of the previous limitation because in many cases the failure to realize a goal service using a set of component services can be attributed to incompleteness of the goal specification.

- *Inability to Analyze Infinite-State Behavior of Services:* Often, Web services have to cope with apriori unknown and potentially unbounded data domains (e.g., data types defined by users in WSDL documents). Analyzing the behavior of such a service requires

consideration of all possible valuations, which makes the resulting system infinite-state. However, most of the service composition approaches do not take into account the infinite-state behavior exhibited by Web services.

- *Lack of Formal Guarantees:* Formally guaranteeing the soundness and completeness of an algorithm for service composition is a vital requirement for ensuring the correctness of the composite service (generated by the algorithm). In this context, we say that an algorithm is complete if it is able to find a feasible composition whenever it exists, whereas the algorithm is sound if ascertains that the composition generated realizes the goal service. Nevertheless, most of the existing service composition techniques do not provide soundness and completeness guarantees.

Against this background, we propose *MoSCoE*[1] [146, 148, 149, 152, 153, 154, 155]—an approach for developing composite services in an incremental fashion through iterative reformulation of the functional specification of the goal service. MoSCoE accepts from the user (i.e., service developer), an *abstract* (high-level and possibly incomplete) specification of a goal service. In our current implementation, the goal service specification takes the form of a labeled transition systems (LTS) (Definition 3) that provide a formal, yet intuitive specification of the desired goal functionality. Similarly, the component services (i.e., available services) are also represented as LTSs. As mentioned in Section 3.2, the LTS representation of the services can be obtained either from the corresponding state machine or BPEL process description. Thus, given the goal service and component service specifications as input, the objective is to *compose* a subset of the available component services $(c_1, c_2 \cdots c_n)$ with the corresponding LTS representations $(LTS_1, \cdots, LTS_n)$, such that the resultant composition "realizes" the desired goal service LTS_g. As noted above, this process might fail either because the desired service cannot be realized using the available component services or because the specification of the goal service is incomplete. A novel feature of MoSCoE is its ability to identify, in the event of failure to realize a goal service arising from an incomplete goal specification, the specific

[1]MoSCoE stands for Modeling Web Service Composition and Execution. More information is available at: http://www.moscoe.org.

states and transitions in the LTS description of the goal service that require modification. This information allows the developer to *reformulate* the goal specification, and the procedure[2] can be repeated until a feasible composition is realized or the developer decides to abort. We refer to this approach as modeling Web services using *abstraction*, *composition* and *reformulation* and explain its salient aspects in details in the remainder of this chapter. We begin with the description of an illustrative example.

4.2 Illustrative Example

Assume that a service developer is assigned to model a new Web service, `Health4U`, which allows senior citizens to make a doctor's appointment to receive medical attention for a particular ailment. To achieve this, `Health4U` relies on five existing (possibly independent) services: `Appointment, MedInsurance, MedRecord, e-Ride` and `Validate`. `Appointment` accepts patient data (*name*, *ailment* s/he is suffering from) and scheduling information (preferred *date* and *time*) as input to make an appointment. `Appointment` takes into account: (a) information about patient's insurance coverage plan to identify the designated physicians from whom the patient can receive treatment, and (b) the medical history (if any) that provides information about patient's previous appointments for the particular ailment. To obtain the needed information, `Appointment` communicates with `MedInsurance` (case (a)) and `MedRecord` (case (b)), both of which require the patient's *SSN* (Social Security Number). `Appointment` attempts to schedule an appointment for the patient with a physician who has treated the patient in the past. If no such physician is available, it makes an appointment with a physician who is among those designated by the insurance provider. Furthermore, `Health4U` arranges transportation for the patient to the medical center via the `e-Ride` service. This service needs the *date* and *time* for pick-up, as well as the patient's *address*. In addition, `e-Ride` communicates with `Validate` to determine whether the patient has provided a valid payment information (e.g., credit card) before completing the reservation.

We discuss in the next section how the composition of a service like `Health4U` can be

[2]Note that determination of the cause for failure of composition, and use of that information for reformulation of the goal specification is carried out at *design-time* as opposed to *run-time*.

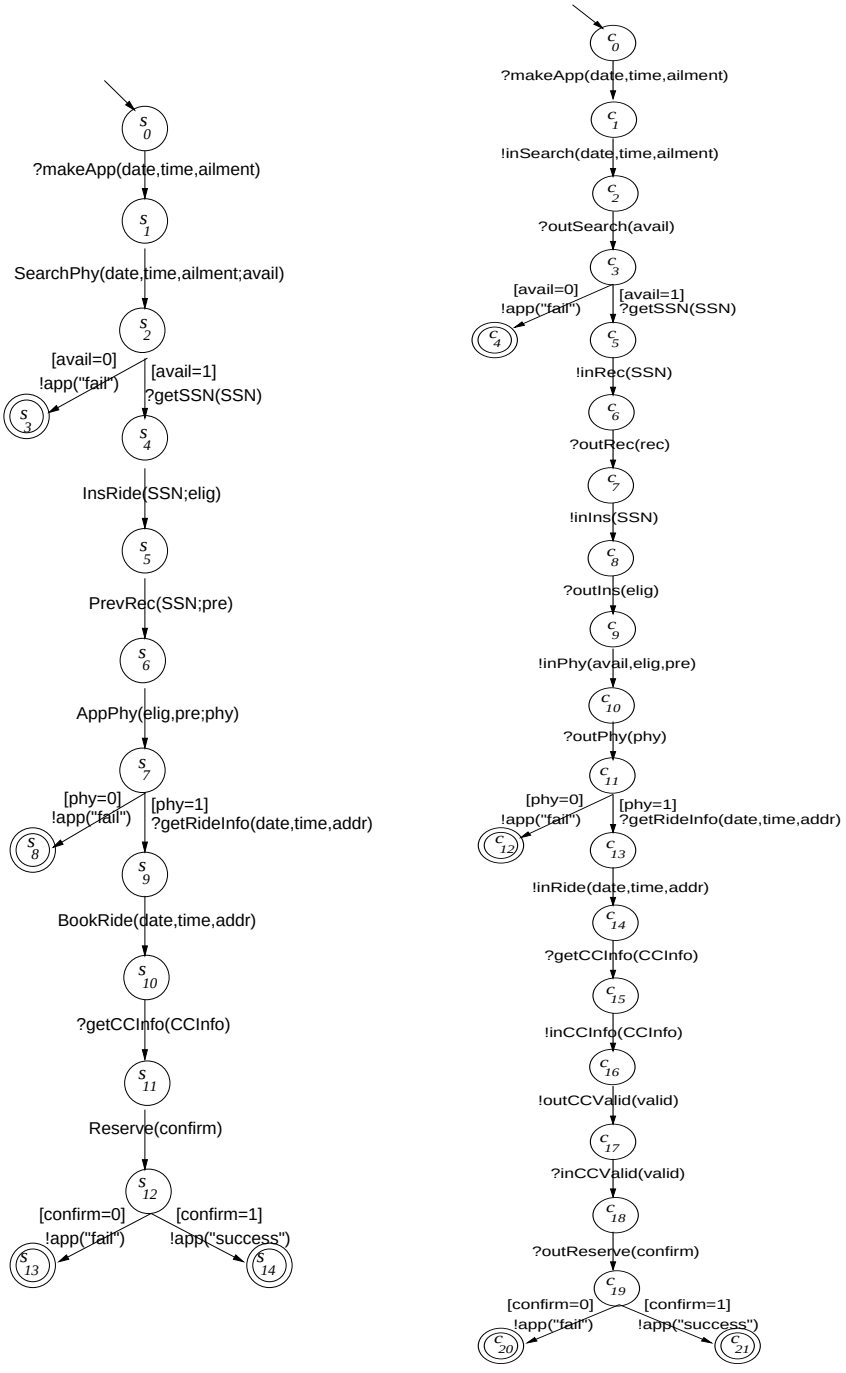


Figure 4.1 LTS representation of (a) `Health4U` (b) The Mediator

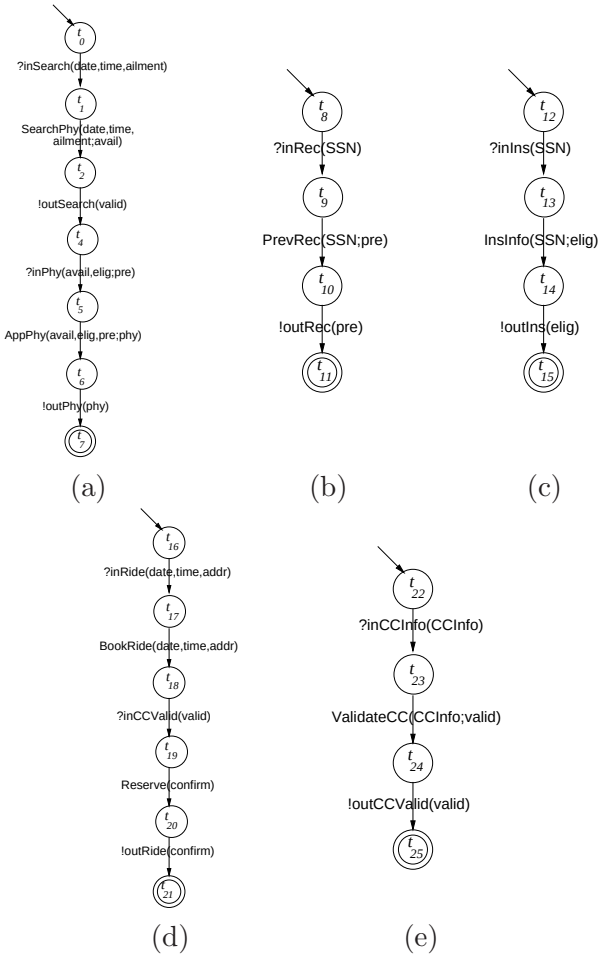


Figure 4.2 LTS representation of (a) `Appointment` (b) `MedRecord` (c) `MedInsurance` (d) `e-Ride` (e) `Validate` component services

accomplished by MoSCoE, which adopts the *orchestrator*-based model for composition. In particular, MoSCoE receives from the service developer an LTS specification of the desired goal service `Health4U,` as shown in Figure 4.1(a), which is used to construct a *mediator* that enables the interaction between (a subset of) the component services to provide the desired goal service functionality. Figure 4.1(b) shows a mediator that realizes `Health4U` using component services shown in Figure 4.2.

4.3 Our Approach

4.3.1 Service Composition in MoSCoE: An Overview

Given a goal service T_g and a set of available component services $T_1, T_2, \ldots, T_n$, solving the service composition problem entails identifying a composition of the necessary component services that realizes the functionality of T_g. In the setting of orchestration-based composition (Figure 1.4(a)), this entails generating a mediator T_M which realizes the functionality of T_g by orchestrating the necessary interactions among the selected component services. As noted earlier, the mediator T_M replicates the behavior of the input/output actions of the goal service and is responsible for communications between component services; it relies on the component services for atomic actions needed to realize the goal service. In MoSCoE, the operation of the goal service as well as the component services are represented by the corresponding LTSs.

Based on the definition of composition and equivalence on LTSs described in Section 3.1, and the previously introduced notion of a mediator, the service composition problem can be described as:

$$\exists T_M : (\ldots((T_M||T_i)||T_j)||\ldots||T_k)\backslash L \approx^{tt} T_g$$

where, L contains all the input and output message headers of the component services. Thus, solving the service composition problem entails to constructing a mediator which can enable interaction between the component services so as to yield a behavior that is *strong equivalent* (bisimilar) to that of the desired goal service.

4.3.2 Algorithm for Mediator Synthesis

We now proceed to describe an algorithm for constructing a mediator for a desired service from a set of component services. Since the goal service specification includes the descriptions of the desired *functions*, we select the subset of component services whose LTSs provide the necessary atomic actions to yield a set of candidate component services which the mediator can work with.

Because the task of a mediator is to orchestrate the interactions among component services,

the algorithm for constructing the mediator requires information regarding dependencies between components, i.e., the dependency of an input message of a component on the output of another. For example, if a component T_i requires an input of the form `?m(`$\vec{x}$`)` and a component T_j provides an output of the form `!m(`$\vec{x}$`)`, we say that T_i is dependent on T_j via the message header `m`. In such a setting, the mediator needs to synchronize with the output message from T_j and pass on the output of T_j as an input message to T_i. To make this notion of dependency more precise, we define *flow links* which capture the dependencies between multiple component services.

Definition 7 (Flow Links) *For services T_i and T_j, if `?m(`$\vec{x}$`)` and `!m(`$\vec{x}$`)` are present in the specifications of the respective components `STS`$_i$ and `STS`$_j$, then `m` is said to be a member of the flow link (from j to i component) set denoted by FL_{ij}.*

For example, consider the component services `e-Ride` (Figure 4.2(d)) and `Validate` (Figure 4.2(e)). In order for `e-Ride` to reserve a ride, it needs valid payment information. This information is provided by `Validate` after it validates the credit card information provided by the patient. Hence, there exists a flow link from `Validate` to `e-Ride`.

The algorithm for modeling a mediator (Algorithm 1) that is "equivalent" to the goal service works as follows: the procedure GENERATE$(r, [s_1, s_2, \ldots, s_n], t, G, R)$ is invoked by providing the start states of the goal LTS (r), the component LTSs in $\mathcal{S}$ $(s_1, s_2, \ldots, s_n)$, and the mediator LTS (t) that is being modeled. The initial guard condition G is set to `true` and R corresponds to a store that contains all the input and output message headers of the component services, which is initially empty. A global set *done* is used to keep track of whether a particular atomic action requested by the goal service is realized in the composition. There are four cases to consider:

Case 1: If the transition from the current state r in the goal LTS to state r' has an input action, i.e., receiving a message from the client, then a corresponding transition with the input action is created in the mediator (`line 8`) and R is updated with the `msgSet` of the input action (`line 9`). The procedure GENERATE is recursively invoked in `line 10`.

```
/*
- r is the goal state; s_i is the component state; t is the generated mediator state.
- G is the conjunction of guard conditions that will be accumulated along each DFS path. All variables in G are
  universally quantified.
- R is a store that contains all the input & output message headers of the component services.
*/
1:  procedure GENERATE(r, [s_1, s_2, ..., s_n], t, G, R)
2:    if (!visited(r, [s_1, s_2, ..., s_n], t, G, R)) then // Traverse path for the first time.
3:      mark as visited(r, [s_1, s_2, ..., s_n], t, G, R);
4:    end if
5:    for all ((r -g,a-> r') && (G ∧ g)) do
6:      case 1: /* input action from the client */
7:        if (a = ?m(x⃗)) then
8:          create a transition t -g,a-> t';
9:          R := R ∪ x⃗;
10:         call GENERATE(r', [s_1, s_2, ..., s_n], t', G ∧ g, R ∪ x⃗);
11:       end if
12:     end case

13:     case 2: /* output to the client */
14:       if (a = !m(x⃗)) then
15:         if (x⃗ ∈ R) then
16:           create a transition t -g,a-> t';
17:           call GENERATE(r', [s_1, s_2, ..., s_n], t', G ∧ g, R);
18:         else Requested output cannot be created for client. Return partial mediator.
19:         end if
20:       end if
21:     end case

22:     case 3: /* atomic action to be provided by the components */
23:       if ((a = funcName(I; O)) && (no s_i has a transition on the action a)) then
24:         select the component T_i that is capable of generating the function;
25:       end if
26:       if ((s_i -g_i,?m(x⃗)-> s_i') && (x⃗ ∉ R)) then
27:         if (m ∈ FL_ij) then
28:           msgH:=m; k := j;
29:         else Return partial mediator. Failure at action a.
30:         end if
31:         while ((s_k -g_k,a_k-> s_k') && header(a_k) ≠ msgH) do
32:           if ((a_k = ?m_k(y⃗)) && (y⃗ ∉ R)) then
33:             if (m_k ∈ FL_kl) then
34:               msgH := m_k; k := l;
35:             end if
36:           else if (((a_k = ?m_k(y⃗)) && (y⃗ ∈ R)) || (a_k = !m_k(y⃗))) then
37:             if (G ⇒ g_k) then
38:               create transition t -G,a_k-> t' to communicate with s_k;
39:               call GENERATE(r, [s_1, s_2, ..., s_k', ..., s_n], t', G, R ∪ y⃗);
40:               if (t' is the root of a partial mediator) then
41:                 select next transition from s_k;
42:               else
43:                 break;
44:               end if
45:             end if
46:           else
47:             Return partial mediator. Failure at action a.;
48:             break;
49:           end if
50:         end while
```

Algorithm 1 Algorithm for Modeling the Mediator & Failure-Cause Detection

```
51:          if ((s_k --g_k,a_k--> s'_k) && (header(a_k) = msgH)) then
52:            if (G ⇒ g_k) then
53:              create transition t --G,¬a_k--> t' to communicate with s_k;
54:              call GENERATE(r, [s_1, s_2, ..., s'_k, ..., s_n], t', G, R ∪ vars(a_k));
55:            else
56:              Return partial mediator. Failure at action a.;
57:            end if
58:          else if ((s_k ∉ S_k^F) || (funcName(I;O) ∉ done)) then
59:            Return partial mediator. Failure at action a.;
60:          elsereturn;
61:          end if
62:        else if ((s_i --g_i,?m(x)--> s'_i) && (x ∈ R) && (G ⇒ g_i)) then
63:          create transition t --G,!m(x)--> t' to communicate with s_i;
64:          call GENERATE(r, [s_1, s_2, ..., s'_i, ..., s_n], t', G, R);
65:        else if ((s_i --g_i,!m(x)--> s'_i) && (G ⇒ g_i)) then
66:          create transition t --G,?m(x)--> t' to communicate with s_i;
67:          call GENERATE(r, [s_1, s_2, ..., s'_i, ..., s_n], t', G, R ∪ x);
68:        else
69:          Return partial mediator. Failure at action a.;
70:        end if
71:      end case

72:      case 4: /* atomic action to be provided by the components */
73:        if (a = funcName(I; O)) && (s_i has a transition on action a)) then
74:          if ((s_i --g_i,a--> s'_i) && (G ∧ g ⇒ g_i)) then
75:            done = done ∪ funcName(I;O);
76:            call GENERATE(r', [s_1, s_2, ..., s'_i, ..., s_n], t, G ∧ g, R ∪ ovars(a));
77:          else Return partial mediator with failure at guarded action (g,a).
78:          end if
79:        end if
80:      end case
81:    end for
82: end procedure
```

Case 2: If the transition from the current state r in the goal LTS to state r' has an output action, i.e., transmitting a message to the client, then a corresponding transition with the output action is created in the mediator if the `msgSet` of the action is already present in R (`line 16`). Note that here the `msgSet` required to produce the output message can be only retrieved from R (assuming it was placed there as a result of preceding interactions between the component services). The procedure GENERATE is recursively invoked in `line 17`.

Case 3: This case corresponds to a situation in which the transition action in the goal is an atomic action a and none of the component services can provide a transition on that action from their current states s_i. In such a scenario, the algorithm first selects a component service T_i which can provide the required function a (`line 24`)[3]. Now there are three scenarios: s_i has an input action for which the mediator cannot provide input messages (`line 26`); s_i has an input action for which the mediator can provide input messages (`line 62`); and s_i has an output action (`line 65`).

The last two of the preceding three scenarios are easily dealt with: the mediator transitions are generated to provide appropriate output or input message as the case may be and the procedure GENERATE is invoked recursively. Thus, in the last case, i.e., `line 65`, the store R is updated to include the output messages from the state s_i. The first scenario (`line 26`) is more involved. As the `msgSet` required at the input action from state s_i is not present in R (`line 26`), the flow links (Definition 7) are explored to determine a component T_j which can provide the message as output. However, it is possible that T_j, in turn, is at a state s_j which needs a different input or output message. If the message is on input action provided by the mediator or if the message in on output action, then appropriate mediator transition is created and GENERATE is invoked recursively (`lines 36--45`). At `line 38`, $\overline{a_k}$ denotes the complement of a_k, i.e. $\overline{a_k} := !\mathtt{m}_k(\vec{y})$ if $a_k = ?\mathtt{m}_k(\vec{y})$; otherwise $\overline{a_k} := ?\mathtt{m}_k(\vec{y})$. In this case, after the recursive call to GENERATE, a new transition from s_k is selected at the while-condition (`line 31`). If the input message at s_j cannot be provided by the mediator another component via

[3]In practice, there might be more than one component service that can provide the required atomic action a, in which case, each choice is explored to find a feasible mediator.

flow link is selected and the process is iterated (`lines 31--34`).

Outside the `while` loop, if there exists a component which has the output action at its current state (s_k in Figure at `line 51`) required by the input action at state s_i of T_i responsible for providing the atomic action (`lines 24--30`), then the mediator transition communicating with this component (`line 53`) is generated. Finally, at `line 58--60`, if the state s_k is not a final state or the global store *done* does not include `funcName(I;O)`, i.e., there exists a transition with atomic action from s_k (fall-through case from `lines 31, 51`) or `funcName(I;O)` requirement is not provided along any of the paths by recursion, then failure is reported; otherwise the procedure returns with no error.

Case 4: Finally, this case considers a situation when the transition action in the goal is an atomic action a and there exists a component T_i which has a transition from its current state s_i on action a (`line 72--80`). The message store R is updated with the return values of the function and global store *done* is updated to reflect that `funcName(I;O)` invocation requirement is realized.

We use a constraint solver to check the (un)-satisfiability of guards on LTS transitions. All the variables in the guard are universally quantified. At present, MoSCoE works with only equality and disequality constraints on infinite domain variables for which satisfiability checking of guards is decidable [23], although we plan to investigate a of larger classes of infinite state systems for which the construction of mediator can be made decidable [109]. The preceding algorithm may fail to construct a mediator because of either due to the absence of an action that is necessary to achieve the goal service functionality or the unsatisfiability of guards. Analysis of the cause of such failure is discussed in Section 4.3.3.

4.3.2.1 Modeling a Mediator for `Health4U`

In what follows, we show how to model a mediator for the `Health4U` composite service introduced in Section 4.2 using the formal framework and algorithm described above. Figure 4.1(a) shows an LTS representation of the `Health4U` goal service and Figure 4.2 shows the corresponding LTSs of a set of available services. Given the goal service specification and a

set of available component services, MoSCoE's task is to construct a mediator (Figure 4.1(b)), which enables the interaction between the client and component services, and is "bisimulation equivalent" to the goal service.

The algorithm begins with the start state s_0 of the goal LTS and considers its transition to state s_1. Here, the transition takes place due to an input action `?makeApp(...)` from the client (Case 1), so MoSCoE creates an appropriate transition ($c_0 \longrightarrow c_1$) in the mediator to receive the input message. For the transition $s_1 \longrightarrow s_2$ in the goal STS, the associated transition label is an atomic action (`SearchPhy(...)`). However, since none of the current component states ($t_0, t_8, t_{12}, t_{16}, t_{22}$) can make a transition on this action (Case 3), the algorithm first selects the component `Appointment` because it can provide the requested function, and then creates an appropriate transition in the mediator to send a message to `Appointment`. Once `Appointment` executes the function `SearchPhy(...)`, it transmits an output message (in this case, indicating the availability of physician(s) for treatment of the ailment on the requested date and time), which is received by the mediator. This behavior is modeled by the mediator by the transition $c_2 \longrightarrow c_3$ (Case 1). Depending on whether a physician is available or not, the algorithm creates transitions $c_3 \longrightarrow c_4$ and $c_3 \longrightarrow c_5$ to send/receive output/input message to/from the client (Cases 2 & 1), respectively. The algorithm proceeds in a similar fashion to model transitions for atomic actions `InsInfo(...)` and `PrevRec(...)`, and reach the goal state s_6 and mediator state c_9. Now, to model a corresponding transition for the atomic action `AppPhy(...)`, the mediator refers to the message store R for previous message exchanges between the client and component services, and generates an output message `!inPhy(avail,elig,pre)`. Note that the values for the variables (`avail, elig, & pre`) in the message were placed in R as a result of previous message exchanges between the mediator and component services. Since R contains every message that the mediator receives from the client and the component services, to select the relevant components (and their messages), the mediator exploits the flow links (Definition 7) between the components, as illustrated in Case 3 of the algorithm. This process for constructing the mediator terminates with success when for each transition leading to a final state in the goal, a corresponding transition in the mediator is established.

Now we proceed to discuss the scenario in which the algorithm for constructing the mediator fails.

4.3.3 Analysis of Failure of Composition

Algorithm 1 for constructing a mediator that realizes a specified goal service using the available component services fails when some aspect of the goal specification cannot be realized using the available component services. In the event of such failure, MoSCoE seeks to provide to the user (i.e., the service developer) information about the cause(s) of the failure in a form that can be used to reformulate the goal specification. Recall that mediator construction fails when there exists no mediator that can enable the interaction among the available components to realize a behavior that is "bisimulation equivalent" to that of the goal service. In particular, bisimulation equivalence is not satisfied when:

1. The mediator composed with components fails to create the transition relation (see bisimulation in Definition 4). These transitions are generated by transitive closure of τ-transitions obtained via synchronization between mediator and components.

2. The actions between the goal and component transitions do not match.

3. The guard conditions are unsatisfiable.

Returning to the mediator construction algorithm (Algorithm 1), we note that failures might be encountered during different stages of execution of the algorithm. For instance, `line 18` might result in a failure cause corresponding to Case 1 because the messages required for generating the output message to the client are not present in R. Similarly, in `lines 29` and `47` the failures might arise because either the input message required by a component services cannot be provided by some other component service or by the client itself. In `line 56, 77`, failure might occur because the guard conditions do not hold (the guards on the component transition are stronger than those on the goal). Finally, a failure could occur when there is a mismatch between an action that is required by the goal and actions that are provided by the available components (see `lines 59, 69`).

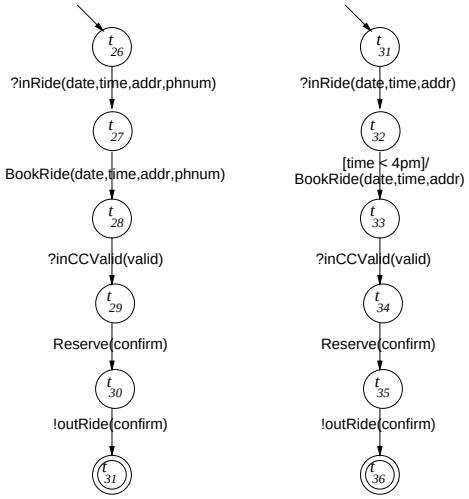


Figure 4.3 LTS representation of (a) `e-Ride`' (b) `e-Ride`"

4.3.3.1 Failure Cause Analysis for `Health4U`

In our example from Section 4.2, suppose we replace the `e-Ride` component service (Figure 4.2(d)) with component services `e-Ride`' and `e-Ride`" yielding two separate instances of the `Health4U` composition problem (Figure 4.3)(a) & 4.3(b)). Suppose the behavior of `e-Ride`' is exactly the same as that of `e-Ride`, but it additionally requires a phone number to reserve a ride. Suppose on the other hand that `e-Ride`" can only reserve a ride if the time for pick-up is before 4pm. Note that in both these instances, the algorithm for constructing the mediator fails when it encounters the transition $s_9 \longrightarrow s_{10}$ in the goal LTS (see Figure 4.1(a)). Specifically, in the case of the component service `e-Ride`', the actions for `Health4U` and `e-Ride`' do not match, whereas in the case of `e-Ride`", the corresponding guard condition is not satisfied. Thus, in the case of `e-Ride`' a failure results from an exception being raised either at `line 59` or `69`, indicating that a particular action present in the goal STS does not match with the component action for the particular transition. In the case of `e-Ride`" a failure arises due to an exception being raised either at `line 56` or `77`, indicating a mismatch in guards for the corresponding transition relation in the goal STS. MoSCoE provides such information about the cause of a failed attempt at service composition to the service developer. The developer

can then reformulate the original goal specification (e.g., changing the function parameters or pre-conditions) to realize a suitable mediator. These steps can be iterated until such a mediator is eventually realized or the user decides to abort.

4.3.4 Theoretical Analysis

Theorem 1 (Soundness & Completeness) *Given a goal service T_g with start state $s0_g$ and n component services $T_1 \dots T_n$ with the corresponding start states $s0_1 \dots s0_n$ the procedure* GENERATE*$(s0_g, [s0_1, s0_2, \dots, s0_n], t0, true, \emptyset)$ in Algorithm 1 is guaranteed to terminate with a mediator T_M with start state $t0$ if and only if $(\dots((T_M||T_1)||T_2)||\dots||T_n)\backslash L \approx^{tt} T_g$ whenever such a mediator exists, and with a failure otherwise.*

Proof Sketch: We prove the theorem by contradiction. Suppose the procedure GENERATE$(s0_g, [s0_1, s0_2, \dots, s0_n], t0, true, \emptyset)$ in Algorithm 1 yields a mediator T_M with start state $t0$ which when used to orchestrate the component services under the restrictions imposed by the guards L, fails to realize the goal service T_g, i.e., the composition is not bisimulation equivalent to T_g. There are four cases to consider: (i) for an input action in T_g, there is no corresponding input action in T_M; (ii) for an output action in T_g, there is no corresponding output action in T_M; (iii) an atomic action present in T_g is not modeled by the composition; and finally (iv) some sequence of actions in the goal is not provided by the composition due to the unsatisfiability of one or more guards.

However, case (i) is ruled out by the algorithm because for each message sent from the client to T_g, a corresponding input action is created in T_M to receive the message (Case 1 of GENERATE). Case (ii) is ruled out because for each output message that is to be sent to the client (as modeled in T_g), a corresponding output action is created in T_M if that message can be retrieved from the message store R (otherwise an exception is raised resulting in termination of the algorithm with failure (Case 2 of GENERATE)). Case (iii) is ruled out because the atomic actions in T_g are modeled by first determining the component(s) that can provide the relevant functions and then creating the relevant transitions in T_M to communicate with the respective component(s) (otherwise the algorithm terminates with failure). Note that the

communications between T_M and any T_i leads to transitions labeled by τ (Definition 6). The desired goal-function will be matched by the composition after zero steps if there is a component at a state with outgoing transition labeled by the function; otherwise the composition will lead to a state with an outgoing transition labeled by the desired function, after multiple τ-steps representing component-mediator synchronous communications. Finally, in all the above cases, if the guards do not match or the guards in the component(s) are stronger than those in T_g (and T_{cr}), the algorithm terminates with an appropriate failure cause, thereby ruling out case (iv).

Next, consider the case where there exists a mediator T_M that can orchestrate the component services $T_1 \dots T_n$ under the constraints imposed by L to realize the behavior specified by T_g but the procedure GENERATE terminates with a partial T_M or fails to terminate. We can rule out this possibility of generation of partial T_M through an argument similar to the one used above. Finally, the component services T_is and the goal service T_g are defined over guarded transitions with no variable operations. As such the variable domain can be finitely partitioned making the state-space of the component and the goal services finite. Therefore, the procedure GENERATE, which exhaustively explores the state-space of the services, terminates for all possible valuations of the variables.

Complexity. The worst-case complexity of the composition algorithm is determined by the number of recursive invocations of GENERATE. Assume that $|T_g|$ is the number of states in the goal service LTS, $|T_c|$ is the number of states in each component service LTS, and n is the total number of component services. In the worst case, each state in the goal LTS can be associated with any potential combination of states in the component LTSs, yielding $|T_c|^n$ combinations. Additionally, each pairing of a goal state with a combination of component states is interpreted in the context of a guard G and the messages stored in R. Guards and message stores are updated whenever the procedure GENERATE explores a transition from a goal or a component state. The number of distinct Gs and Rs is $O(2^{|T_g|\times|T_c|^n})$. The worst-case complexity of GENERATE is therefore $\mathrm{O}(|T_g| \times |T_c|^n \times 2^{|T_g|\times|T_c|^n})$.

4.3.5 Composition using Non-Functional Requirements

In the above, we relied only on "functional requirements" provided by the service developer to determine a feasible composition. Essentially, functional requirements specified the behavior or the functionality of the composition. However, more often service consumers want the service providers to not only satisfy the functional requirements, but also non-functional requirements which specify criteria that can be used to judge the operation of a system, rather than specific behaviors. Typical non-functional requirements are reliability, scalability, and cost, and play an important role in addressing various problems related to service discovery and composition. It is simple to imagine a scenario in which multiple services which provide the same functionality can fulfill a user request. In this case the ability of the user to differentiate between the services depends upon their non-functional properties.

Within our MoSCoE framework, such non-functional requirements are specified as part of the goal service model. Specifically, given the component services $LTS_1, LTS_2, \ldots, LTS_n$ and a goal service LTS_g, the objective is to compose a mediator LTS_M for communicating with a set of component services, such that the composition satisfies both the functional and non-functional requirements. The non-functional requirements are quantified using *thresholds*, where a composition is said to conform to a non-functional requirement if it is below or above the corresponding threshold, as the case may be. For example, for a non-functional requirement involving the `cost` of a service composition, the threshold may provide an *upper-bound* (maximum allowable `cost`) while for requirements involving `reliability`, the threshold usually describes a *lower-bound* (minimum tolerable `reliability`). If more than one composition model meets the goal specifications (i.e., there exists more than one mediator which can satisfy both the functional and non-functional requirements), our approach generates all such compositions and ranks them. Compositions with higher rank are better than those with the lower rank in terms of meeting the non-functional requirements. For example, given two valid composition models LTS_M and $LTS_{M'}$, if the `cost` of LTS_M is more than $LTS_{M'}$, then LTS_M is ranked lower than $LTS_{M'}$. It is left to the user's discretion to select the best model according to the requirements. Note that it is desirable to identify all the composition models, not just

```
    /*
    R is the repository of component services.
    a is the current atomic action in the goal LTS that is being analyzed.
    v is the non-functional attribute-value.
    v0 is the non-functional user-defined threshold value.
    F is the optimization function.
    Op is the optimization operator.
    C is the attribute-value comparison operator that is dependent on F.
    */
 1: procedure SELECT(C, a, v, v0, F, Op)
 2:   S = {Φ};
 3:   select any ci in R do
 4:     if ((ci provides a) && (F(v0, vi))) then
 5:       v = v Op vi;
 6:       if (F(v0, v)) then
 7:         S = ∪ ci;
 8:       end if
 9:     end if
10:   end select

11:   for (each i in 1 to length(S)) do
12:     for (each j in length(S) downto i+1) do
13:       if (vj C vj−1) then
14:         SWAP(S, cj, cj−1);
15:       end if
16:     end for
17:   end for

18:   if (S ≠ {Φ}) then
19:     return c0;
20:   end if
21: end procedure
```

Algorithm 2 Algorithm for Service Selection using Non-Functional Requirements

the best one, since a particular model is likely to be used multiple times in future to realize the goal service, and the component services that are part of the model may become unavailable at the time of execution. In such situations, the user can select an alternate model from the generated set of alternative composition models.

In our context of generating a mediator using Algorithm 1, the appropriate component service is selected during the analysis of case 3 (see `line 24` of Algorithm 1), and this selection is done entirely based on satisfaction of functional requirements. Consequently, we have developed a simple procedure called SELECT (Algorithm 2) that can augment Algorithm 1 to select component services based on both functional and non-functional requirements. Invocation of SELECT requires as input the repository R of component services, the atomic action a in the goal LTS that is being analyzed (in `line 23` of Algorithm 1), the value `v` of the non-functional attribute for the entire composition, the threshold value $\mathtt{v}_0$ of the non-functional attribute set

```
    /*
    R is the repository of component services.
    T_M is the mediator generated in the 1^st iteration of GENERATE (Algorithm 1).
    C_M is the set of component services used for composing T_M.
    */
 1: procedure GENERATEALL(R, T_M)
 2:   T_all = T = ∪ T_M;
 3:   for all T_i in T do
 4:     for all components c_j in C_i do
 5:       R' = R - {c_j};
 6:       if ((T_M' = GENERATE(g_0, [s_1, s_2, ..., s_{j-1}, s_{j+1}, ..., s_n], true, Φ)) ≠ null) then
 7:         if (C_M' ≠ C_i) then
 8:           T_all = T = ∪ T_M';
 9:         end if
10:       end if
11:       R = R' ∪ {c_j};
12:     end for
13:     T = T - T_i;
14:   end for
15: end procedure
```

Algorithm 3 Algorithm for Generating Multiple Mediators

by the user, the user-specified optimization function `F`, and the optimization operator `Op`. At first, the procedure randomly selects a component service c_i from the repository R that can provide the required action a (`line 3`). If the value of the non-functional attribute of c_i is lesser or greater than the threshold value $\mathtt{v}_0$ (`line 4`) and also does not violate the global requirement (`line 6`), then c_i is the added to the list of candidate services. The above steps are repeated for all the services in R (`lines 3--10`), and the resultant set of candidate services are sorted based on an appropriate minimization (e.g., `cost` should be minimized) or maximization (e.g., `reliability` should be maximized) criteria (`lines 12--17`). Once the best candidate service is selected from the sorted list (`line 19`), the execution of the case 3 in the GENERATE procedure of Algorithm 1 can proceed normally.

However, the combination of GENERATE and SELECT procedures will only model a single mediator that can potentially satisfy all the functional and non-functional requirements. As mentioned earlier, in practice there might be multiple component services that can satisfy the functional and non-functional requirements of the user, and hence can be analyzed for modeling more than one composition model. The procedure GENERATEALL (Algorithm 3) assists in precisely realizing this requirement. It takes as input the repository R of all component services, and the mediator T_M that was generated by execution of GENERATE (during its first iteration). The basic idea is to replace every component c present in T_M with an alternate c' to generate

$T_{M'}$s, and repeat the procedure for each $T_{M'}$. Thus, every component in R is exhaustively analyzed to generate a set of mediators that satisfy both the functional and non-functional requirements.

4.4 Discussion

In this chapter, we introduced a novel approach to developing composite services through an iterative reformulation of the goal service specifications. Specifically, we have presented a theoretically sound and complete technique for constructing a mediator that enables the interactions among component services to realize the behavior of the desired goal service. We use Labeled Transition Systems (LTSs) augmented with state variables over an infinite domain and guards over transitions to model the services. A unique feature of the proposed approach is its ability to work with an abstract (possibly incomplete) specification of a desired goal service. In the event the goal service cannot be realized (either due to incompleteness of the specification provided by the developer or the limited functionality of the available component services), the proposed algorithm identifies the causes for failure and communicates them to the service developer. The resulting information guides further iterative reformulation of the goal service until a composition that realizes the desired behavior is realized or the user chooses to abort. In addition to the above, we demonstrated how non-functional requirements can be incorporated into the composition framework. The main contributions can be summarized as follows:

- A new paradigm for modeling Web services based on abstraction, composition, and reformulation. The proposed approach allows users to iteratively develop composite services from their abstract descriptions.

- A sound and complete algorithm for selecting a subset of the available component services that can be assembled into a feasible composition that realizes the goal service with the user-specified functional and non-functional requirements, and for determining a mediator to interact with the component services. The proposed approach uses a variant

of LTS with guards on transitions to deal with the case when data and process flow are modeled in an infinite-domain.

- A technique for determining the cause(s) of failure of composition to assist the user (i.e., service developer) in modifying and reformulating the goal specification in an iterative fashion.

In spite of these advances, the proposed approach is restricted to services which can be specified using a limited class of constraints as guards in the LTS. This restriction ensures that the fixed point computation of similarity relation terminates, which necessitates investigation of a larger class of constraints (e.g., range constraints and arithmetic operations) based on the techniques described in [109]. Additionally, we focused on services which demonstrate a deterministic behavior without loops. Handling non-deterministic behavior that often characterizes real-world services is an important area of ongoing research. Furthermore, in our algorithm for composition based on non-functional requirements, we considered only single non-functional attributes for selection of candidate services and generating alternate composition models. However, this is a very restrictive setting and the ability to model compositions that can satisfy multiple non-functional requirements is needed. Finally, one area that needs significant investigation is the evaluation of our approach using real-world and benchmark [136] cases for service composition. As with any existing technique for service composition, the practical feasibility of our approach is also ultimately limited by the computational complexity of the service composition algorithm. Hence, methods for reducing the number of candidate compositions need to be examined e.g., by exploiting domain specific information to impose a partial-order over the available services, or reducing the number of goal reformulation steps needed by exploiting relationships among failure causes (or between failure causes and services, or between services) need further investigation. One possibility is to explore development of heuristics for hierarchically arranging failure-causes to reduce the number of refinement steps typically performed by the user to realize a feasible composition and doing usability studies along those dimensions. Additionally, it is important to understand the precision and recall measures to capture the effectiveness of search for component services that match the specifica-

tions, doing which will require systematic experiments using service composition benchmarks. However, one can draw an analogy with information retrieval systems that respond to user queries (typically expressed using keywords) in a single step as opposed to systems that allow users to iteratively reformulate their query based on the retrieved results [85, 172]. In general, the information retrieval systems that support iterative query reformulation are able to achieve superior performance in terms of precision and recall (relative to documents of interest to the user). Analogously, all other factors being same, the precision and recall achievable (after a few iterations of reformulating the goal) by any service composition system that supports iterative reformulation of specifications would be superior compared to a system that does not.

CHAPTER 5. WEB SERVICE SPECIFICATION REFORMULATION

This chapter introduces the problem of Web service specification reformulation and describes in details our approach to address the problem. The chapter is divided into four sections. The first section provides background to and describes the problem of Web service specification reformulation. An illustrative example is described in the second section to explain the salient features of our proposal, which is discussed in the third section. The fourth section concludes the chapter with a discussion.

5.1 Introduction and Problem Description

In the previous chapter, we introduce the problem of Web service composition where we outlined that in many cases a composite (or goal) service may not be realized using a set of available (or component) services because the specified functional and/or non-functional requirements are not met. In such circumstances, a service developer has to modify the specification of the goal service manually and repeat the composition procedure. Generally, there are two broad classes of scenarios in which a service composition algorithm fails to realize a specified goal service:

- The desired *functionality* of the goal service *cannot* be realized by composing the available component services. In this case, the user needs to either modify the overall functionality of the desired goal service (e.g., settle for a functionality that is not quite the same as what was initially desired) and/or broaden the search for component services beyond those initially considered by the algorithm.

- The desired *functionality* of the goal service *can* be realized by composing the available

component services, but the composition algorithms fail to "mimic" the *structure* of the goal service using the available component services. In this case, it is possible to reformulate the structure of the goal service specification, without altering its overall functionality, into one that can be realized by using the existing services.

Our work illustrated in Chapter 4 has provided automated identification of cause(s) of failure of composition in the form of information about specific inputs or outputs and pre- and post-conditions of the actions as well as the control and data flows that are part of the goal service specification, but cannot be realized using the available component services. However, these methods do not distinguish between the two scenarios described above. Furthermore, in both scenarios, the tedious manual reformulation of an alternative goal service specification to be tried is left to the user. For example, in techniques based on state charts [27], a service developer will have to manually make changes either by adding/deleting state transitions and/or editing transition labels. Similarly, techniques based on LTS, including ours, have the same limitation.

Motivated by this need, we present a novel approach for*composing Web services through automatic reformulation of service specifications* (see Figure 5.1). Without loss of generality, we use LTS to represent the goal service provided by a service developer, the set of available component services, and the generated composite service that realizes the goal. We show that any alternative goal LTS reformulation that does not violate the data and control dependencies that are implicit in the user-supplied goal service LTS specification is *provably functionally equivalent* to the goal service. The reformulation of goal service specification is triggered by the failure of the composition algorithm to realize a composite service using the "original" LTS specification of the goal service provided by the service developer. We describe an efficient data structure, in the form of a *dependency matrix* and algorithms to maintain and analyze the data and control flow dependencies in goal service LTS specification. This data structure can be used to iteratively generate (as yet untried) alternatives that are functionally equivalent to the user-supplied goal service LTS specification until composition succeeds or no alternative reformulations remain to be tried. Because generating the complete set of alternatives that are functionally equivalent to a user-supplied goal service LTS specification is expensive and po-

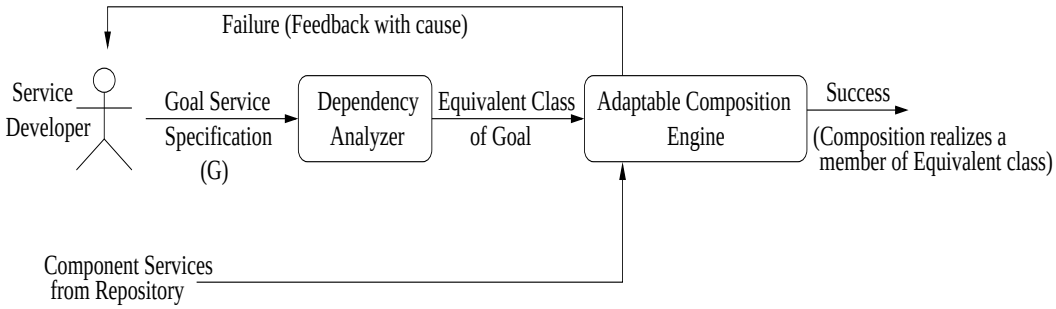
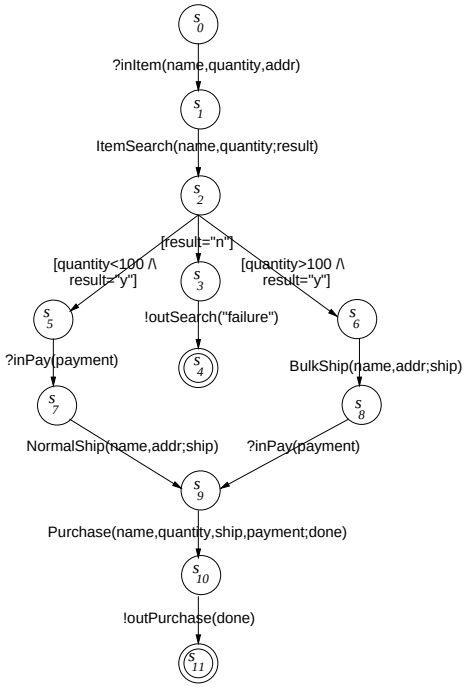


Figure 5.1 Composing Web Services through Automatic Reformulation of Service Specifications

tentially wasteful, we generate the alternatives *on-the-fly*. The result is a significant reduction in the need for the tedious manual intervention in reformulating specifications by limiting such interventions to settings where both the original goal LTS as well as its alternatives cannot be realized using the available component services. We explain the salient features our approach in details in the remainder of this chapter and begin with an illustrative example in the next section.

5.2 Illustrative Example

Assume that a service developer is assigned to model a new composite (or *goal*) service that allows clients to purchase items and ship them to a particular destination. To achieve this, the goal service operates as follows: (*i*) First, it accepts from the client as input the *name* of the item to be purchased along with the desired *quantity* and the *address* where the consignment has to be shipped. (*ii*) Once the input is received, it searches the particular item for the required quantity in an inventory. (*iii*) If the search fails, a failure message is sent to the client. But if the search succeeds, depending on the quantity of the item to be purchased, either *bulk* or *normal* shipping is checked for confirming whether the items can be shipped to the particular *address* or not. (*iv*) Also, if the item search succeeds, the client is asked to provide *payment* information which is eventually used for *purchasing* the items. (*v*) Finally, an appropriate *notification* is sent to the client indicating whether the entire process was a success or failure.



(a)

Actions	Name
a	`?inItem(name,quantity,addr)`
b	`ItemSearch(name,quantity;result)`
c	`?inPay(payment)`
d	`!outSearch("failure")`
e	`BulkShip(name,addr;ship)`
f	`NormalShip(name,addr;ship)`
g	`Purchase(name,quantity,ship,payment;done)`
h	`!outPurchase(done)`
Guards	**Name**
G_1	`[quantity<100 ∧ result="y"]`
G_2	`[quantity>100 ∧ result="y"]`
G_3	`[result="n"]`

(b)

Figure 5.2 (a) LTS Representation of (a) `e-Buyer` service (b) Mapping of Actions/Guards in `e-Buyer`.

Figure 5.2(a) shows the representation of such a goal service, named `e-Buyer`, described using a labeled transition system. Here, `?msgHeader(msgSet)` and `!msgHeader(msgSet)` refer to the input and output actions of the services, essentially corresponding to the reception and emission of messages, respectively. Communication between different services occurs via *synchronization* between actions with the same `msgHeader` resulting in the transfer of `msgSet` from the entity performing an output action to the one performing an input action. For example, `?inItem(name,quantity,addr)` is an input action in Figure 5.2(a) where `inItem` is the message header and `name`, `quantity` and `addr` are variables in the input message. The services also include atomic actions denoted by `funcName (inputSet; output)`. Additionally, a transition is annotated by guards (denoted by `[guards]`) which control whether or not it is enabled; the absence of a guard implies that the guard is *true* (always enabled). In essence, the guards are used to capture conditional-transitions and model the branching behavior of a

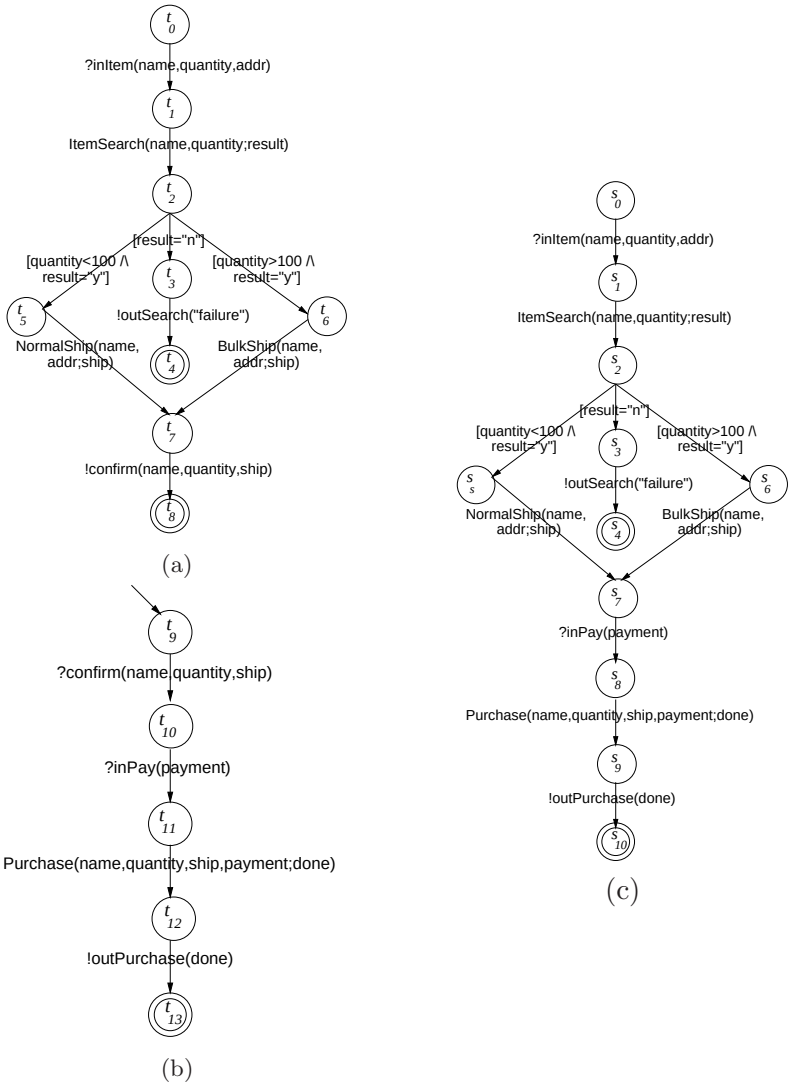


Figure 5.3 LTS Representation of (a) `Search-N-Ship` (b) `Purchaser` (c) `e-Buyer'` services

service.

A closer analysis of Figure 5.2(a) would reveal various data and control flow (in-) dependencies between the actions present in the `e-Buyer` service. For example, the input action `?inItem (name,quantity,addr)` has to occur *before* the atomic action `ItemSearch (name,quantity;result)` can be executed since the information required to execute the latter is provided by the former. Similarly, the atomic actions `NormalShip (name,addr; ship)` and `BulkShip(name,addr;ship)` are executed *mutual exclusively* depending on the valuation of the variables `quantity` and `result`.

Consider a scenario where a service developer wants to determine whether a goal service such as `e-Buyer` can be realized using a set of component services (Figures 5.3(a) & 5.3(b)). If `Search-N-Ship` is selected first, it leads to a composition failure due to different branching behaviors since `e-Buyer` requires the execution of an input action `?InPay(payment)` when [`quantity` $<$ `100` $\wedge$ `result = y`] is true, whereas `Search-N-Ship` invokes the atomic action `NormalShip` when the same condition is satisfied. Observe that, `?inPay(payment)`, which is provided by `Purchaser`, is only possible after it synchronizes with `Search-N-Ship` on the message `confirm`. Similarly, selecting the `Purchaser` service first leads to a failure as well. For both the circumstances, typically the service developer will be required to modify/reformulate the goal service representation (in this case, adding/deleting transitions and/or changing transition labels in the LTS representation of `e-Buyer`) and re-initiate the composition process.

However, it will be beneficial if there exists a technique that can automatically carry out all the potential modifications on behalf of the service developer and iterate through the composition process. As a matter of fact, in the current example, there exists an alternate goal service specification (`e-Buyer'`, see Figure 5.3(c)) which is (a) functionally equivalent to but structurally different from `e-Buyer`, and (b) realizable from the existing component services.

Note that, the above composition process failed for the original goal service (i.e., `e-Buyer`) since typically the composition algorithms aim to realize the *exact structural* representation of the goal service using the component services. Instead, we will show in the proceeding sections that it suffices to realize a composition using *alternate* structural representations (such as

`e-Buyer'`) of the original goal service, as long as the generated alternatives have the same functionality as the original goal service.

5.3 Our Approach

As described in Chapter 4, composition within our framework amounts to generating a mediator that will realize the goal service from a set of component services. In our setting, we only consider the functional specifications which are represented by LTSs. Similar approaches have been proposed in literature that are based on state charts [27], finite-state automata [33], and logic programming [169]. However, a common thread to all the techniques is that realization of the goal service is based on the exact realization of the structure of the specification. For instance, in techniques based on LTSs, composition requires the exact realization of the LTS representation of the goal service from the sequence of operations in the component LTSs, even though, in many cases exact realization requirement can be relaxed to realize another LTS which is functionally equivalent to the original one, but structurally different.

5.3.1 Functionally Equivalent Web Services

We define the notion of functional equivalence of two services in terms of the corresponding equivalence between the respective LTSs. Given an LTS $L = (S, \longrightarrow, s0, S^F)$, its behavior can be represented as the sequences of actions and guards from $s0$ to some state in S^F. Notationally, we will describe such a sequence as a string $(s_0\sigma_0)(s_1\sigma_1)\ldots(s_n\sigma_n)$, where WLOG we can assume that every transition in LTS is labeled either with an action or a guard and $\forall i(0 \leq i \leq n) : s_i \xrightarrow{\sigma_i} s_{i+1}$ such that $s_0 = s0$ and $s_{n+1} \in S^F$. We refer to all such sequences in L as the behavior of L and denote it by $\mathcal{B}(L)$. The definition of functional equivalence based on $\mathcal{B}(L)$ is as follows:

Definition 8 (Functional Equivalence) *An LTS L_1 is said to be functionally simulated by an LTS L_2, denoted by $L_1 \sqsubseteq L_2$, if and only if for all `seq` $= (s_0\sigma_0)(s_1\sigma_1)\ldots(s_n\sigma_n) \in \mathcal{B}(L_1)$ there exists `seq'` $\in \mathcal{B}(L_2)$ such that `seq` and `seq'` are permutation of each other with the following conditions:*

1. *For $i < j$, $(s_i\sigma_i)$ appears before $(s_j\sigma_j)$ in* `seq'` *if $ovars(\sigma_i) \cap ivars(\sigma_j) \neq \emptyset$.*

2. *For $i < j$, $(s_i\sigma_i)$ appears before $(s_j\sigma_j)$ in* `seq'` *if s_i is a branch point in L_1 and σ_j does not appear in all sequences in $\mathcal{B}(L_1)$ that contains s_i.*

L_1 and L_2 are functionally equivalent, denoted by $L_1 \equiv L_2$, if and only if $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$.

In the above, the first condition asserts that if LTS L_1 demands input of an operation (for either an action or guard) which must be obtained from the output of another operation, then it must be conformed by the corresponding sequence in LTS L_2. The second condition ensures that if an operation depends on a guard (i.e., appears in a specific branch) in L_1, it must similarly depend on the same guard in L_2. Functional equivalence demands that L_1 and L_2 *functionally simulate* each other. Note that this notion of equivalence is different from simulation or bisimulation equivalence applied traditionally in process algebra [129] where the objective is to analyze equivalences based on structural similarities.

It can be shown that given an LTS L_1, we can synthesize a set of LTSs that are functionally equivalent to L_1 by appropriately analyzing the control and data flow requirements present in the L_1. This forms the central theme of our technique. Given an LTS representation of the goal service, our technique automatically identifies functionally equivalent alternates that can be successfully realized from the available component services. For example, the atomic action `NormalShip(name,addr;ship)` in Figure 5.2(a) does not depend on the input action `?inPay` (`payment`), even though in the LTS specification they "appear" to be dependent on each other. Consequently, this gives the liberty to manipulate the ordering of the operation sequences in the LTS, which potentially amounts to generating alternate functionally equivalent LTS specifications (in this case Figure 5.3(c)), and hence an opportunity to determine the existence of feasible compositions using the alternatives.

Our work has resulted in the development of algorithms for automatically generating such alternate representations which could potentially lead to a feasible composition. In particular, our technique automatically extracts the dependency relations between the operations in the

goal LTS L_g (Section 5.3.3), which are represented using a dependency matrix (Section 5.3.2), and reformulates L_g such that it can be realized from the existing components without altering its "overall" desired functionality (Section 5.3.4).

5.3.2 Web Service Dependency Matrix

Given an LTS describing a service, there are two types of dependencies:

1. *Data Dependency:* if the input of an action a or the valuation of a guard g depends on the output of another action b, then a or g is said to be data dependent on b.

2. *Control Dependency:* if the execution of an action a or the valuation of a guard g depends on the valuation of another guard g', then a or g is said to be control dependent on g'.

The dependencies are captured using a dependency matrix defined as follows.

Definition 9 (Dependency Matrix) *Given an LTS $L = (S, s0, \longrightarrow, S^F)$, its dependency matrix D_L is a $N \times N$ matrix, where N is the number of actions (atomic, input and output) and guards in the transition-labels. For a row i and column j in D_L, the cell $C_{i,j}$ is assigned such that:*

- *if $C_{i,j} = \{X\}$, then the i-th element is control-dependent on the j-th element. The assignment X is done between action-guard or guard-guard pairs denoting that the guard j has to be analyzed before action i can be executed or guard i can be analyzed.*

- *if $C_{i,j} = ovars(i) \cap ivars(j)$, then the i-th element is data-dependent on the j-th element i.e., outputs from the j-th element are used for the analysis/evaluation of the i-th element. This assignment is done between guard-action or action-action pairs.*

- *if $C_{i,j} = \{Y\}$, then the i-th and j-th elements are guards which label different transitions from a branch-point in L. That is, the elements i and j are mutually exclusive and cannot appear in the same path in L.*

Figure 5.4 above shows the dependency matrix of `e-Buyer` service shown in Figure 5.2(a). For example, it states that action `a` (`?inItem`) has to occur before action `b` (`ItemSearch`) because the variables `name` and `quantity` required to execute `b` (input parameters of `ItemSearch`) are provided by `a` (inputs to the service). Similarly, the guards $\mathtt{G}_1$, $\mathtt{G}_2$ and $\mathtt{G}_3$ are mutually exclusive since they appear in separate execution paths. Here, `a`, `b`, $\mathtt{G}_1$, $\mathtt{G}_2$, and $\mathtt{G}_3$ correspond to actions and guards in Figure 5.2(b).

Theorem 2 *For any two LTSs L and L', $L \equiv L'$ if and only if their dependency matrices D_L and $D_{L'}$ are identical.*

Proof: Let $L \equiv L'$ and $D_L \neq D_{L'}$. D_L and D'_L must have the same set of row or column labels since from Definition 8, $L \equiv L'$ implies that they have same set of operations. Therefore, $D_L \neq D'_L$ implies that there exists $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ where $\mathtt{C}_{i,j} \in D_L$ and $\mathtt{C}'_{i,j} \in D'_L$. Note that, $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ implies that $\mathtt{C}_{i,j} \cap \mathtt{C}'_{i,j} = \emptyset$. Therefore, the only way $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ is when $\mathtt{C}_{i,j} \neq \emptyset$ and $\mathtt{C}'_{i,j} = \emptyset$ or vice versa. Let $\mathtt{C}_{i,j} = \{\mathtt{X}\}$ and $\mathtt{C}'_{i,j} = \emptyset$ (case 1 in Definition 9). This implies that i is control dependent on j in L and i is not control dependent on j in L'. Therefore, in all sequences in $\mathcal{B}(L)$ which contains i, j appears before i while there exists some sequence in $\mathcal{B}(L')$, where i is present and j is absent or i is present before j. This contradicts our initial assumption that $L \equiv L'$ by Definition 8. Same type of contradiction can be realized for other cases where $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$.

Next assume $D_L = D'_L$ and $L \not\equiv L'$. Therefore, there exists a pair of operations i and j such that j appears before i in all sequences in $\mathcal{B}(L)$ containing i. However, there exists at least one sequence in $\mathcal{B}(L')$, containing i and j, where i appears before j. The case implies that i depends on j in L while it is not dependent on j in L'. In other words, $\mathtt{C}_{i,j} \neq \emptyset$ while $\mathtt{C}'_{i,j} = \emptyset$. This leads to contradiction of the initial assumption of $D_L = D'_L$. □

In our setting, we only consider the functional specifications which are represented by LTSs. Consequently, composition requires the exact realization of the LTS representation of the goal service from the sequence of operations in the component LTSs. However, this condition can be relaxed due to the following: though the dependencies between operations can be effectively captured in LTSs, it is not possible to truly capture *independent operations*. In fact, operations

that are independent may appear to be dependent due to the order in which they appear in the LTS. For example, `NormalShip(name,addr;ship)` in Figure 5.2(a) does not depend on `?inPay(payment)` operation, but in the LTS specification, they appear in a specific order. As a result, if the component services do not realize the exact sequence of operations in the goal LTS, the composition fails.

To counter this situation, we precisely identify the required operation dependencies from the LTS and determine a composition (if it exists) that realizes these dependencies. In essence, our technique automatically extracts the dependency relations between the operations in the goal LTS L (Section 5.3.3), which are represented using a dependency matrix (Section 5.3.2), and adapts L such that it can be realized from the existing components without altering the "overall" desired functionality (Section 5.3.4).

5.3.3 Generation of the Dependency Matrix

Identifying Data Flow Dependency. Procedure DATA FLOWDEP in Algorithm 4 identifies the data dependencies in an LTS and appropriately updates the dependency matrix. It takes as argument the current state (`curr`) of the LTS being explored for dependency analysis, the operation (guard or the action, `Op`) and the set of variables (`VSet`) in `Op` whose data dependencies are being analyzed. Backward exploration of the LTS is performed from `curr`. If a parent-state is reachable via an action "`a`" such that the intersection of its output and `VSet` is non-empty (`lines 6--9`), then the corresponding cell in the dependency matrix ($C_{Op,a}$) is assigned `output(a)`. The `VSet` is updated by removing `output(a)` from the set. Finally, the procedure is recursively invoked (`line 10`). The recursion terminates (`lines 2--4`) when `VSet` is empty (i.e., all the data-dependencies of `VSet` have been identified) or `curr` is the start-state of the transition system (i.e., there exists no incoming transition to `curr`). For example, invocation of DATAFLOWDEP(s_1, `ItemSearch`, {`name,quantity; result`}) in `e-Buyer` of Figure 5.2(a), will create a dependency with the action `?inItem` since the latter (i.e., `?inItem`) provides the inputs required to execute the former (i.e., `ItemSearch`).

Identifying Control Flow Dependency. Procedure CTRL FLOWDEP in Algorithm 5 iden-

```
1: procedure DataFlowDep(curr, Op, VSet)
2:   if (curr is a start-state Or VSet = ∅) then
3:     return
4:   end if
5:   for all parent --g,a--> curr do
6:     if (output(a) ∩ VSet ≠ ∅) then
7:       C_Op,a := output(a)
8:       VSet := VSet − output(a)
9:     end if
10:    DataFlowDep(parent, Op, VSet)
11:  end for
12: end procedure
```

Algorithm 4 Identifying Data Flow Dependency

tifies the control dependencies in an LTS and appropriately updates the dependency matrix. Similar to DataFlowDep, it takes `curr` and `Op` as arguments and performs backward traversal from `curr`. If a parent state of `curr` has more than one outgoing transition (i.e., it is a branch-point in the LTS with guarded transitions originating from it), then the cell ($\mathtt{C}_{\mathtt{Op,g}}$) in the dependency matrix corresponding to the operation `Op` and the guard `g` (associated with the transition from the parent to `curr`) is assigned `X` (`lines 6--8`). The procedure is recursively invoked (`line 9`) and terminates when `curr` is the start-state of the transition system (i.e., there exists no incoming transition to `curr`). For example, invocation of CtrlFlowDep (s_7, `NormalShip{name,addr;ship}`) (Figure 5.2(a)), will create a dependency with the guard [`quantity<100` ∧ `result=` "`y`"] since its valuation will decide whether or not `NormalShip` will be executed.

As can be observed, the procedure CtrlFlowDep conservatively classifies operations as control dependent on a branch-point (more precisely on the guard associated with the branch-point) in which it appears. However, there are cases where an operation might appear in all the possible branches. In such a situation, the said operation is *not* control dependent on the branch-point as it is invoked for all possible valuation of the guards at the branch-point. In order to precisely identify control dependencies by eliminating such cases, CtrlFlowDep is followed by invocation of UpdateCtrlFlowDep (Algorithm 5). This procedure also marks dependency matrix cells with `Y` to identify the operations which must not appear at branches associated with a particular guard in the transition system. his procedure, unlike CtrlFlowDep and DataFlowDep, performs forward traversal of the LTS from each branch-point. This pro-

	a	b	c	d	e	f	g	h	G_1	G_2	G_3
a											
b	name,quantity										
c									X	X	
d											X
e	name,addr									X	
f	name,addr								X		
g	name,quantity		payment		ship	ship			X	X	
h							order		X	X	
G_1	quantity	result								Y	Y
G_2	quantity	result							Y		Y
G_3		result							Y	Y	

(a)

	b	c	d	e	f	g	h	G_1	G_2	G_3
b										
c								X	X	
d										X
e									X	
f								X		
g		payment		ship	ship			X	X	
h						order		X	X	
G_1	result								Y	Y
G_2	result							Y		Y
G_3	result							Y	Y	

(b)

	c	d	e	f	g	h	G_1	G_2	G_3
c							X	X	
d									X
e								X	
f							X		
g	payment		ship	ship			X	X	
h					order		X	X	
G_1								Y	Y
G_2							Y		Y
G_3							Y	Y	

(c)

	c	f	g	h	G_1
c					X
f					X
g	payment	ship			X
h			order		X
G_1					

(d)

	c	e	g	h	G_2
c					X
e					X
g	payment	ship			X
h			order		X
G_2					

(e)

	d	G_3
d		X
G_3		

(f)

	c	f	g	h
c				
f				
g	payment	ship		
h			order	

(g)

Figure 5.4 Dependency Matrices (a) D_L (b) D_L^1 (c) D_L^2 (d) D_L^3 (e) D_L^4 (f) D_L^5 (g) D_L^6

```
1: procedure CtrlFlowDep(curr, Op)
2:   if (curr is a start-state) then
3:     return
4:   end if
5:   for all parent --g,a--> curr do
6:     if (outGoingTrans(parent) > 1) then
7:       C_{Op,g} := X
8:     end if
9:     CtrlFlowDep(parent, Op)
10:  end for
11: end procedure

12: procedure UpdateCtrlFlowDep(branchPoint)
13:   T := {paths from branchBegin to branchEnd}
14:   GuardSet := {g | branchBegin --g,a--> next}
15:   for all Op such that ∃g ∈ GuardSet : C_{Op,g} = X do
16:     if (∀t ∈ T : Op ∈ t) then
17:       ∀g' ∈ GuardSet : remove X from C_{Op,g'}
18:     end if
19:   end for
20:   for all g_1, g_2 ∈ GuardSet do
21:     C_{g_1,g_2} := Y
22:   end for
23: end procedure
```

Algorithm 5 Identifying Control Flow Dependency

cedure identifies (a) the set of paths originating from each branch-point to a state which is either a final state and/or a joint point of the branch (`line 13`), and (b) the set of guards at a branch-point (`line 14`). If there is an operation `Op` (obtained from procedure CtrlFlowDep) which is dependent on at least one guard associated with the branch-point (`line 15`) and also appears in *all* paths in T (`line 16`), then `Op` is *not* control-dependent on any guard appearing in the branch-point under consideration. Accordingly, all the `X`s in $\mathtt{C}_{\mathtt{Op,g}}$ are removed (`line 17`). Furthermore, `Y` is assigned to the cells corresponding to the guards associated with the same branch point as all the guards cannot evaluate to true simultaneously, i.e., they are mutually exclusive (`lines 20--21`). For example, invocation of UpdateCtrlFlowDep(s_2) in `e-Buyer` of Figure 5.2(a) will assign `Y`s to the cells $\mathtt{C}_{\mathtt{G}_1,\mathtt{G}_2}$, $\mathtt{C}_{\mathtt{G}_1,\mathtt{G}_3}$, $\mathtt{C}_{\mathtt{G}_2,\mathtt{G}_3}$, $\mathtt{C}_{\mathtt{G}_2,\mathtt{G}_1}$, $\mathtt{C}_{\mathtt{G}_3,\mathtt{G}_1}$ and $\mathtt{C}_{\mathtt{G}_3,\mathtt{G}_2}$ in the dependency matrix.

5.3.4 Algorithm for Reformulation-based Web Service Composition

Once we have obtained a dependency matrix D_L of the goal service LTS L as outlined above, our objective is to analyze the matrix and identify alternate models (with the same data and control dependencies) of L which can be used for determining feasible compositions.

```
    /*
    D_L is the Dependency matrix of the goal LTS L.
    S is the set of component service states.
    */
 1: procedure ReformulateService(D_L, S)
 2:   if (D_L is null) then
 3:     return true;
 4:   end if
 5:   R := {i | ∀j ∈ D_L^col : C_{i,j} is empty or only contains Y}
 6:   select any r ∈ R do
 7:     if (for any j ∈ D_L^col, C_{r,j} = Y) then
 8:       DSet := Reduce(D_L, r, true);
 9:       for all D'_L ∈ DSet do
10:         if ¬ReformulateService(D'_L, S) then
11:           break and backtrack to line 6
12:         end if
13:       end for
14:       return true
15:     else
16:       {D'_L} := Reduce(D_L, r, false);
17:       if CreateTransition(r, S') then
18:         if ¬ReformulateService(D'_L, S') then
19:           backtrack to line 6
20:         else return true
21:         end if
22:       else backtrack to line 6
23:       end if
24:     end if
25:   end select  return false
26: end procedure

27: procedure Reduce(D, r, flag)
28:   DSet := ∅;
29:   if (flag = true) then
30:     for all (i ∈ {Op | C_{Op,r} = Y} ∪ {r}) do
31:       WorkingSet := {j | C_{i,j} = Y}
32:       DNew := D
33:       remove Y from all the C_{i,r} and C_{r,i} in DNew
34:       while (WorkingSet ≠ ∅) do
35:         for all j ∈ WorkingSet do
36:           if (C_{k,j} ⊈ ⋃{C_{k,l} | l ≠ j}) then
37:             add k to WorkingSet
38:           end if
39:           remove j from WorkingSet
40:           remove j-th row and column from DNew
41:         end for
42:       end while
43:       DSet := DSet ∪ DNew
44:     end for
45:   else
46:     remove r-th row and column from D
47:     DSet := {D}
48:   end if
49:   return DSet
50: end procedure
```

Algorithm 6 Reformulation-based Service Composition

This technique essentially allows reformulation during composition, that is, given a goal LTS L which results into failure of composition, the technique automatically reformulates L to identify a functionally equivalent model L' and checks for feasible compositions *on-the-fly*, that is, L' is generated as and when the composition feasibility is checked, as opposed to, generating L' first and then checking for its feasibility. The main intention behind this step is due to the fact that generating the complete set of alternatives L's that are functionally equivalent to L is expensive and potentially wasteful. Algorithm 6 shows our approach for identifying alternate models from the dependency matrix, which we explain below using the example described in Section 5.2.

The procedure ReformulateService (Algorithm 6) takes as argument D_L, the dependency matrix of the goal service (e.g., `e-Buyer`, Figure 5.2(a)), and S, the set of states of the component services (e.g., `Search-N-Ship`, Figure 5.2(c) and `Purchaser`, Figure 5.2(d)). Initially, the procedure determines the set of operations in D_L which are not dependent on any other operation (`line 5`). These operations are required to be realized by the component services. For example, the operation `a` in the D_L (Figure 5.4(a)) of `e-Buyer` is not dependent on any other operation and can be realized by the transition from state t_0 to t_1 of `Search-N-Ship` (CreateTransition[1] in `line 17` holds true). As a result, D_L is updated to D_L^1 (Figure 5.4(b)) by removing the row and column corresponding to `a` signifying that `a` is already realized (removing the row) and all the dependencies on `a` are, therefore, eliminated (removing the column). This is achieved by executing `lines 16` in ReformulateService and `46--47` in Reduce (Algorithm 6).

In the next step, ReformulateService is recursively invoked with D_L^1 and the new set of states of the component services, S', reached after realizing operation `a` (e.g., `Search-N-Ship` is in state t_1) (`line 18`). In D_L^1, operation b is not dependent on any other operation and can be again realized by `Search-N-Ship` (`line 17`). Thus, D_L^1 is updated to generate D_L^2 (Figure 5.4(c)) following the same steps as described above and ReformulateService is recursively

[1]The procedure CreateTransition is used to generate the alternate LTS specification, L', as part of the reformulation-based composition process. It takes as argument the operation r being analyzed and the set of component states S' reached after realization of r (by one of the component services) and generates a corresponding transition in L'. Details are present in Chapter 4.

invoked.

Proceeding further, in D_L^2 the only possible operations that can be considered are $\mathtt{G}_1$, $\mathtt{G}_2$ and $\mathtt{G}_3$ since they are independent of other operations (`line 5`). However, all the three operations are guards and lead to different branches of a branch-point. Consequently, the component services must realize each individual branch. Furthermore, since the branches are mutually exclusive (i.e., their rows and columns are marked `Y`), if a branch $\mathtt{G}_i$ is considered, then all the branches at the same branch point corresponding to the guards $\mathtt{G}_j$ $(j \neq i)$ must be removed from the dependency matrix. This is achieved when ReformulateService at `line 8` invokes Reduce. The procedure Reduce executes the statements from `lines 29--45` to create a set of matrices corresponding to each guard. During the execution of Reduce, initially a working set of operations that must be removed is created (`line 31`). Referring to our example, consider the case where we are exploring the branch corresponding to guard $\mathtt{G}_1$ (`line 6`) in D_L^2 , and the working set is $\{\mathtt{G}_2, \mathtt{G}_3\}$. Firstly, all the `Y`-marks are removed from the cells $\mathtt{C}_{\mathtt{G}_i,\mathtt{G}_j}$ (`line 33`) in D_L^2. Then for each operation x in the working set, any operation y that is *solely* dependent on x is added to the working set (`line 37`). For example, for $\mathtt{G}_2$ in D_L^2, the operation `e` is solely dependent on $\mathtt{G}_2$, whereas operation `c` is not since $\mathtt{C}_{\mathtt{c},\mathtt{G}_1} = \mathtt{X}$. Thus, `e` is added to the working set and operations solely dependent on `e` are identified iteratively for addition to the working set. On the other hand, since operation `c` is not solely dependent on $\mathtt{G}_2$, it is not added to the working set. Furthermore at each iteration, an element is removed from the working set and its corresponding rows and columns are removed from the dependency matrix (`line 40`) and the above process continues until the working set becomes empty. In our example, execution of `line 8` with D_L^2 will result in the creation of matrices D_L^3 (corresponding to $\mathtt{G}_1$ being selected at `line 30`), D_L^4 (corresponding to $\mathtt{G}_2$) and D_L^5 (corresponding to $\mathtt{G}_3$) as shown in Figures 5.4(d), 5.4(e) and 5.4(f), respectively.

The procedure ReformulateService will be invoked with each of these matrices as inputs (`line 9`). The procedure terminates successfully when the dependency matrix is empty denoting all operations are successfully realized by the component services and there were no failures during composition (`lines 2--3`). Otherwise, if a particular operation is not realiz-

able then backtracking is performed to pick an alternate operation (`line 11`). For example, assuming that D_L^3 is selected in `line 9`, after realizing the guard $\mathtt{G}_1$ ([`quantity` $<$ 100 $\wedge$ `result = y`]) by `Search-N-Ship`, it will be updated to create a new dependency matrix D_L^6 (Figure 5.4(g)). In D_L^6, the operations that can be considered are `c` (`?inPay(payment)`) and `f` (`NormalShip (name,addr;ship)`) since they are independent of other operations (`line 5`). However, if operation `c` is selected first in `line 6`, it will result into a composition failure since such a behavior cannot be realized by any of the existing component services. That is, none of the component services (Figures 5.2(c) & 5.2(d)) has a transition associated with the guard $\mathtt{G}_1$ immediately followed by another transition associated with the action `c`. As a result, ReformulateService will backtrack, and select `f` and determine if it can be realized. Thus, in essence, where the existing algorithms for service composition would have failed at this point, our approach automatically adapts the goal service based on the analysis of control and data flow dependencies for identifying feasible compositions. For this particular example, the composition obtained eventually will correspond to the (alternate) goal service `e-Buyer'` (Figure 5.3(c)). Note that even though the original goal service `e-Buyer` and its alternate model `e-Buyer'` are structurally different, they are *functionally equivalent* since they realize the same functionality and have the same data and control dependencies.

Theorem 3 (Soundness & Completeness) *Given a service L and set of component services CS with start state-set S, there exists a service L' such that* ReformulateService *(D_L, S) returns true if and only if $L \equiv L'$ and CS realizes L'.*

Proof: Let ReformulateService(D_L, S) return true and for all L's realized from CS, such that $L \not\equiv L'$. From Theorem 2, $\forall L' : L' \not\equiv L \Rightarrow D'_L \neq D_L$. In other words, there exists at least one pair of operations in L such that $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ ($\mathtt{C}_{i,j} \in D_L$ and $\mathtt{C}'_{i,j} \in D'_L$) that is not realizable from CS. Proceeding further, $\mathtt{C}_{i,j}$ demands a specific ordering or mutual exclusion of i and j in all sequences and this is not realizable from CS. This, in turn, implies that ReformulateService fails at `line 17` for all possible choices at `line 6`, and eventually returns false at `line 25`. This leads to contradiction of our initial assumption that ReformulateService returns true.

Next, consider that case where ReformulateService returns true but CS *does not* realize any L' ($\equiv L$). I.e., for all possible alternate sequences in $\mathcal{B}(L)$, their exists some operation in each sequence for which CreateTransition fails. If such a failure occurs (`line 17`) in ReformulateService, the algorithm backtracks and selects alternate functionally equivalent sequences using D_L. Finally, when all alternates are exhausted and CreateTransition fails in all of them, ReformulateService returns false (`line 25`). This contradicts our initial assumption that ReformulateService returns true.

Finally, consider that there exists an $L \equiv L'$ and CS realizes L' but ReformulateService(D_L, S) returns false (`line 25`). This will happen when CreateTransition fails for all possible alternates identified by ReformulateService. If we assume that ReformulateService correctly computes all possible alternates, then failure of ReformulateService due to failure of CreateTransition directly contradicts the initial assumption that CS realizes $L'(\equiv L)$.

The other alternate is that ReformulateService does not correctly consider all possible alternates, and hence fails to identify the $\mathcal{B}(L')$ which is realizable from CS. `Line 6` in ReformulateService considers all operations which are not data-dependent on any other operation as candidates for realizability. If such a candidate operation is a guard, ReformulateService invokes Reduce to obtain dependency matrices of all paths beyond the branch point of the guard under consideration. The procedure Reduce selects all the mutually exclusive guards (`line 30`) and for a particular guard i, it firstly removes rows and columns of the guards that cannot appear in the same sequence as i, and then iteratively removes the rows and columns of operations that are solely dependent (directly or indirectly) on these guards (`lines 31--42`). In short, Reduce correctly identifies all the possible dependency matrices beyond a branch point and ReformulateService, in turn, considers all the possible ways of realizability using those matrices. Therefore, if there exists an $L' \equiv L$ which is realized from CS, then the ReformulateService must return true. □

Complexity Analysis. The procedures DataFlowDep and Ctrl FlowDep perform backward depth-first traversal and their complexity is $O(|S| \times | \longrightarrow |)$ where $|S|$ and $| \longrightarrow |$ are the number of states and transitions, respectively, in a given LTS. The procedure Upda-

teCtrlFlowDep considers all possible paths of branches which will result in exponential complexity. However, the algorithm can be written by memorizing (recording) the set of operations that are possible from every state. In that case, the complexity reduces to that of backward depth-first traversal.

Algorithm 6 can be also made efficient by memorizing the arguments used for invoking ReformulateService such that repeated calls with the same arguments are not made. The exploration of the state-space is done in a depth-first fashion, where at each depth, the complexity is determined by the procedure CreateTransition used for realizing an operation from the component services. As a result, reformulation does not add to the overall complexity of the service composition algorithm.

5.4 Discussion

Realizing the full potential of the web as a platform for collaborative construction and deployment of largescale distributed software applications (services) requires effective techniques for composition of a composite service that realizes a specified functionality using a subset of available (often independently developed) component services [67, 102, 137]. Barring a few exceptions [149, 153], most current approaches to service composition [27, 33, 169], adopt a "single-step request-response" paradigm to service composition: If the composition algorithm fails to realize a composite service that satisfies the goal service specification using the available component services, the entire process fails, thereby shifting the responsibility of identifying the cause(s) for failure of composition as well as modification of the goal specifications to the service developer. Although our work in Chapter 4 has explored automated methods for identification of cause(s) of such failure, such approaches require laborious, and hence error-prone manual reformulation of the goal service specification even in settings where the composition algorithm fails to "mimic" the *structure* of the goal service specification using the available components even though it might be possible to do so using a functionally equivalent reformulation of the goal service specification.

To address this requirement, we have proposed a framework for composing Web services

through automatic reformulation of service specifications. We have described an efficient data structure, in the form of a *dependency matrix* to support analysis of the the data and control flow dependencies in goal service LTS specification. We show that any goal LTS reformulation that does not violate the data and control dependencies that are implicit in the specified goal service LTS specification is *provably functionally equivalent* to the specified goal service. We have described an efficient algorithm that is linear in the size of the goal LTS specification, and can generate an (as yet untried) alternative that is *functionally equivalent* to the user-supplied goal LTS. This process proceeds until a composite service is obtained (i.e., composition succeeds) or no alternative reformulations remain to be tried. The resulting framework can help limit in the need for the tedious manual intervention in reformulating specifications by limiting such interventions to settings where neither the specified goal LTS nor any of its its functionally equivalent reformulations (that conform to the data and control flow dependencies implicit in the goal specification) can be realized using the available component services. To the best of our knowledge, the work presented in this chapter, together with our previous results in Chapter 4, represent the first and important steps towards failure-based reformulation of service specifications in Web service composition. The main contributions can be summarized as follows:

- A characterization of the problem of modeling Web service composition via reformulation which amounts to automatic alteration of the composition specification without altering its overall functionality.

- A simple data structure and an algorithm for enabling automatic analysis of control and data flow dependencies in Web services.

- An end-to-end framework for on-the-fly Web service composition and reformulation of service specifications based on automata-theoretic approaches that provide formal guarantees in terms of soundness and completeness.

Some interesting directions for further research include:

1. Consideration of more expressive specifications than those captured by LTSs: Our current framework represents services using LTSs, which are essentially discrete-event systems. Some application scenarios require modeling of actions that extend over temporal intervals (e.g, duration of action `a` spans duration of action `b`). In this context, interval-based temporal representations (e.g., temporal algebra introduced by James Allen [9] would be interesting to explore.

2. Modeling asynchronous communications between services: Our current framework assumes synchronous communication between services. Asynchronous communication between services [55] requires buffers for managing messages between services. It would be interesting to explore extending our framework to such a setting.

3. Benchmarks and empirical evaluation: Work in progress is aimed at assessing the extent to which the our framework eliminates the need for manual intervention in real-world service composition scenarios, using benchmark service composition by leveraging the MoSCoE (`http://www.moscoe.org`) testbed.

CHAPTER 6. WEB SERVICE SUBSTITUTION

This chapter introduces the problem of Web service substitution and describes in details our approach to address the problem. The chapter is divided into four sections. The first section provides background to and describes the problem of Web service substitution. An illustrative example is described in the second section to explain the salient features of our proposal, which is discussed in the third section. The fourth section concludes the chapter with a discussion.

6.1 Introduction and Problem Description

In the earlier chapters (Chapters 4 and 5) of this thesis, we introduced the problems of Web service composition and specification reformulation, and proposed sound and complete techniques for addressing the issues concerning those problems. Specifically, our techniques provide a way to automatically generate composite services either directly from the goal service specification (represented as a labeled transition system, Definition 3), or in the event of failure of composition, by attempting to reformulation the specification without changing the "overall" functional requirements.

However, assembling a composite service that satisfy a desired set of requirements is only the first step. Ensuring that a composite service, once assembled, can be successfully deployed presents additional challenges that need to be addressed. Suppose a composite service Q relies on component/pre-existing services $Q_1 \cdots Q_n$. Consider a scenario wherein one of the component services, say Q_1, becomes unavailable either because the service provider for Q_1 chooses not to offer it any more or updates it (e.g., by adding/removing some of Q_1's features), thereby altering its behavior. Consequently, the behavior of the composite service Q that relies

on Q_1 is also altered. Because assembly of composite services in general is computationally costly, it is desirable to replace only the affected component(s) e.g. Q_1, with an alternative, say $Q_1^{'}$, while ensuring that the resulting composite service $Q^{'}$ obtained by replacing Q_1 with $Q_1^{'}$ can support (minimally) all of the functionality that was originally offered by Q.

As a result, identifying a component service that can substitute for another service has become an important problem in service-oriented computing. Of particular interest is the problem of determining whether a service can be replaced by another service in a specific *context* (or *property*) φ, which essentially refers to the functionality of the composition that must be preserved after the substitution. Previous solutions [25, 45, 117, 120] to this problem have relied on establishing functional or behavioral equivalence between the service that is being replaced and the replacement service (see Section 2.3.1 for related work).

We note that the requirement of functional/behavioral equivalence is stronger than that is often needed in practice for substituting one service with another. Hence, we introduce two variants of the *context-specific service substitutability* problem that are based on weaker and flexible requirements than those assumed by previous approaches [151]. The solution makes it possible to safely replace a service Q_1 with $Q_1^{'}$ within the context of a given composition, even though $Q_1^{'}$ may not meet the stronger requirement of being functionally or behaviorally equivalent to Q_1. More precisely, we represent a composition (denoted by $||$) of two services Q_1 and Q_2 that realizes a specific functionality or property (denoted by φ and expressed in temporal logic) by $Q_1 \,||\, Q_2 \models \varphi$. In the event Q_1 becomes unavailable, the goal is to identify candidates (Q_1') that can be used as replacement for Q_1 in the *environment* Q_2 and *property* φ. Similar to our previous work on service composition and specification reformulation, we represent services in our setting as labeled transition systems (Definition 3) and properties by mu-calculus [70, 105] formulas, and introduce the notion of *quotienting* such formulas. Informally, quotienting can be regarded as "factoring" an existing property φ by a system (Web services in our case), to yield another property ψ (in the same logic as φ). We show how the quotienting technique can be used to identify a substitute for another service within the specific environment and context of a particular composition in the remainder of this chapter.

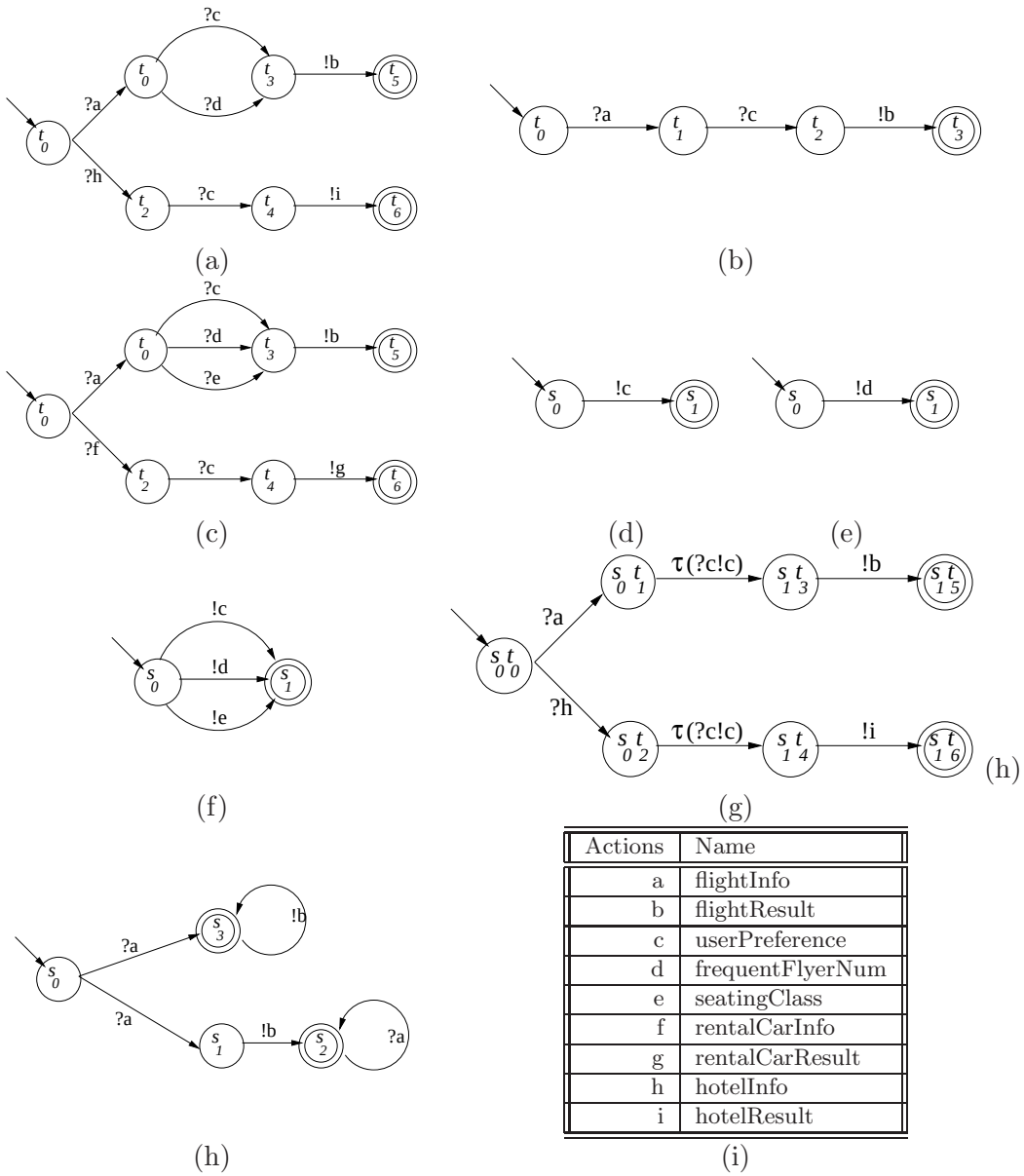
We begin with an illustrative example in the next section to explain the salient aspects of our technique.

6.2 Illustrative Example

Consider a setting wherein a traveler is interested in getting information about airline reservations by interacting with an existing Web service called `FunTravel`. This service is composed of two component services namely, `TravelSearch` (denoted by Q_1) and `ProfileInfo` (denoted by Q_2). Q_1 allows its clients to search for flight tickets as well as hotel rooms, whereas Q_2 stores and provides personal profile information (e.g., airline/hotel preferences) of its clients. An interaction between the client and `FunTravel` (and the two component services) can be described as follows: (*i*) First the client sends a message to Q_1 to search for a flight with required inputs (e.g., email address, departure/arrival cities); (*ii*) On receipt of the message, Q_1 interacts with Q_2 to retrieve client's profile information (e.g., airline preference); (*iii*) Once this information is received, Q_1 searches for available flight options, and sends the search results back to the client. Thus, the functionality (or *property* denoted by φ) realized by this composition is: *given an input for searching flight reservations, the composite service provides a list of available options* (if any). We will show later (Section 6.3.2) how to represent such properties in temporal logic using mu-calculus formulas.

Similar to our work described in previous chapters, we represent the behavioral description of the services as labeled transition systems (LTS, see Definition 3). Figures 6.1(a) & 6.1(d) shows the LTS representation of Q_1 and Q_2, respectively, whereas Figure 6.1(g) show their composition (see Definition 6). Note that, it is possible to compose Q_1 with Q_2' (Figure 6.1(e)) and Q_2'' (Figure 6.1(f)) as well because both $Q_1 \parallel Q_2' \models \varphi$ and $Q_1 \parallel Q_2'' \models \varphi$.

Now, assume that Q_1 becomes unavailable and needs to be substituted. Analyzing the sample services, it can be noticed that Q_1' (Figure 6.1(b)), which allows only to search for flight reservations, can act as a candidate replacement and can be composed with Q_2 (i.e., the environment) to satisfy the property φ. However, it cannot be replaced for *all* possible Q_2s because, for example, composition of Q_1' and Q_2' does not satisfy the required property



Actions	Name
a	flightInfo
b	flightResult
c	userPreference
d	frequentFlyerNum
e	seatingClass
f	rentalCarInfo
g	rentalCarResult
h	hotelInfo
i	hotelResult

(i)

Figure 6.1 LTS representation of Sample Services. (a) Q_1. (b) Q_1'. (c) Q_1''. (d) Q_2. (e) Q_2'. (f) Q_2''. (g) Composition of Q_1 and Q_2 ($Q_1 \parallel Q_2$). (h) Q_3. (i) Action-Name mapping.

(since Q_2' does not provide the user airline preference required by Q_1'). Typically, to identify a candidate replacement of Q_1 for all possible Q_2s, it is required that the candidate exhibits more functionality than Q_1. However, if the property φ is considered, the condition of substitutability can be relaxed. For example, Q_1'' (Figure 6.1(c)) can act as a replacement for Q_1 for all possible Q_2s since φ is satisfied in all the cases. It is worth mentioning that Q_1 and Q_1'' are *not* functionally/observationally/simulation equivalent.

6.3 Our Approach

6.3.1 Overview

Based on the above discussion, we introduce two variants to the problem of determining whether a service can be substituted by another as follows:

Environment-Independent Substitutability. *Given a property φ, and services Q_1 and Q_1', can Q_1' substitute Q_1 regardless of the environment of Q_1?* Formally, for specific Q_1, Q_1' and φ:

$$\forall Q_2 : (Q_1 \mid\mid Q_2 \models \varphi) \stackrel{?}{\Rightarrow} (Q_1' \mid\mid Q_2 \models \varphi) \tag{6.1}$$

i.e. can Q_1' replace Q_1 such that, when Q_1' is composed with *any* Q_2,[1] the resulting composition realizes φ? Observe that we only consider the Q_2s where $Q_1 \mid\mid Q_2 \models \varphi$. Compositions with Q_2s for which $Q_1 \mid\mid Q_2 \not\models \varphi$ (i.e., compositions that do not satisfy a desired property) are not interesting; the antecedent of the implication is false leading to satisfiability of the formula in Equation 6.1.

Note that this notion of substitutability is applicable in the setting where it is unknown apriori the services with whom the substitute (Q_1') is going to interact, and hence it is useful to guarantee that the substitute can minimally interact with *any* service (i.e., the environment) that can interact with the original (Q_1) [45]. A relaxed version of the above problem is where we consider the substitution only for a *particular* environment that is known apriori; a problem that we refer to as environment-dependent substitutability.

[1] In this case, Q_2 becomes the environment.

Environment-Dependent Substitutability. *Given a property φ, and services Q_1 and Q'_1, can Q'_1 substitute Q_1 for a specific environment of Q_1?* I.e., for a particular Q_1, Q'_1, Q_2 and φ, does $Q_1 \mid\mid Q_2 \models \varphi$ imply $Q'_1 \mid\mid Q_2 \models \varphi$? Notationally,

$$(Q_1 \mid\mid Q_2 \models \varphi) \stackrel{?}{\Rightarrow} (Q'_1 \mid\mid Q_2 \models \varphi) \tag{6.2}$$

Note that environment-independent substitutability implies environment-dependent substitutability, but not the other way around. Thus, the solution set for the latter is a superset of the solution set for the former.

To address the problems defined in Equations 6.1 and 6.2, we will use the technique of *quotienting*. As outlined above, quotienting of a property φ by Q, denoted by (φ/Q), results in a property ψ (in the same logic as φ) which if satisfied by Q' leads to $Q \mid\mid Q' \models \varphi$. Formally:

$$\forall Q' : (Q \mid\mid Q' \models \varphi) \Leftrightarrow (Q' \models (\varphi/Q))$$

Quotienting operation, therefore, captures the (temporal) *obligation* imposed by Q on its environment (Q') in order to satisfy φ.

Going back to Equation 6.1, the result of (φ/Q_1) denotes the property that must be satisfied by all the possible Q_2s such that $Q_1 \mid\mid Q_2 \models \varphi$. Similarly, (φ/Q'_1) must be satisfied by all the services (say Q'_2) such that $Q'_1 \mid\mid Q'_2 \models \varphi$. Proceeding further, if the set of Q_2s is a subset of set Q'_2s, then in all environments of Q_1 where φ is satisfied, Q'_1 when placed in those environments also satisfies φ. Therefore, the problem in Equation 6.1 can be reduced to satisfiability (model checking [70, 105]) of $(\varphi/Q_1) \Rightarrow (\varphi/Q'_1)$.

Next consider the problem in Equation 6.2. Here, (φ/Q_2) is the property that Q_1 satisfies when $Q_1 \mid\mid Q_2 \models \varphi$. In other words, Q'_1 must also satisfy (φ/Q_2) in order to be able to substitute Q_1 in the context of Q_2 and φ. The substitutability problem, therefore, can be reduced to satisfiability of (φ/Q_2) by Q'_1. I.e. Solution to Equation 6.2 holds if and only if $Q'_1 \models (\varphi/Q_2)$.

In what follows, we provide a brief introduction to a class of temporal logics called Mu-Calculus [70] and describe how to model Web service properties (φ) using Mu-Calculus.

1.	$[\![tt]\!]_e$	$=$	S
2.	$[\![ff]\!]_e$	$=$	$\emptyset$
3.	$[\![X]\!]_e$	$=$	$e(X)$
4.	$[\![\varphi_1 \wedge \varphi_2]\!]_e$	$=$	$[\![\varphi_1]\!]_e \cap [\![\varphi_2]\!]_e$
5.	$[\![\varphi_1 \vee \varphi_2]\!]_e$	$=$	$[\![\varphi_1]\!]_e \cup [\![\varphi_2]\!]_e$
6.	$[\![\langle a \rangle \varphi]\!]_e$	$=$	$\{s \mid \exists s \xrightarrow{a} s' \wedge s' \in [\![\varphi]\!]_e\}$
7.	$[\![[a]\varphi]\!]_e$	$=$	$\{s \mid \forall s \xrightarrow{a} s' \Rightarrow s' \in [\![\varphi]\!]_e\}$
8.	$[\![\mu X.\varphi]\!]_e$	$=$	$f^n_{X,e}(\emptyset)$
9.	$[\![\nu X.\varphi]\!]_e$	$=$	$f^n_{X,e}(S)$

Table 6.1 Semantics of Mu-Calculus formula

6.3.2 Representing Web Service Properties in Mu-Calculus

Mu-Calculus is an expressive logic with explicit least and greatest fixed point operators for representing temporal properties. It is more general than logics like LTL (linear temporal logic), CTL (computation tree logic), CTL*—properties expressed in these logics can be represented using mu-calculus.

The syntax of mu-calculus formulas is defined over a set of fixed point variables $\mathcal{X}$ and actions A as follows:

$$\phi \rightarrow tt \mid ff \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi \mid X \mid \sigma X.\phi$$

where $a \in A$, $X \in \mathcal{X}$ and $\sigma \in \{\mu, \nu\}$. The $\langle . \rangle$ and $[.]$ are modal operators referred to as diamond and box modalities, respectively. The operator μ is the least fixed point operator while ν is greatest fixed point operator. The formula of the form $\sigma X.\varphi$ is a fixed point formula where X is said to be *bound* by the fixed point operator σ. We will consider formulas that contain only bound variables. We will write $def(X) = \sigma X.\varphi$ for $\sigma X.\varphi$.

The semantics of mu-calculus formula φ, denoted by $[\![\varphi]\!]_e$ is given by the set of states of an LTS $M = (S, s_0, A, \Delta)$ which satisfies the formula. Here e is a mapping of the form $e : \mathcal{X} \rightarrow 2^S$. Table 6.1 presents the semantics of mu-calculus formulas using M and e. In the figure, the propositional constant tt is a satisfied by all states while ff is not satisfied by any state. The semantics of conjunctive and disjunctive formula expressions are the intersection and the union

of the semantics of the conjuncts and disjuncts, respectively. $\langle a\rangle\varphi$ is satisfied by states which have at least one a-successor that satisfies φ. The dual $[a]\varphi$ is satisfied by the states whose all a-successors satisfy φ. The semantics of fixed point variable X is defined by the mapping function e. Finally, semantics of least and greatest fixed point formula expressions are defined using the function $f_{X,e}(\hat{S}) = [\![\varphi]\!]_{e[X\mapsto\hat{S}]}$ where $def(X) = \sigma X.\varphi$ and $\hat{S} \subseteq S$. Here, $e[X \mapsto S']$ denotes an update to the mapping function such that $e[X \mapsto S'](Y) = S'$ if $X = Y$ and $e(Y)$ otherwise. It can be immediately shown that $f_{X,e} : 2^S \to 2^S$ is monotonic over the lattice of subsets of state-set S, i.e. for all $S_1 \subseteq S_2 \subseteq S$: $f_{X,e}(S_1) \subseteq f_{X,e}(S_2)$. Following Tarski-Knaster theorem [183], the fixed point semantics as n applications of the function $f_{X,e}$ where $n = |S|$. The semantic-computation of least fixed point starts from the bottom of the subset-lattice $\emptyset$ while that of the greatest fixed point proceeds from the top element in the lattice S. We will use the above semantic definition in the subsequent sections.

We say that an LTS $M = (S, s_0, A, \Delta)$ satisfies a fixed point formula φ ($M \models \varphi$) if and only if $s_0 \in [\![\varphi]\!]_e$. Note that, if φ only contains bounded fixed point variables then its semantics is independent of e. We will use $s \in [\![\varphi]\!]$ and $s \models \varphi$ interchangeably.

Example 2 *Consider the LTS Q_2 shown in Figure 6.1(d) where s_0 is the start state. We want to verify whether the $M \models \varphi$ where φ is defined as $\mu X.\langle !c\rangle \mathrm{tt} \vee \langle -\rangle X$. We use "-" as a short-hand to "any" action. The semantic computation proceeds as follows:*

$$\begin{aligned} f_{X,e}(\emptyset) &= [\![\langle c!\rangle \mathrm{tt} \vee \langle -\rangle X]\!]_{e[X\mapsto\emptyset]} = \{s_0\} \\ f^2_{X,e}(\emptyset) &= f_{X,e}(f_{X,e}(\emptyset)) = [\![\langle !c\rangle \mathrm{tt} \vee \langle -\rangle X]\!]_{e[X\mapsto f_{X,e}(\emptyset)]} \\ &= [\![\langle !c\rangle \mathrm{tt} \vee \langle -\rangle X]\!]_{e[X\mapsto\{s_0\}]} = \{s_0\} \end{aligned}$$

The computation can be terminated as fixed point is reached and the semantics is $\{s_0\}$. The formula is satisfied by states which eventually reach a state that has an "!c" transition. Therefore, $M \models \varphi$ as $s_0 \models \varphi$.

Consider next a formula $\nu X.[a]\mathrm{ff} \wedge [-]X$. The set of states in S that satisfies the formula is $\{s_3\}$. The formula holds in states whose all reachable states do not have an a transition.

The preceding example contains one fixed point variable, although in general multiple fixed

point variables may appear in a formula resulting in a *nested* fixed point formula. The nesting depth of the formula is defined by the number of nestings of fixed point formula expressions present in the formula. We will use $nd(\varphi)$ to denote nesting depth of the formula φ.[2]

Example 3 *The formula* $\nu X.(\mu Y.(\langle ?a\rangle \mathrm{tt} \vee \langle -\rangle Y) \wedge [-]X)$ *presents the nesting of a least fixed point formula inside a greatest fixed point one. It is satisfied by LTS states which can only reach states where* $\mu Y.(\langle ?a\rangle \mathrm{tt} \vee \langle -\rangle Y)$ *is satisfied. The states* s_0, s_1 *and* s_2 *of LTS in Figure 6.1(h) satisfies this property.*

Going back to the example in Figure 6.1, composition of Q_1 and Q_2 realizes the functionality or the property where *after the client sends an input message for searching flight reservations (*`?a`*), the composite service eventually provides an output message (*`!b`*) with the list of available options*. No other input/output is demanded/provided from/to the client. We will refer to the required functionality as φ which can be represented as:

$$\langle \texttt{?a}\rangle \mu X.\ (\langle \texttt{!b}\rangle tt \vee \langle \tau\rangle X) \tag{6.3}$$

The formula represents the behavior where an `?a` action is followed by a `!b` action after finitely many τ steps, where `?a` corresponds to flight input information to be used for search and `!b` corresponds to the search results.

6.3.3 Quotienting Mu-Calculus Properties

We now proceed to describe the quotienting of a mu-calculus property (or formula) against an LTS. Given a formula φ and an LTS Q, quotienting (φ/Q) results in a formula ψ which must be satisfied by the environment of Q such that the overall composition satisfies φ. Quotienting of φ against Q is equivalent to the quotienting of φ against s_0, the start state of Q. Such techniques have been used to solve problems in (a) model checking ring protocols [13], (b) verification of parameterized systems [194] and (c) controller synthesis of discrete event systems [22]. Each of these techniques define quotienting on the basis of the definition of

[2] A more general form of nested formula, referred to as *alternating fixed point*, is one where the inner fixed point may refer to the formula name defined by the outer fixed point. Refer to [11, 12, 168] for details.

composition of two components and with respect to the specific domain being considered. In particular, [24, 194] introduces quotienting of equational mu-calculus against CCS [130] process expressions and uses it to analyze compositions containing unbounded number of components. In [22], on the other hand, quotienting of mu-calculus is used for controller synthesis taking into consideration the controller-problem specific requirements, e.g. controllability constraint. The closest to our notion of quotienting is the work by Andersen [13] where synchronous composition of LTSs is used to quotient equational mu-calculus formulas.

We will define the quotienting function $(\varphi/_{T,R}s)$ as $/ : \Phi \times S \times \mathcal{R} \times \mathcal{T} \rightarrow \Phi$ where $\varphi \in \Phi$, $s \in S$ of an LTS Q, $R \in \mathcal{R}$ is the restricted action set (the actions on which Q must synchronize with its environment) and $T \in \mathcal{T}$ is a tag set. The tag set contains elements of the form X_i^s where X is a fixed point variable in φ, $s \in S$ and i is an integer. The tag set is necessary to ensure termination of the recursive quotienting. The result of $(\varphi/_{T,R}s)$ is another mu-calculus formula that must be satisfied by the *environment* state t such that $(s,t) \models \varphi$ under the restriction R.

Figure 6.2 presents the quotienting function. Each rule follows from the semantics of mu-calculus formula expression described in Table 6.1. Rule 1 states that any environment state when composed with s can satisfy tt while Rule 2 states that there is no environment state that can be composed with s to satisfy ff.

Rules 3 and 4 follow from the fact that semantics of conjunctive and disjunctive formulas are intersection and union of the semantics of conjuncts and disjuncts, respectively.

Rule 5 handles quotienting of diamond modal formula expressions. There are three possible cases by which (s,t), where t is the environment state composed with s, can satisfy $\langle a \rangle \varphi$. Each case leads a separate disjunct in the result of quotienting:

- t can make a move on a to t' such that (s,t') satisfies φ. This is represented by the first disjunct where the environment state (in this case t) is left with the obligation to satisfy the diamond modality $\langle a \rangle$ and at least one its a-successor must satisfy the result of $(\varphi/_{T,R}s)$.

- The second case corresponds to the case when $a = \tau$ and there exists transitions from s

1. $(tt/_{T,R}s) = tt$

2. $(ff/_{T,R}s) = ff.$

3. $(\varphi_1 \wedge \varphi_2/_{T,R}s) = (\varphi_1/_{T,R}s) \wedge (\varphi_2/_{T,R}s).$

4. $(\varphi_1 \vee \varphi_2/_{T,R}s) = (\varphi_1/_{T,R}s) \vee (\varphi_2/_{T,R}s).$

5. $$(\langle a\rangle\varphi/_{T,R}s) = \langle a\rangle(\varphi/_{T,R}s) \vee \begin{cases} \left(\bigvee_{s':s\xrightarrow{c}s'} \langle b\rangle(\varphi/_T s')\right) & \\ \qquad \textbf{if } a = \tau \wedge \exists s' : s \xrightarrow{c} s' & \\ \qquad\quad \wedge\ inv(b,c)\ \wedge\ b,c \in R & \\ ff \qquad \textbf{otherwise} & \end{cases} \vee \begin{cases} \left(\bigvee_{s':s\xrightarrow{a}s'} (\varphi/_T s')\right) & \\ \qquad \textbf{if } \exists s' : s \xrightarrow{a} s' \wedge\ a \notin R & \\ ff \qquad \textbf{otherwise} & \end{cases}$$

6. $$([a]\varphi/_{T,R}s) = [a](\varphi/_{T,R}s) \wedge \begin{cases} \left(\bigwedge_{s':s\xrightarrow{c}s'} [b](\varphi/_T s')\right) & \\ \qquad \textbf{if } a = \tau \wedge \exists s' : s \xrightarrow{c} s' & \\ \qquad\quad \wedge\ inv(b,c)\ \wedge\ b,c \in R & \\ tt \qquad \textbf{otherwise} & \end{cases} \wedge \begin{cases} \left(\bigwedge_{s':s\xrightarrow{a}s'} (\varphi/_T s')\right) & \\ \qquad \textbf{if } \exists s' : s \xrightarrow{a} s' \wedge\ a \notin R & \\ tt \qquad \textbf{otherwise} & \end{cases}$$

7. $$(\sigma X.\varphi_x/_{T,R}s) = \begin{cases} \sigma X_i^s.(\varphi_x/_{T\cup\{X_i^s\},R}s) \textbf{ if } X_i^s \notin T & \\ \sigma X_{i+1}^s.(\varphi_x/_{T[X_i^s/X_{i+1}^s],R}s) & \\ \quad \textbf{otherwise} & \end{cases}$$

8. $$(X/_{T,R}s) = \begin{cases} X_i^s \ \textbf{ if } \ X_i^s \in T & \\ (\sigma X.\varphi_x/_{T,R}s) \textbf{ otherwise} & \\ \qquad \textbf{where } def(X) = \sigma X.\varphi_x & \end{cases}$$

Figure 6.2 Quotienting Rules

and t on which they can synchronize and move to s' and t', respectively, such that (s', t') satisfies φ. This case represents the situation when both s and t makes a synchronous move. As such the second disjunct in quotienting imposes on the environment to satisfy at least one diamond modal obligation $\langle b \rangle$ when s has a c-successor and b and c are inverse of each other. Further, b and c must be present in the restricted set.

- Finally, the state s can satisfy the diamond obligation $\langle a \rangle$. This case corresponds to the situation when s makes a move on a while t remains idle.

Note that quotienting automatically handles the possible non-determinism at the state s by considering disjunction over the all the relevant outgoing transitions. The Rule 6 is the dual of Rule 5 and can be similarly explained.

Rules 7 and 8 represent the quotienting of fixed point formula expressions and fixed point formula variables. The rules closely follow the fixed point semantics as presented in Section 6.3.2. Consider $(\sigma X.\varphi /_{T,R} s)$. Recall that (s, t) belongs to the semantics of $\sigma X.\varphi$ if it belongs to the semantics of φ. Quotienting $\sigma X.\varphi$ results in a new formula over fixed point variable X_i^s (case 1 of Rule 7). The new variable X_i^s is added to the tag set T. Case 2 in Rule 7 states that if X_i^s is already present in the tag set denoting that $\sigma X.\varphi$ has already been quotiented against s (i times), then a new formula variable X_{i+1}^s is used and the tag set is appropriately updated; $T[X_i^s / X_{i+1}^s]$ means that X_i^s is replaced by X_{i+1}^s in T.

The new formula generated from quotienting φ against s may lead to quotienting X against s. The situation corresponds to the case where $(s, t) \in [\![\sigma X.\varphi]\!]_e$ when $(s, t) \in e(X)$. As such, quotienting of X against s is equal to X_i^s (the last fixed point variable resulting from quotienting of $\sigma X.\varphi$ against s). This is shown in Rule 8, case 1. On the other hand, if quotienting φ against s leads to quotienting X against s' where s' has not be used to quotient $\sigma X.\varphi$ before, the situation corresponds to the case where $(s, t) \in [\![\sigma X.\varphi]\!]_e$ when $(s', t') \in e(X)$. Furthermore, since s' has not been used to quotient $\sigma X.\varphi$, it implies that $(s', t') \in e(X)$ can only occur if $(s, t) \in f^k_{X,e}(\hat{S})$ and $(s', t') \in f^{k-1}_{X,e}(\hat{S})$. This leads to case 2 in Rule 8 where X is replaced by its definition and quotiented against the state (s' in the above example case) under consideration.

$(\varphi/_{\emptyset,R} Q_1)$
$= (\langle a? \rangle \mu X.(\langle b! \rangle tt \vee \langle \tau \rangle X))/t_0$
$= (\langle a? \rangle \mu X_1^{t_0}.(\langle b! \rangle tt \vee \langle \tau \rangle X_1^{t_0})) \vee \varphi_{X_1^{t_1}}$ Rules 5, 1

$\varphi_{X_1^{t_1}}$
$= \mu X_1^{t_1}.(\langle b! \rangle tt \vee \langle c! \rangle \varphi_{X_1^{t_3}} \vee \langle d! \rangle \varphi_{X_1^{t_3}} \vee \langle \tau \rangle X_1^{t_1})$ Rules 7, 5, 4, 1

$\varphi_{X_1^{t_3}}$
$= \mu X_1^{t_3}.(tt)$ Rules 7, 5, 1

(i)

$(\varphi/_{\emptyset,R} Q_1'')$
$= (\langle a? \rangle \mu X.(\langle b! \rangle tt \vee \langle \tau \rangle X))/t_0$
$= (\langle a? \rangle \mu X_1^{t_0}.(\langle b! \rangle tt \vee \langle \tau \rangle X_1^{t_0})) \vee \varphi_{X_1^{t_1}}$ Rules 5, 1

$\varphi_{X_1^{t_1}}$
$= \mu X_1^{t_1}.(\langle b! \rangle tt \vee \langle c! \rangle \varphi_{X_1^{t_3}} \vee \langle d! \rangle \varphi_{X_1^{t_3}} \vee \langle e! \rangle \varphi_{X_1^{t_3}}$ Rules 7, 5, 4, 1
$\vee \langle \tau \rangle X_1^{t_1})$

$\varphi_{X_1^{t_3}}$
$= \mu X_1^{t_3}.(tt)$ Rules 7, 5, 1

(ii)

Figure 6.3 Results of quotienting φ (Equation 6.3) by: (i) Q_1 (Figure 6.1(a)) and (ii) Q_1'' (Figure 6.1(c))

Theorem 4 states that the expression φ, with a nesting depth of $nd(\varphi)$, can be *at most* quotiented $|S|^{nd(\varphi)}$ times by a particular state.

Example 4 *Consider the sample services Q_1 and Q_1'' in Figure 6.1(a, c) and the mu-calculus formula φ in Equation 6.3. The goal is to verify whether Q_1'' can substitute Q_1 for all possible Q_2s in Figure 6.1(d, e, f) in the context of φ. The results, $\psi = (\varphi/_{\emptyset,R} Q_1)$ and $\psi'' = (\varphi/_{\emptyset,R} Q_1'')$, are shown in Figure 6.3 where $R = \{?c, !c, ?d, !d\}$. Note that quotienting by Q_1'' generates a formula ψ'' which is same as ψ, implying that $\psi \Rightarrow \psi''$ is satisfiable, i.e. Q_1 can be indeed substituted by Q_1'' in the context of φ for all possible environments Q_2s.*

6.3.4 Substitutability of Web services

We now proceed to show that the environment-independent and environment-dependent variants of the service substitutability problem introduced in Section 6.3.1 (Equations 6.1 and

6.2) can be reduced to mu-calculus *satisfiability* using the notion of quotienting presented above.

6.3.4.1 Environment-Independent Substitutability

In this case, the problem is to determine whether Q_1' can replace Q_1 for *all* possible environments Q_2s for which $Q_1 \parallel Q_2 \models \varphi$ (Equation 6.1). Thus, the property to be satisfied by all Q_2s, such that $Q_1 \parallel Q_2 \models \varphi$, is $(\varphi /_{\emptyset,R} Q_1)$. Similarly, the obligation on possible environments of Q_1' (say Q_2's) is $(\varphi /_{\emptyset,R} Q_1')$.

If the set of Q_2s is a subset of Q_2's, then the following holds:

$$\begin{aligned} \forall Q_2 : (Q_1 \parallel Q_2)\backslash R \models \varphi \quad &\Leftrightarrow \quad Q_2 \models (\varphi /_{\emptyset,R} Q_1) \\ &\Rightarrow \quad Q_2 \models (\varphi /_{\emptyset,R} Q_1') \\ &\Leftrightarrow \quad (Q_1' \parallel Q_2)\backslash R \models \varphi \end{aligned}$$

The environment-independent substitutability is, therefore, reduced to satisfiability of $(\varphi /_{\emptyset,R} Q_1) \Rightarrow (\varphi /_{\emptyset,R} Q_1')$.

6.3.4.2 Environment-Dependent Substitutability

Assume that the composition of services Q_1 and Q_2 under the restriction R realizes the functionality described by the mu-calculus formula φ: $(Q_1 \parallel Q_2)\backslash R \models \varphi$. In the event it is required to replace Q_1 by Q_1', it suffices to verify whether Q_1' satisfies $(\varphi /_{\emptyset,R} Q_2)$. The verification of $Q_1' \models (\varphi /_{\emptyset,R} Q_2)$ can be done using mu-calculus model checkers which takes as input the mu-calculus formula, LTS and returns true or false depending on whether the LTS satisfies the formula or not (using semantics of mu-calculus as described in Section 6.3.2). If the Q_1' satisfies $(\varphi /_{\emptyset,R} Q_2)$, it follows that $(Q_1' \parallel Q_2)\backslash R \models \varphi$. Therefore, Q_1' can replace Q_1 in the environment in which Q_1 is composed with Q_2 to satisfy φ (Equation 6.2).

6.3.4.3 Mu-Calculus Satisfiability

Satisfiability of mu-calculus formula is performed by reducing the problem to emptiness problem of alternating tree automata [71] or identifying the winning strategy in a parity game

[18, 22, 182]. Details of the technique are beyond the scope of this thesis. At a high-level, these techniques determines the satisfiability of mu-calculus formula on the basis of satisfiability of its subformulas and take special care to handle fixed point satisfiability. The complexity of satisfiability checking, is therefore, exponential to the number of subformulas of the formula under consideration.

6.3.5 Theoretical Analysis

Theorem 4 (Quotienting & Nesting Depth) *Given a fixed point expression φ in mu-calculus, the number of times the expression is quotiented by a state has an upper bound of $|S|^{\mathrm{nd}(\varphi)}$, where $\mathrm{nd}(\varphi)$ is the nesting depth of formula φ.*

Proof Sketch: For $nd(\varphi) = 1$, the proof of the above statement is trivial since for a formula expression of the form $\sigma X.\varphi_x$, φ is quotiented at most $|S|$ times (see Rules 7 and 8). Next consider the case, where $\sigma X.\varphi_x$ and $\sigma Y.\varphi_y$ are two subformulas in φ with the former being the outer formula expression. Furthermore, let X be a subformula of φ_y. Quotienting $\sigma X.\varphi_x$ against s may lead to quotienting $\sigma Y.\varphi_y$ against s which in turn may lead to quotienting X against s'. According to Rule 8, quotienting X against s' is defined as quotienting $\sigma X.\varphi_x$ against s'. This may again lead to quotienting $\sigma Y.\varphi_y$ against s. As there are $|S|$ states, the number of times $\sigma Y.\varphi_y$ can quotiented by the *same* state s is of the order $O(|S|)$. Proceeding further, the number of times $\sigma Y.\varphi_y$ is quotiented by *any* state is, therefore, $O(|S|^2)$. For the general case, let $g(n)$ be the number of times a fixed point expression in φ is quotiented by any state when $nd(\varphi) = n$. Now, let us construct a new formula expression $\sigma Z.\varphi_z$ such that φ is a subformula of φ_z and Z is a subformula in φ. That is, the nesting depth of $\sigma Z.\varphi_z$ is $n+1$ and $\sigma Z.\varphi_z$ can be quotiented by every state in the LTS such that, for each such quotienting operation, the inner formula φ will be quotiented $g(n)$ times (induction hypothesis). Therefore, the total number of times a formula in $\sigma Z.\varphi_z$ is quotiented against any state is $|S| \times g(n)$, i.e. $g(n+1) = |S| \times g(n)$. Proceeding further, $\forall i \geq 1.g(i) = |S|^i$.

Theorem 5 (Soundness & Completeness) *Given $Q_1 = (S_1, s_{01}, A_1, \Delta_1)$, $Q_2 = (S_2, s_{02},$*

A_2, Δ_2), the restriction set R and a mu-calculus formula φ, the following holds:

$$((Q_1 \mid\mid Q_2)\backslash R \models \varphi) \Leftrightarrow (Q_2 \models (\varphi/_{\emptyset,R} Q_1))$$

Proof: The proof follows from the discussion in Sections 6.3.2 and 6.3.3.

Complexity of Quotienting. The complexity of quotienting operation can be derived from the size of the result of the quotient. Given a formula φ and the set of states S against which it is quotiented, the number of times each subformula in φ is quotiented by each state in S is $|S|^{nd(\varphi)}$ (see above) which is also the nesting depth of the resultant quotient. Next, observe that the Rules 5 and 6 considers all (*matching*) outgoing transitions of the participating state and generates modal obligation following the transitions. As such, the size of the quotient is amplified by a factor of B where B is the maximum branching factor of the LTS. The overall size of the quotient is $O(|\varphi| \times |S|^{nd(\varphi)} \times B)$, where $|\varphi|$ is the size of φ.

Complexity of Substitutability. Recall that, quotienting φ against an LTS containing set of states S results in a formula (say ψ) of size $O(|\varphi| \times |S|^{nd(\varphi)} \times B)$ and nesting depth $|S|^{nd(\varphi)}$ (see above). Complexity of satisfiability of ψ is exponential to the number of subformulas in ψ. Note that at each nesting depth in ψ the number of subformulas is $O(|\varphi| \times B)$; therefore the complexity for satisfiability checking is $O(|S|^{nd} \times 2^{|\varphi| \times B})$.

For Q_1, determining whether Q_1' can replace Q_1 in an environment independent fashion in the context of φ, has the complexity $O(\texttt{max}(|S_1|^{nd(\varphi)}, |S_1'|^{nd(\varphi)}) \times 2^{|\varphi| \times \texttt{max}(B_1, B_1')})$ where S_1, B_1 and S_1', B_1' are set of states and maximum branching factors of Q_1 and Q_1', respectively.

6.4 Discussion

Determining substitutability of a service with another is an important problem in service-oriented computing. In this chapter, we focus on the problem of context-specific service substitution which requires that some desired property φ of the component being replaced is maintained despite its substitution by another component. We introduce two variants

of the context-specific service substitutability problem, namely, environment-dependent and environment-independent substitutability that relax the requirements for substitutability relative to simulation or observational equivalence between services. The proposed solution to these two problems is based on the well-studied notion of "quotienting" which is used to identify the obligation of the environment of a service being replaced in a specific context. We demonstrate that both environment-dependent and environment-independent service substitutability problems can be reduced to quotienting of φ against the service being replaced and the replacement service and hence, to satisfiability of the corresponding mu-calculus formulae. The correctness of our technique follows from the correctness of the individual steps of quotienting and satisfiability.

In the current setting, we did not take into consideration the data parameters, i.e. messages being exchanged by the services. In our work on composition (Chapter 4), we have discussed equivalence between services where the communication paradigm includes messages that can potentially have an infinite domain. Thus, investigation of quotienting-based approach to context-specific substitutability to the setting of message-based communication is required. One possibility is to explore the applicability of value-passing LTS/CCS and more powerful value-passing mu-calculus [194]. Furthermore, similar to our approach on composition, we assumed synchronous communication between the services. Hence, consideration of services which communicate asynchronously [55] and analysis for substitutability in such a setting is a topic of potential research. In addition to the above, adopting this work into a more dynamic setting where services can be replaced at runtime by automatic re-composition that takes into consideration not only the functional, but also the non-functional requirements for substitution, will enable realization of a complete end-to-end solution for Web service substitution. Finally, an interesting aspect that deserves further research is analyzing failure of substitution, i.e., what action can be taken when an existing service cannot be replaced by another.

CHAPTER 7. SEMANTIC INTEROPERABILITY

This chapter introduces the problem of semantic interoperability in service-oriented computing and describes in details our approach to address some aspects of the problem. The chapter is divided into four sections. The first section provides background to and describes the semantic interoperability problem. Preliminaries on ontologies and inter-ontology mappings is described in the second section to explain the salient features of our proposal, which is discussed in the third section. The fourth section concludes the chapter with a discussion.

7.1 Introduction and Problem Description

The work on service composition (Chapter 4), specification reformulation (Chapter 5), and substitution (Chapter 6) presented in this thesis so far makes an assumption that the *vocabulary* used to represent the services is uniform throughout. That is, there is no syntactic and/or semantic inconsistency between the various messages, message types, action names, and so on between the services that are being analyzed for composition, reformulation and substitution. However, it is unrealistic to expect such syntactic and semantic consistency across independently developed service libraries. Each such library is typically based on an implicit *ontology* [88], which reflects the assumptions concerning the objects that exist in the world, the properties or attributes of the objects, the possible values of attributes, and their intended meaning from the point of view of the creators of the services in question. Because the services that are created for use in one context often find use in other contexts or applications, syntactic and semantic differences between independently developed services are unavoidable.

For example, consider a composition comprising of two simple component services: F-Sensor and Weather Description, as shown in Figure 7.1. The objective of this composite

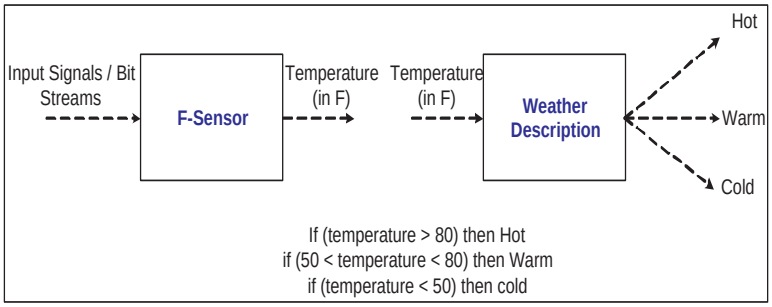


Figure 7.1 Weather Description with F-Sensor

service is to determine whether the day is hot, or warm or cold based upon the temperature. The input to the F-Sensor component consists of signals from one or more sensors and its output is the current temperature (in degree F) and the input to the Weather Description component is the current temperature (in degree F from the output of F-Sensor component) and its output is a description of the day (hot or warm or cold). Note that in this example, the output produced by the F-Sensor component has the same semantics as the input of the Weather Description component; furthermore, the name `Temperature` used in the vocabulary associated with the F-Sensor component has the same *meaning* as the term `Temperature` in the vocabulary associated with the Weather Description component. In the absence of syntactic or semantic mismatches between the components, their composition is straightforward.

Now, consider the scenario where we replace the F-Sensor component with a new component: C-Sensor. Suppose C-Sensor behaves very much like F-Sensor except that it outputs the temperature, denoted by `Temp`, and measured in degree Centigrade instead of degree Fahrenheit. Now we can no longer compose C-Sensor and Weather-Description components because of the syntactic and semantic differences between the two components. Effective use of independently developed components in a given context requires reconciliation of such syntactic and semantic differences between them. Because of the need to define compositions in different application contexts in terms of vocabulary familiar to users of the composite service, there is no single privileged ontology that will serve all users, or for that matter, even a single user, in all context.

A similar argument also applies in the case of discovering and executing component services

where differences in semantics can lead to unprecedented failures.

Consequently, there is a need to overcome this major hurdle in the reuse of independently developed services in new applications that arise from the semantic differences between the services. Towards this end, realizing the vision of the Semantic Web [34], i.e., supporting seamless access and use of information sources and services on the Web, we build on recent developments in ontology-based solutions on information integration [58, 171] to develop principled solutions to addressing the semantic interoperability problem in service-oriented computing. Specifically, we introduce *ontology-extended components* and *mappings* between ontologies to facilitate discovery [158] and composition [156] of semantically heterogeneous component services.

We begin by providing background on ontologies and mappings between ontologies in the next section.

7.2 Ontologies and Mappings

An *ontology* is a specification of *objects*, *categories*, *properties* and *relationships* used to conceptualize some domain of interest. In what follows, we introduce a precise definition of ontologies.

Definition 10 (Hierarchy [43]) *Let S be a partially ordered set under ordering $\leq$. We say that an ordering $\preceq$ defines a hierarchy for S if the following three conditions are satisfied:*

(1) $x \preceq y \rightarrow x \leq y$; $\forall\ x, y \in S$. We say $(S, \preceq)$ is better than $(S, \leq)$),

(2) $(S, \leq)$ is the reflexive, transitive closure of $(S, \preceq)$,

(3) No other ordering $\sqsubseteq$ satisfies (1) and (2).

For example, let $S = \{$`Weather, Wind, WindSpeed`$\}$. We can define the partial ordering $\leq$ on S according to the *part-of* relationship. For example, `Wind` is part of the `Weather` characteristics, `WindSpeed` is part of the `Weather` characteristics, and `WindSpeed` is also part of `Wind` characteristics. Besides, everything is part of itself. Thus, $(S, \leq) = \{$(`Weather, Weather`), (`Wind,`

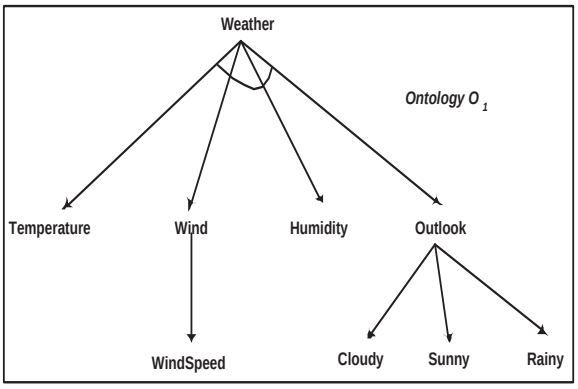


Figure 7.2 Weather Ontology of Company K_1

`Wind`), (`WindSpeed, WindSpeed`), (`Wind, Weather`), (`WindSpeed, Weather`), (`WindSpeed, Wind`)}. The reflexive, transitive closure of $\leq$ is the set: $(S, \preceq) = \{$(`Wind, Weather`), (`WindSpeed, Wind`)$\}$, which is the only hierarchy associated with $(S, \leq)$.

Definition 11 (Ontologies [43]) *Let Δ be a finite set of strings that can be used to define hierarchies for a set of terms S. For example, Δ may contain strings like is-a, part-of corresponding to is-a or part-of relationships, respectively. An Ontology O over the terms in S with respect to the partial orderings contained in Δ is a mapping Θ from Δ to hierarchies in S defined according to the orderings in Δ. In other words, an ontology associates orderings to their corresponding hierarchies. Thus, if part-of $\in \Delta$, then Θ(part-of) will be the part-of hierarchy associated with the set of terms in S.*

For example, suppose a company K_1 records information about weather in some region of interest (see Figure 7.2). From K_1's viewpoint, weather is described by the attributes `Temperature, Wind, Humidity` and `Outlook` which are related to weather by *part-of* relationship. Assume that `Wind` is described by `WindSpeed`. The values `Cloudy, Sunny, Rainy` are related to `Outlook` by the *is*-a relationship. In the case of a measurement (e.g., `Temperature, WindSpeed`) a unit of measurement is also specified by the ontology. In K_1's ontology, O_1, `Temperature` is measured in degrees Fahrenheit and the `WindSpeed` is measured in miles per hour. For contrast, an alternative ontology of weather O_2 from the viewpoint of a company K_2 is shown in Figure 7.3.

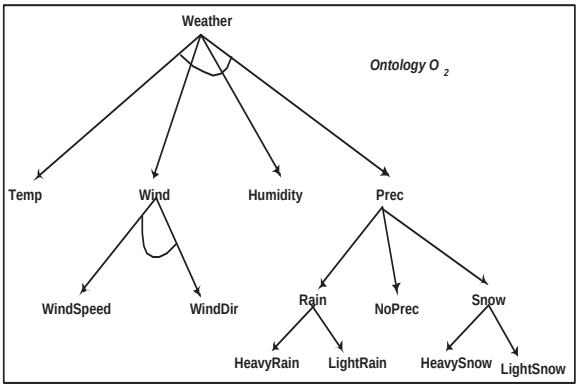


Figure 7.3 Weather Ontology of Company K_2

Suppose O_1,...,O_n are ontologies associated with components C_1,...,C_n, respectively. In order to compose such semantically heterogeneous components, the user (i.e., the service developer) needs to specify the mappings between these ontologies of the various components. For example, a company K_3, with ontology O_3 uses meteorology components supplied by K_1 and K_2. Suppose in O_3, `Weather` is described by `Temperature` (measured in degrees Fahrenheit), `WindSpeed` (measured in mph), `Humidity` and `Outlook`. Then, K_3 will have to specify a suitable mapping $M_{O_1 \mapsto O_3}$ from K_1 to K_3 and a mapping $M_{O_2 \mapsto O_3}$ from K_2 to K_3. For example, `Temperature` in O_1 and `Temp` in O_2 may be mapped by $M_{O_1 \mapsto O_3}$ and $M_{O_2 \mapsto O_3}$ respectively to `Temperature` in O_3. In addition, conversion functions to perform unit conversions (e.g. `Temp` values in O_2 from degrees Centigrade to degrees Fahrenheit) can also be specified. Suppose K_3 considers `Precipitation` in O_2 is equivalent to `Outlook` in O_3 and maps `Rain` in O_2 to `Rainy` in O_3. This would implicitly map both `LightRain` and `HeavyRain` in O_2 to `Rainy` in O_3. These mappings between ontologies are specified through *interoperation constraints*.

Definition 12 (Interoperation Constraints [43, 58]) *Let $(H_1, \preceq_1)$ and $(H_2, \preceq_2)$, be any two hierarchies. We call set of Interoperation Constraints (IC) the set of relationships that exists between elements from two different hierarchies. For two elements, $x \in H_1$ and $y \in H_2$, we can have one of the following Interoperation Constraints:*

- $x : H_1 = y : H_2$
- $x : H_1 \neq y : H_2$

- $x : H_1 \leq y : H_2$
- $x : H_1 \not\leq y : H_2$

For example, in the weather domain, if we consider *part-of* hierarchies associated with the companies K_1 and K_2, we have the following interoperation constraints: `Temperature` : $1 =$ `Temp` : 2, `Outlook` : $1 =$ `Prec` : 2, `Humidity` : $1 \neq$ `Wind` : 2, `WindDir` : $2 \not\leq$ `Wind` : 1, and so on.

Definition 13 (Type, Domain, Values) *We define $T = \{\tau \mid \tau$ is a string$\}$ to be a set of types. For each type τ, $D(\tau) = \{v|v$ is a value of type $\tau\}$ is called the domain of τ. The members of $D(\tau)$ are called values of type τ. For instance, a type τ could be a predefined type, e.g. int or string or it can be a type like USD (US Dollars) or kmph (kilometers per hour).*

Definition 14 (Type Conversion Function) *We say that a total function $f(\tau_1, \tau_2)$: $D(\tau_1) \mapsto D(\tau_2)$ that maps the values of τ_1 to values of τ_2 is a type conversion function from τ_1 to τ_2. The set of all type conversion functions satisfy the following constraints:*

- *For every two types τ_i, $\tau_j \in T$, there exists at most one conversion function $f(\tau_i, \tau_j)$.*
- *For every type $\tau \in T$, $f(\tau, \tau)$ exits. This is the identity function.*
- *If $f(\tau_i, \tau_j)$ and $f(\tau_j, \tau_k)$ exist, then $f(\tau_i, \tau_k)$ exists and $f(\tau_i, \tau_k) = f(\tau_i, \tau_j) \circ f(\tau_j, \tau_k)$ is called a composition function.*

7.3 Our Approach

Based on the definitions described above, we outline our approach for discovering (Section 7.3.1) and composing (Section 7.3.2) semantically heterogeneous component services in the following sections.

7.3.1 Ontology-based Service Discovery

Service discovery is the problem of finding suitable service(s) that satisfy functional and/or non-functional requirements of a user. Typically, in a Service-Oriented Architecture (SOA), there exists a directory in which service providers can advertise their services in a form that enables potential clients to find and invoke them over the Internet. The notion of Semantic Web services [124] takes us one step closer to interoperability of autonomously developed and deployed services, where a software agent or application can dynamically *find* and *bind* services without having a priori hard-wired knowledge about how to discover and invoke them. OWL-S [4] is a specific OWL [3] ontology designed to provide a framework for semantically describing such services from several perspectives (e.g., discovery, invocation, composition). During the development of a service, the abstract procedural concepts provided by OWL-S ontology can be used along with the domain specific OWL ontologies which provide the terms, concepts, and relationships used to describe various service properties (i.e., Inputs, Outputs, Preconditions, Effects or IOPE's). In general, ontology-based matchmaking is used to discover and invoke service providers against a specific service request [115, 140]. However, the existing techniques either do not consider the ontologies used to describe the services to be semantically heterogeneous or do not provide the support for consideration of both functional and non-functional requirements for service discovery.

To address these limitations, we propose a technique [158] that allows the users to specify context-specific semantic correspondences between multiple ontologies to resolve semantic differences between them. These correspondences are used for selecting services based on the user's functional and non-functional requirements, which are then ranked based on a user-specified criteria. In particular, our technique comprises of two main steps:

- specifying *mappings* between the terms and concepts of the user ontologies and the domain ontologies (which are used to describe the services).
- specifying a *service selection criteria* which uses the mappings to select candidate service providers against a service request query and rank/order them based on user-specified

ranking criteria.

For the sake of simplicity, we assume that inter-ontology mappings and correspondences (Definition 12) can be specified by a domain expert using existing tools such as INDUS [147]. Once the mappings are provided, the objective is to specify a suitable service selection criteria, which in our technique comprises to two aspects: *Selection* of the service providers and then, *Ranking* the selected providers.

7.3.1.1 Service Selection

The first step in service selection is to determine a set of service providers which offer the requested functionality. We call this set as *candidate service providers*.

Definition 15 (Candidate Service Providers) *Let* $\mathbb{S} = \{S_1, \cdots, S_n\}$ *denote the set of services which are available (or registered with our system). We call,* $\mathbb{S}' \subseteq \mathbb{S}$*, the set of candidate providers, if they meet the requested functional properties of the user (in terms of IOPE's).*

In general, some services will match *all* the requested IOPE parameters, while others will not. To distinguish between them, we categorize them based on the *degree of match* [115, 140]: *Exact, Plug-in, Subsumption, Intersection,* and *Disjoint*. Such a categorization also provides an (implicit) ranking amongst the potential providers (e.g., *Exact* match is given the highest rank). Since, the set of services which fall under *Intersection* and *Disjoint* categories do not match the service request (in terms of functional aspects), we ignore them for the rest of the service selection process and only consider the services which belong to *Exact, Plug-in* and *Subsumption* categories.

The second step in the service selection process further refines the set of candidate service providers based on user-specified non-functional attributes, namely Quality of Service (QoS). In unison with [166], we define Quality of Service as a set of non-functional attributes that may impact the service quality offered by a Web service. Because, Web services are distributed as well as autonomous by their very nature, and can be invoked dynamically by third parties over the Internet, their QoS can vary greatly. Thus, it is vital to have an infrastructure

which takes into account the QoS provided by the service provider and the QoS desired by the service requester, and ultimately find the (best possible) match between the two during service discovery.

However, different aspects of QoS might be important in different applications and different classes of web services might use different sets of non-functional attributes to specify their QoS properties. For example, `bits per second` may be an important QoS criterion for a service which provides online streaming multimedia, as opposed to, `security` for a service which provides online banking. As a result, we categorize them into: *domain dependent* and *domain independent* attributes. As an example, Figure 7.4 shows the taxonomy that captures the QoS properties of those restaurant Web services which provide home delivery. The domain-independent attributes represent those QoS characteristics which are not specific to any particular service (or a community of services). Examples include `Scalability, Availability` etc. A detailed list and explanation about such attributes can be found in [166]. On the other hand, the domain-dependent attributes capture those QoS properties which are specific to a particular domain. For example, the attributes `OverallRestaurantRating, PresentationDecor` etc. shown in Figure 7.4 correspond to the restaurant domain. As a result, the overall QoS taxonomy is flexible and enhanceable as different service providers (or communities) can define QoS attributes corresponding to their domain.

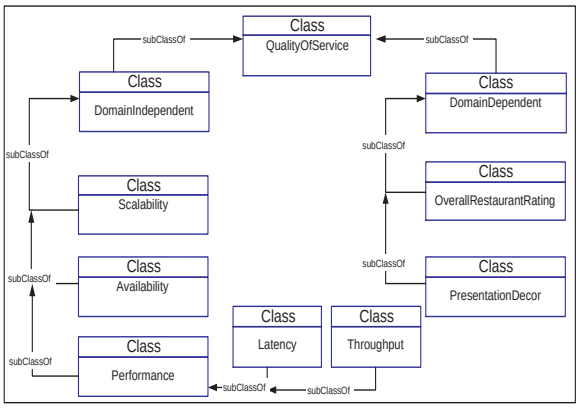


Figure 7.4 Sample QoS Taxonomy

However, in certain cases, a user might consider *some* non-functional attributes valuable for his/her purpose (and hence, defined in the user ontology), instead of *all* the attributes in the QoS taxonomy (Figure 7.4). We use those attributes to compose a *quality vector* comprising of their values for each candidate service. These quality vectors are used to derive a *quality matrix*, $\mathbb{Q}$.

Definition 16 (Quality Matrix) *A quality matrix,* $\mathbb{Q} = \{V(Q_{ij});\ 1 \leq i \leq m; 1 \leq j \leq n\}$, *refers to a collection of quality attribute-values for a set of candidate services, such that, each row of the matrix corresponds to the value of a particular QoS attribute (in which the user is interested) and each column refers to a particular candidate service. In other words,* $V(Q_{ij})$, *represents the value of the* i^{th} *QoS attribute for the* j^{th} *candidate service. These values are obtained from the profile of the candidate service providers and mapped to a scale between 0 & 1 by applying standard mathematical maximization and minimization formulas based on whether the attribute is positive or negative. For example, the values for the attributes* `Latency` *and* `Fault Rate` *needs to be minimized, whereas* `Availability` *needs to be maximized.*

In addition to the above, to give relative importance to the various attributes, the users can specify a *weight value* for each attribute, which are used along with the QoS attribute values to give relative scores to each candidate service using an *additive value function*, f_{QoS}. Formally,

$$f_{QoS}(Service_j) = \sum_{i=1}^{m} (V(Q_{ij}) \times Weight_i) \tag{7.1}$$

where, m is the number of QoS attributes in $\mathbb{Q}$.

For a particular service request query, our technique selects one or more services which satisfies user's constraints (in terms of IOPE's) and has an overall score (for the non-functional attributes) greater than some threshold value specified by the user. If several services satisfy these constraints, then they would be ranked according to the user-specified ranking criteria (Section 7.3.1.2). But, if no service exist, then an exception is raised and the user is notified appropriately. For example, let $\mathbb{S} = \{S_1, S_2, S_3\}$ be the set of candidate service providers which match the requested IOPE's. Assuming, that the user is interested in attributes `Scalability`

and `Availability`, let the quality matrix be:

$$\mathbb{Q} = \begin{pmatrix} & S_1 & S_2 & S_3 \\ \textit{Scalability} & 0.90 & 0.80 & 0.30 \\ \textit{Availability} & 0.90 & 0.45 & 0.20 \end{pmatrix}$$

Further assuming that, the user specifies $Weight_{\texttt{Scalability}} = 0.80$, $Weight_{\texttt{Availability}} = 0.50$, and threshold score value, $U_{Threshold} = 0.50$, only S_1 and S_2 will be selected (after calculation of their respective f_{QoS} scores).

7.3.1.2 Service Ranking

In a real world scenario, given a service request, it is conceivable that there exist scores of service providers, which not only satisfy the functional requirements of the requester, but also the non-functional requirements. As a result, it is of vital importance to let the requesters specify some ranking criteria (as part of the service request query), which would rank the retrieved results (i.e., the list of potential service providers). The traditional approach for ranking the results of matchmaking is completely based on the *degree of match* [115, 140] between the profiles of the service requester and service provider. In our framework also, we use degree of match to categorize (and implicitly order) the set of candidate service providers based on the functional requirements of the user. We further refine each category and select only those candidate service providers which satisfy the non-functional requirements of the user.

Although this is beneficial, we believe the requester should have additional capabilities to specify personalized ranking criteria as part of the service request query. For example, restaurants which may not have the highest quality ratings for food tastiness, but provide speedier home delivery, may be of higher value for a person who is in hurry (and hence wants faster food delivery), compared to a food connoisseur, who will have a preference for tastier food. As a result, the former user would want to rank the candidate service providers based on their promptness of delivery, whereas the later would prefer to have the service providers ranked based on the quality of food they serve.

To achieve this, we introduce the notion of ranking attributes and a ranking function (based on those attributes), which will be used to rank the selected candidate service providers. Once the service providers are ranked, it is left at user's discretion to select the most suitable provider (e.g., the user may do some trade off between the services which meet all the non-functional requirements, but not all the functional requirements exactly).

Definition 17 (Ranking Attributes) *The set of ranking attributes, R_A, comprises of all the concepts (its sub-concepts, properties) in the domain QoS taxonomy which have correspondences (via interoperation constraints) to the concepts in the user ontology, O_U, that capture the non-functional aspects/requirements of the user. For example, if O_U has a QoS concept* `ServicePerformance` *which has a correspondence to the concept* `Performance` *in the domain QoS taxonomy (Figure 7.4), then* {`Performance, Throughput, Latency`} $\in R_A$.

Definition 18 (Ranking Function) *Let $\mathbb{S}$ represent the set of candidate services which match the functional and non-functional requirements of the user, $x \in R_A$ is the ranking attribute, and $R_O \in \{ascending, descending\}$ is the ranking order, then: $f_{Rank}(\mathbb{S}, x, R_O) = \mathbb{S}'$, is called the ranking function, which produces $\mathbb{S}'$, the ordered set of candidate services. For example, let $\mathbb{S} = \{S_1, S_2\}$ be the set of services selected based on the desired QoS properties (from the previous section/example), $x =$* {`Cost`}*, and, $R_O = \{ascending\}$. Assuming,* `Cost` *of S_1 is more than S_2, we have, $f_{Rank}(\mathbb{S}, x, R_O) = \{S_2, S_1\} = \mathbb{S}'$.*

7.3.2 Ontology-based Service Composition

We have introduced and addressed the problem of service composition in Chapter 4 of this thesis. However, our techniques were based on the assumption that all the component services considered for analysis had uniform semantics—an assumption that does not hold in the real world. To overcome this restriction, we develop ontology-extended components and introduce semantically consistent methods for assembling such components into a feasible composition.

Our work, in particular, leverages the existing research on graph-based workflow languages (GBWL) [191], which allows to model various aspects of traditional workflows. A GBWL specification of a workflow, known as *workflow schema* (WFS), describes the components of

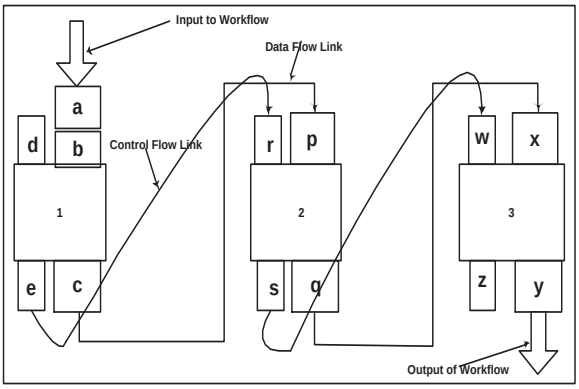


Figure 7.5 Workflow Schema Graph

the workflow and the characteristics of the environment in which the workflow will be executed. The workflow schemas are connected to yield directed graphs of workflow schemas, called *workflow schema graphs* (WSG). The nodes of a WSG correspond to the workflow components and edges specify the constraints between the components. Figure 7.5 shows a WSG consisting of three components. Note that each workflow component trivially has a WSG description.

When a workflow is to be executed, a WFS is instantiated resulting in the creation of a *workflow instance* (WFI). Each WFI created from a well-formed WFS is guaranteed to conform to the conditions specified by the WFS. The *functional aspect* of a workflow schema specifies the task to be performed by the corresponding workflow instances. The *information aspect* of a WSG specifies the data flow between the individual components. Associated with each component is a set of typed inputs and outputs. At the initiation of a workflow, the inputs are read, while on termination the results of the workflow are written to the outputs. The data flow which is defined in terms of the inputs and outputs, models the transfer of information through the workflow. For example, in Figure 7.5, component 1 has inputs a and b and an output c, and component 2 has an input p and an output q. Note that the data flow between components 1 & 2 is represented by the data flow link (c, p). The *behavioral aspect* of a WFS specifies the conditions under which an instance of the component will be executed. The behavior of a workflow is determined by two types of conditions: *Control* conditions and *Instantiation* conditions. The relation between the components is determined by the control

conditions, which are expressed by the control flow links. These control flow links specify the execution constraints. For example, Figure 7.5 shows control flow links (e, r) specifying that the execution of component 1 has to precede the execution of component 2. In order for a workflow component to be executed, its instantiation conditions have to be set to True. Specifically, the existence of a control flow link from 1 to 2 does not imply that 2 will necessarily be executed as soon as 1 is executed (unless the instantiation conditions are satisfied). Note that in general, it is possible to have cyclic data and control flow links.

From the preceding discussion it follows that modeling workflows is akin to developing composite services where appropriate data and control flow links are appropriately generated (either automatically or manually). Furthermore, a workflow or a composite service can be regarded as a "black box" and encapsulated as a component in a more complex composition model. Thus, to develop a technique for ontology-based service composition, it suffices to show how components can be extended with ontologies and how the resulting ontology-extended components can be composed to yield more complex component services (or equivalently, workflows).

Recall that a component has associated with it, input, output and control flow attributes. The control flow attributes take values from the domain $D(CtrlType) = \{\texttt{true}, \texttt{false}, \phi\}$, where ϕ corresponds to the initial value of a control flow attribute indicating that the control flow link is yet to be signaled.

Definition 19 (Ontology-Extended Component) *An ontology-extended workflow component, s, consists of (see Figure 7.6):*

- *An associated ontology O_s.*
- *A set of data types τ_1, τ_2,..., τ_n, such that $\tau_i \in O_s$, for $1 \leq i \leq n$.*
- *A set of input attributes $input_s$ represented as an r-tuple (A_{1_s}:τ_{i_1},...,A_{r_s}:τ_{i_r}) (e.g., `Temp:C` is an input attribute of type `Centigrade`).*
- *A set of output attributes $output_s$ represented as a p-tuple (B_{1_s}:τ_{j_1},...,B_{p_s}:τ_{j_p}) (e.g., `Day:DayType` is an output attribute of type `DayType` whose enumerated domain is {`Hot`,*

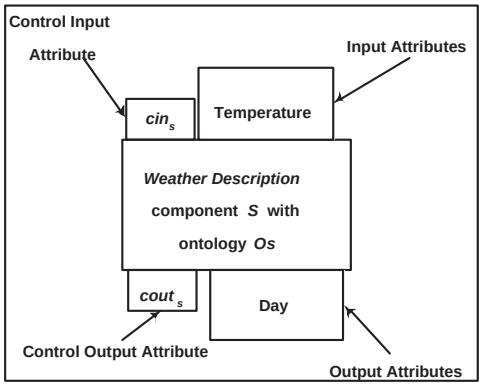


Figure 7.6 Ontology-Extended Workflow Component

`Warm, Cold`}*).*

- *A control input attribute, cin_s, such that $\tau(cin_s) \in CtrlType$. A true value for cin_s indicates that the component s is ready to start its execution.*

- *A control output attribute, $cout_s$, such that $\tau(cout_s) \in CtrlType$. A true value for $cout_s$ indicates the termination of the execution of component s.*

The composition of two components specifies the data flow and the control flow links between the two components. In order for the meaningful composition of ontology-extended components to be possible, it is necessary to resolve the semantic and syntactic mismatches between such components.

Definition 20 (Ontology-Extended Component Composition) *Two components s (source) (with an associated ontology O_s) and t (target) (with an associated ontology O_t) are* composable *if some (or all) outputs of s are used as inputs for t. This requires that there exists:*

- *A directed edge, called control flow link, $C_{link}(s,t)$, that connects the source component s to the target component t. This link determines the flow of execution between the components. We have:*

$$C_{link}(s,t) \in cout_s \times cin_t,$$

which means that there exists $x \in cout_s$ *and* $y \in cin_t$ *such that* $\tau(x) \in CtrlType$ *and* $\tau(y) \in CtrlType$. *For example, in Figure 7.5, (e, r) is a control flow link between the components 1 and 2.*

- *A set of data flow links,* $D_{link}(s,t)$ *from the source component* s *to the target component* t. *These links determine the flow of information between the components. We have:*

$$D_{link}(s,t) \subseteq output_s \times input_t,$$

which means that there exist attributes $x \in output_s$ *and* $y \in input_t$, *such that* $\tau(x) = \tau_i \in O_s$ *and* $\tau(y) = \tau_j \in O_t$. *For example, in Figure 7.5, (c, p) is a data flow link between the components 1 and 2.*

- *A set of (user defined) interoperation constraints,* $IC(s,t)$, *that define a mappings set* $MS(s,t)$ *between outputs of* s *in the context of the ontology* O_s *and inputs of* t *in the context of the ontology* O_t. *Thus, if* $x : O_s = y : O_t$ *is an interoperation constraint, then* x *will be mapped to* y, *and we write* $x \mapsto y$.

- *A set of (user defined) conversion functions* $CF(s,t)$, *where any element in* $CF(s,t)$ *corresponds to one and only one mapping* $x \mapsto y \in IC(s,t)$. *The identity conversion functions may not be explicitly specified. Thus,* $|IC(s,t)| \leq |CF(s,t)|$.

Note that, in general, a component may be connected to more than one *source* and/or *target* component(s). The mappings set $MS(s,t)$ and the conversion functions $CF(s,t)$ together specify a *mapping component*, which performs the mappings from elements in O_s to elements in O_t.

Definition 21 (Mapping Component) *A mapping component,* $MAP(s,t)$, *which maps the output and the control output attributes of the source* s *to the input and the control input attributes of the target* t *respectively, consists of:*

- *Two ontologies,* O_s *and* O_t, *where* O_s *is associated with the inputs of* $MAP(s,t)$, *and* O_t *is associated with its outputs.*

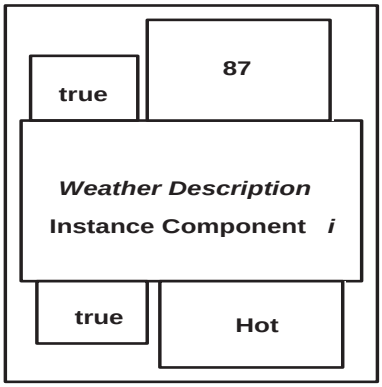


Figure 7.7 Ontology-Extended Component Instance

- *A set of mappings* $MS(s,t)$ *and their corresponding conversion functions* $CF(s,t)$ *that perform the actual mappings and conversions between inputs and outputs.*

- *A set of data inputs* $input_{map}$=$(A_{1_M} : \tau_{s_1}, \cdots, A_{p_M} : \tau_{s_p})$, *which correspond to the output attributes of component s, that is,* $input_{map} \equiv output_s$. *Also,* τ_{s_1},..., τ_{s_p} *is a set of data types such that* $\tau_{s_i} \in O_s$, $\forall\ 1 \leq i \leq p$.

- *A set of data outputs* $output_{map}$=$(B_{1_M} : \tau_{t_1}$,...,$B_{r_M} : \tau_{t_r})$, *which correspond to the input attributes of component t, that is,* $output_{map} \equiv input_t$. *Also,* τ_{t_1},...,τ_{t_r} *is a set of data types such that* $\tau_{t_i} \in O_t$, $\forall\ 1 \leq i \leq r$.

- *A control input* cin_{map}, *which corresponds to the control output attribute,* $cout_s$ *of component s. Also,* $\tau(cin_{map}) = CtrlType$.

- *A control output* $cout_{map}$, *which corresponds to the control input attribute,* cin_t *of component t. Also,* $\tau(cout_{map}) = CtrlType$.

Ontology-extended component instances (see Figure 7.7) are obtained by instantiating the ontology-extended components at execution time. This entails assigning values to each of the component attributes. These values need to be of the type specified in the component schema. If a component instance *ins* is based on a component schema *sch* of the component s, we say that $hasSchema(ins) = sch$. We also say that for a given attribute, p, $v(p) \in D(t)$ refers to its value, if $\tau(p) = t \in O_s$.

Definition 22 (Ontology-Extended Component Instance) *The instance corresponding to an ontology-extended workflow component s has to satisfy the following constraints:*

- *For every input attribute* $x \in input_s$, $v(x) \in D(t)$, *if* $\tau(x) = t \in O_s$ *(e.g.,* `Temperature` $= 87$*).*
- *For every output attribute* $y \in output_s$, $v(y) \in D(t)$, *if* $\tau(y) = t \in O_s$ *(e.g.,* `DayType` $=$ `Hot`*).*
- *For the control input attribute,* $cin_s \in \{true, false, \phi\}$*, a true value indicates that the component s is ready for execution.*
- *For the control output attribute,* $cout_s \in \{$`true`, `false`, $\phi\}$*, a* `true` *value indicates that the component s has finished its execution.*
- *For an instantiation condition,* $insc_s \in \{$`true`, `false`$\}$*. If the evaluation of this condition returns* `true`*, then the execution of the component begins. This condition is defined as:*

$$insc_s \equiv \{(cin_s)\ \Lambda\ (\forall\ x \in input_s, \exists\ v(x)\ )\},$$

such that $\tau(x) = t$ *and* $t \in O_s$*.*

Semantic Consistency of composition of ontology extended components is necessary to ensure the soundness of the composition. In particular, the composition of any two ontology-extended components s (source) and t (target) is said to be *consistent*, if the following conditions are satisfied:

- The *data & control flow* between s and t must be consistent, i.e., control flow should follow data flow.
- The data and control flow links must be *syntactically consistent* i.e., there should be no syntactic mismatches for data flow links.
- The data and control flow links must be *semantically consistent*, i.e., there should be no unresolved semantic mismatches along the data and control flow links. Note that the semantic mismatches between the components are resolved by the mapping components.

- Data and control flow links should be *acyclic*.[1]

Thus, an ontology-extended workflow (or a composite service) W is semantically consistent if the composition of each and every pair of *source* and *target* components is consistent.

7.4 Discussion

The work proposed in this chapter provides an approach for flexible discovery and composition of semantically heterogeneous Web services. We lay stress on the fact that, since different users may use different ontologies to specify the desired functionalities and capabilities of a service, the ability to specify inter-ontology mappings during service discovery and composition is needed. Such mappings enable terms and concepts in the service requester's ontologies to be brought in correspondence with that of the service provider's ontologies. In particular, to address the ontology-based service discovery problem, we propose a taxonomy for the non-functional attributes, namely QoS, which provides a better model for capturing various domain-dependent and domain-independent QoS attributes of the services. These attributes allow the users to dynamically select services based on their non-functional aspects. We also introduced the notion of personalized ranking criteria, which is specified as part of the service request, for ranking the (discovered) candidate service providers (e.g., ranking service providers from high to low based on their `Availability`). Such a criteria 'enhances' the traditional ranking approach, which is primarily based on the degree of match [115, 140]. On the other hand, to address the ontology-based composition problem, we introduced the notion of an ontology-extended component, and illustrated how such components can be composed into syntactically and semantically consistent compositions (or workflows). A key idea of our technique is to specify consistent data and control flow between the components that take part in a composition.

In our setting, we assumed that a domain expert is responsible for specifying specifying the inter-ontology mappings and correspondences between the terms and concepts. However, in practice, this is a cumbersome and error-prone process. Towards this end, it is of interest to

[1] Our framework at present cannot handle cyclic data and control flow links.

develop techniques that will enable to model the mappings semi-automatically. Additionally, the ability to verify the correctness of the semantic correspondences using formalisms such as Distributed Description Logics [46, 84] is required. Furthermore, in our approach to service discovery, we only consider non-functional properties in terms of quality of service (QoS). In reality, services used and provided by individual organizations are based on rigorous service-level agreements (SLAs). Consequently, consideration of both QoS properties and SLAs in the context of ontology-based service discovery is of importance. Finally, another topic that needs attention is the ability to analyze and verify the dynamical and behavioral aspects of workflow/composite service execution along with the ability to enable semantic mediation during execution.

CHAPTER 8. SYSTEM ARCHITECTURE AND EVALUATION

This chapter describes the MoSCoE (Modeling Web Service Composition and Execution) system. The chapter is divided into three sections. The first and second sections describe the MoSCoE architecture and implementation details, respectively, whereas empirical evaluation is shown in the third section. An open-source implementation of the system is available at http://www.moscoe.org.

8.1 MoSCoE Architecture

This thesis proposes a new framework MoSCoE (Modeling Web Service Composition and Execution) for (semi-)automatically realizing new services from a pre-existing set of component services. The salient features of the proposed framework are (a) it is interactive (unlike traditional single-step request-response approaches) which leads to efficient and incremental service development, (b) it uses automata-theoretic approach to generate provably correct construction of composite service and (c) the modules of MoSCoE are well-defined and clearly partitioned to allow developers to evaluate and test newer modules by replacing one module by another with the same functionality.

Figure 8.1 shows the architectural diagram of MoSCoE, which comprises of two main modules: a *composition management* module responsible for statically identifying a composition of existing services that can realize the desired goals; and an *execution management* module that deploys the composite services identified statically. In the current implementation, the system accepts from the user a high-level (and possibly incomplete) specification of the goal service in the form of a labeled transition system (LTS, Definition 3). Additionally, the component services that are also represented using LTSs. Note that in practice, the service providers pub-

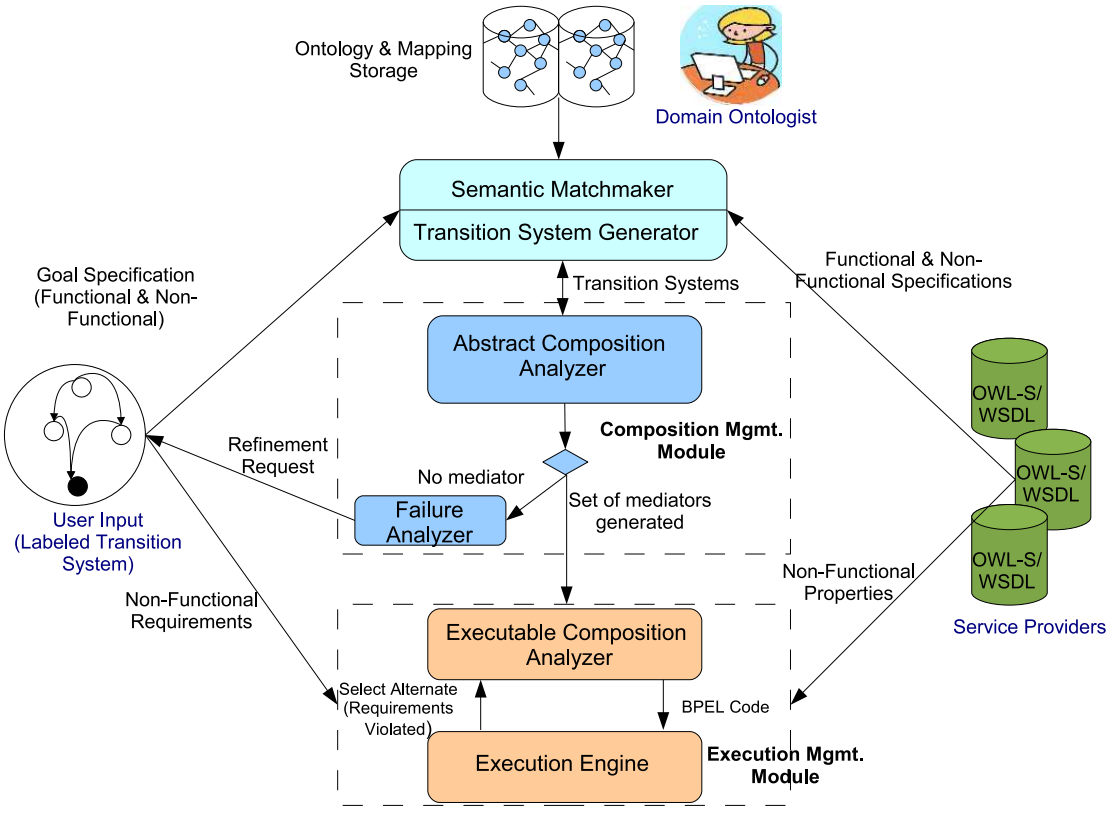


Figure 8.1 MoSCoE Architectural Diagram

lish their services with widely used specification languages such as WSDL [62] and BPEL [16]. Consequently, we have developed a number of translators to convert the WSDL/BPEL descriptions of existing services into LTS. MoSCoE manipulates these input data (user-provided service specification and published component service descriptions) and automatically identifies a composition that realizes the goal service. However, in the event that a composition cannot be realized, the system identifies the cause(s) for the failure of composition and provides that information to the developer for appropriate reformulation of the goal specification. We describe both the modules in the following paragraphs.

Composition Management Module: Given the LTS representations of a set of N component services $\{LTS_1, LTS_2, \ldots, LTS_N\}$ and a desired goal LTS_G, service composition in MoSCoE amounts to identifying a subset of component services, which when composed with a mediator

(to be generated) LTS_M, realize the goal service LTS_G. The role of the mediator is to replicate input/output actions of the user as specified by the goal and to act as a message-passing interface between the components and between the component(s) and the client. It is not capable of providing any functionality (e.g., credit card processing) on its own; these are provided only by the component services. Algorithm 1 presents a technique for generating such a mediator and essentially identifies whether LTS_M realizes LTS_G using the notion of simulation and bisimulation equivalence. Informally, simulation equivalence ensures that every behavioral pattern in the goal is present in the composed mediator, whereas bisimulation equivalence is a symmetric relation which ensures that the composition offers exactly the same behavior as specified in the goal, and nothing more.

However, algorithm 1 suffers from the state-space explosion problem since the number of ways the component services can be composed is exponential to the number of component service states. This becomes a challenge with the increasing size of the search space of available component services. Hence, to address this limitation, we consider non-functional aspects (e.g., Quality of Service) in Algorithm 2 to winnow components (thereby reducing the search space) and compositions that are functionally equivalent to the goal, but violate the non-functional requirements desired by the user. The non-functional requirements are quantified using *thresholds*, where a composition is said to conform to a non-functional requirement if it is below or above the corresponding threshold, as the case may be. For example, for a non-functional requirement involving the `cost` of a service composition, the threshold may provide an *upper-bound* (maximum allowable `cost`) while for requirements involving `reliability`, the threshold usually describes a *lower-bound* (minimum tolerable `reliability`). If more than one "feasible composition" meets the goal specification (both functional and non-functional requirements), our algorithm generates all such compositions and ranks them (Algorithm 3). It is then left to the user's discretion to select the best composition according to the requirements.[1]

In the event that a composition as outlined above cannot be realized using the available component services, the composition management module provides feedback to the user re-

[1]This feature of our tool is undergoing implementation at the time of writing this dissertation.

garding the cause(s) of the failure (see Section 4.3.3). The feedback may contain information about the function names and/or pre-/post-conditions required by the desired service that are not supplied by any of the component services. Such information can help to identify specific states in the state machine description of the goal service. In essence, the module identifies all un-matched transitions along with the corresponding goal STS states. Additionally, the failure of composition could be also due to non-compliance of non-functional requirements specified by the user. When such a situation arises, the system identifies those requirements that cannot be satisfied using the available components, and provides this information to the service developer for appropriate reformulation of the goal specification. This process can be iterated until a realizable composition is obtained or the developer decides to abort.

Execution Management Module: The result from the composition management module is a set of feasible compositions each defining a mediator that will enable interaction between the client and the component services. The execution management module considers non-functional requirements (e.g., `performance, cost`) of the goal (provided by the user) and analyzes each feasible composition. It selects a composition that meets all the non-functional requirements of the goal, generates executable BPEL code, and invokes the MoSCoE execution engine. This engine is also responsible for monitoring the execution and recording violation of any requirement of the goal service at runtime. In the event a violation occurs, the engine tries to select an alternate feasible composition. Furthermore, during execution, the engine leverages a pre-defined set of inter-ontology mappings to carry out various data and control flow transformations 7.[2]

8.2 Implementation

The MoSCoE tool has been implemented entirely in Java to ensure its portability in multiple environments. In particular, there are two basic segments, namely the front-end and back-end, of the implementation that resonate the architecture (Figure 8.1).

[2]This feature/module of our tool is also undergoing implementation at the time of writing this dissertation.

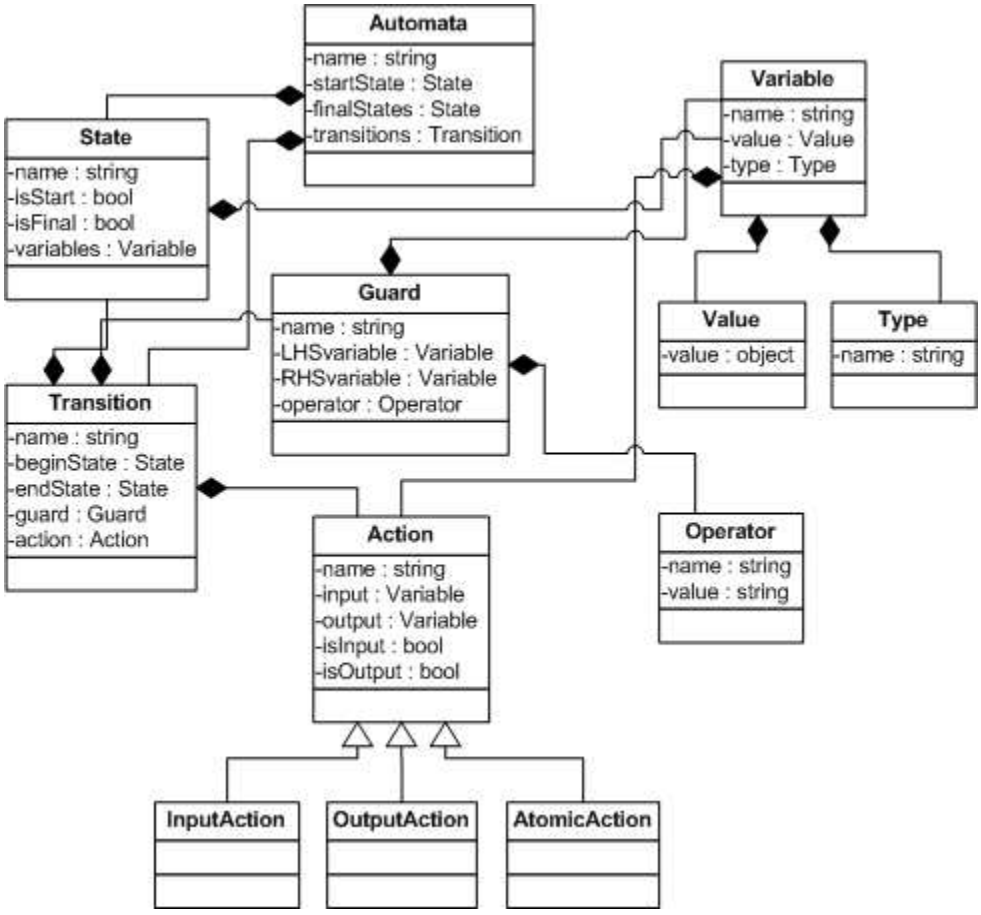


Figure 8.2 UML Representation of a Labeled Transition System

8.2.1 Back-End Implementation

The back-end implementation carries out translation of WSDL/BPEL specifications into corresponding labeled transition systems as well as identifies a feasible composition (i.e., a mediator) that realizes the LTS goal specification according to Algorithm 1. A vital aspect of the back-end implementation is the representation of a labeled transition system in the object-oriented programming paradigm (in our case, Java). Figure 8.2 shows the UML representation of the labeled transition system. The class *Automata* is the root class where the attributes `name`, `startState`, `finalStates` and `transitions` correspond to the name, the start state, the set of final states, and the set of labeled transitions, respectively, of the LTS. Each automaton (or an LTS) comprises of multiple states and transitions, that are represented by the *State* and

Transition classes, respectively. A *State* comprises of multiple attributes namely the name of the state (`name`), the variables associated with the state (`variables`), and an indicator for whether the state is a start state (`isStart`) or a final state (`isFinal`). A *Transition* also comprises of similar attributes: the name of the transition (`name`), the state from which the transition originates (`beginState`), and the state in which the transition ends (`endState`). In addition, each *Transition* is also annotated with a *Guard* and an *Action*. Each *Guard* is represented by its name (`name`), and are essentially a conjunction/disjunction of boolean predicates. In the current implementation, the allowed predicate *Operators* are: less-than (<), greater-than (>), equality (==), and inequality (!=). The *Actions* comprise of input and output variables, represented by `input` and `output`, respectively. It is to be noted that for any given action, either the `input` or `output` variables can be assigned null, but not both simultaneously (i.e., "void" actions are not allowed). The actions are further categorized into `InputActions`, `OutputActions` and `AtomicActions` indicating a situation in which the service receives a message from the environment (`isInput`), sends a message to an environment (`isOutput`), and provides a function that can be invoked, respectively.

The WSDL/BPEL translator of MoSCoE takes as input valid WSDL/BPEL files and instantiates an *Automata* object. To carry out this translation, the translator maps various WSDL/BPEL constructs to the *Automata* representation of the LTS. For example, Appendices A and B show the BPEL and WSDL descriptions of the `e-Auction` service, respectively, and Figure 3.5 shows its corresponding LTS representation as generated by the translator. Interested readers can refer to Section 3.2.2 for details on the translation process. Note that the current implementation of the translator cannot handle complex BPEL constructs such as message correlation, fault handling, and compensation. Furthermore, the implementation of the composition algorithm also generates the mediator LTS which is represented as an *Automata* object. In order to use the generated mediator, the corresponding *Automata* object is re-translated into an executable BPEL which, with human assistance (e.g., modifying the BPEL/WSDL files), can be executed in a BPEL engine (e.g., ActiveBPEL).

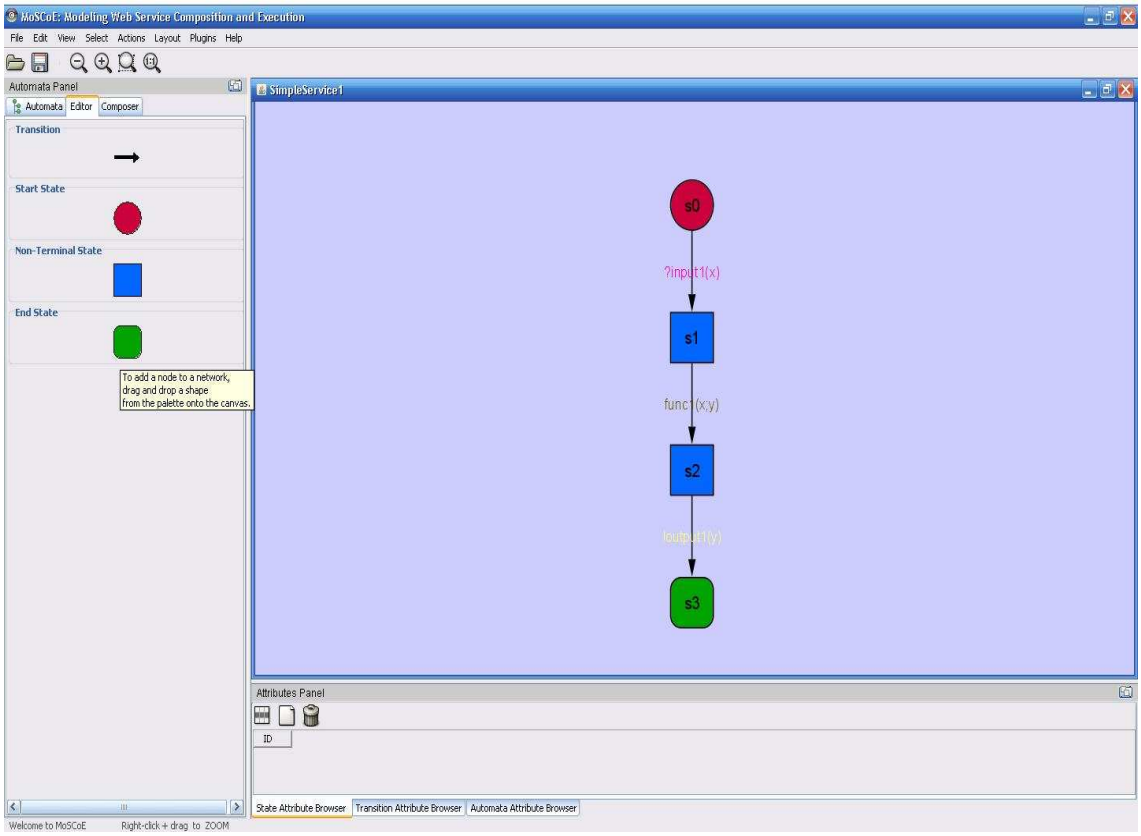


Figure 8.3 Labeled Transition System Editor-1

8.2.2 Front-End Implementation

The front-end implementation consists of multiple aspects. In particular, the MoSCoE graphical interface can be utilized by different types of users. They include:

- A *service provider* who is responsible for providing descriptions of various services that it publishes which can be used for realizing a feasible composition.
- A *service developer* who is responsible for modeling complex goal services and generating a mediator (that realizes the goal service) which can be deployed for execution.
- A *service client* who is simply responsible invoking the deployed composite service (i.e., the mediator).

Note that in many cases, the same entity can assume multiple roles and perform the appropriate actions. Figure 8.3 shows a screenshot of the MoSCoE graphical interface which leverages

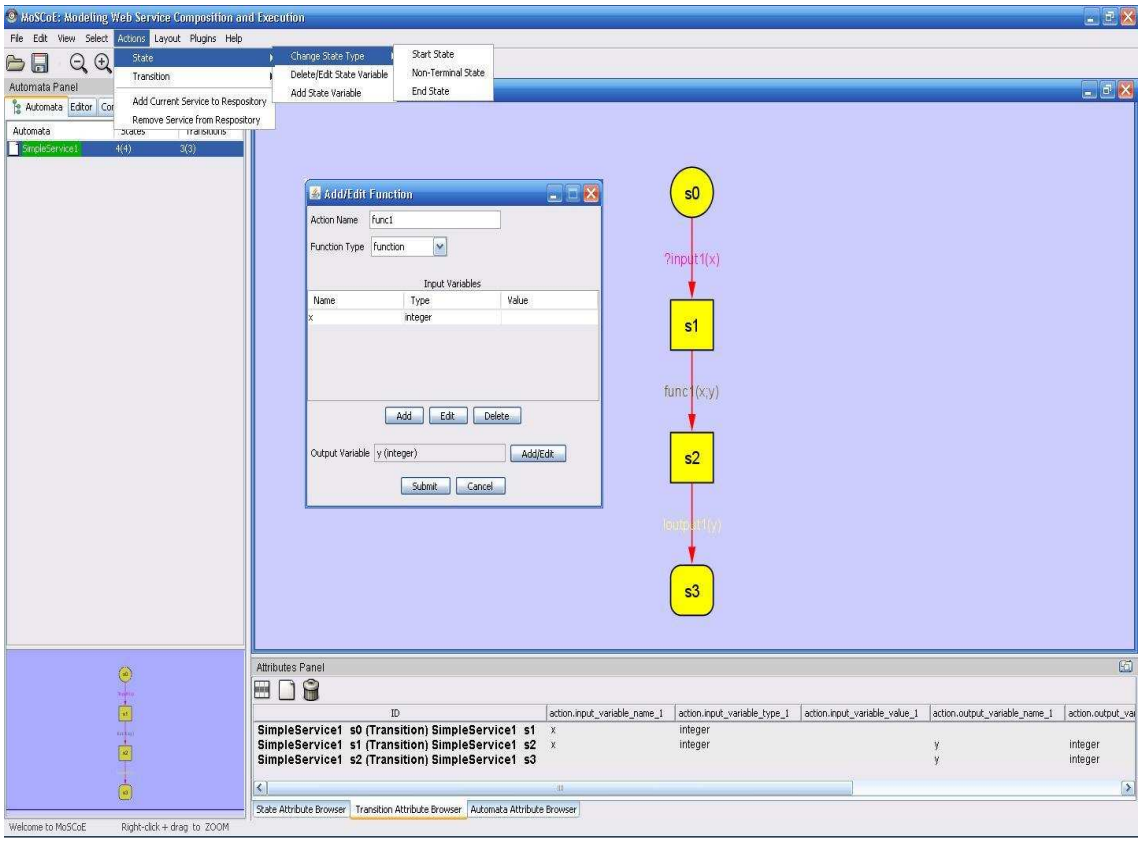


Figure 8.4 Labeled Transition System Editor-2

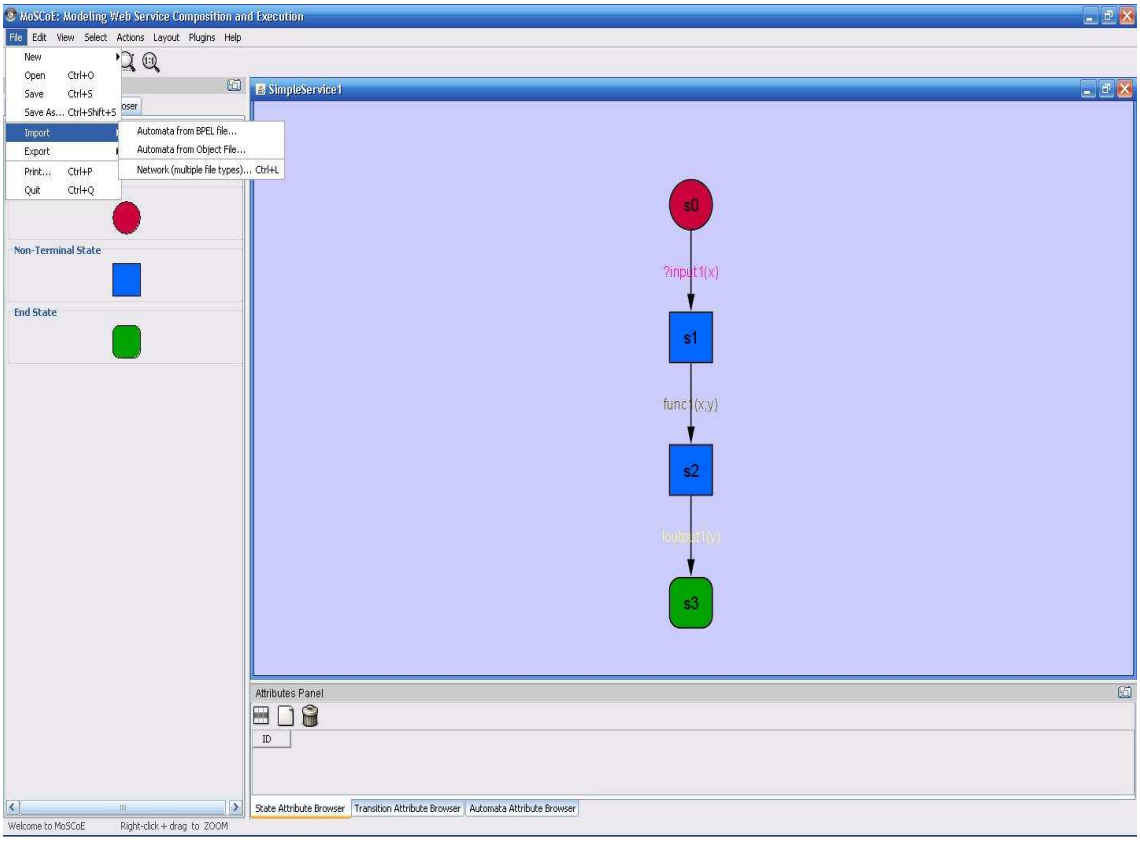


Figure 8.5 Importing Labeled Transition Systems

a widely used open-source platform for visualization called Cytoscape (http://www.cytoscape.org/). In particular, Figure 8.3 shows the editor for modeling services using LTS-based representations and we envision that a service developer would use this editor for modeling the goal services. The states and transitions of the LTS can be modeled by dragging the appropriate construct from the left pane. Each of these constructs can be highlighted and edited to add more information. For instance, Figure 8.4 shows how the transition from state s_1 to s_2 is edited by essentially adding an atomic function `func1` which has an input variable `x` of type *integer* and an output variable `y` also of type *integer*. These aspects of the LTS are displayed in the bottom pane of the graphical interface (see "Attributes Panel" in Figure 8.4).

However, instead of manually modeling the LTS representation of a service, a user of the system also has the option of importing appropriate BPEL and WSDL files or an object-oriented (in this case Java) representation of the Automata model (Figure 8.2) as shown in

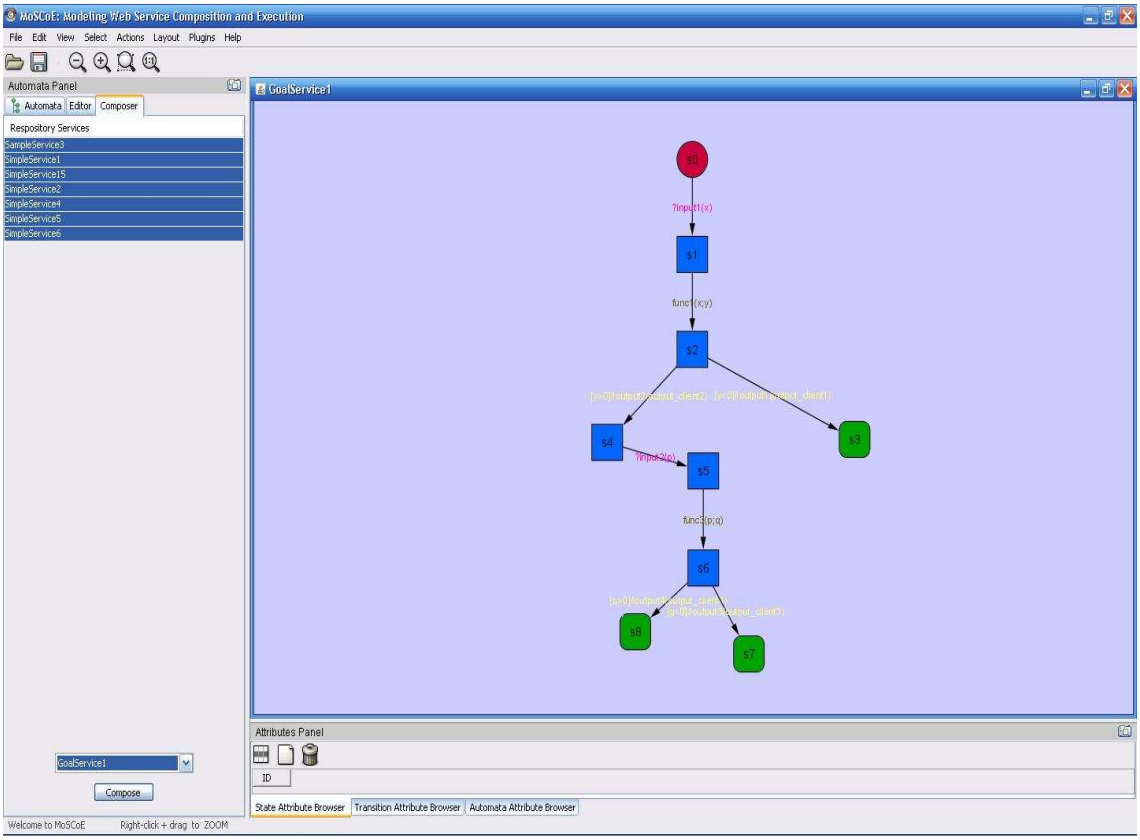


Figure 8.6 Service Composition and Repository

Figure 8.5. This feature can also be used by the service providers to publish their services within the MoSCoE repository which can then be used for analysis during the composition process. Figure 8.6 shows both the service repository and composition views of the tool. In particular, during the composition process, the service developer can select either all or a subset of available component services in the repository (for analysis) along with a suitable goal service (that he/she wants to model) and invoke the composition algorithm (Algorithm 1). This results into either a successful generation of a mediator (indicating that the goal service has been realized using the available component services) or a failure of composition. In the event of a failure, the system highlights the states and transitions along with the guards and actions (if any) that cannot be realized by the component services. For example, Figure 8.7 shows that the function `func12(x;y)` in the transition from state s_1 to s_2 cannot be realized by any of the existing component services, thereby resulting in the failure of composition. Notice

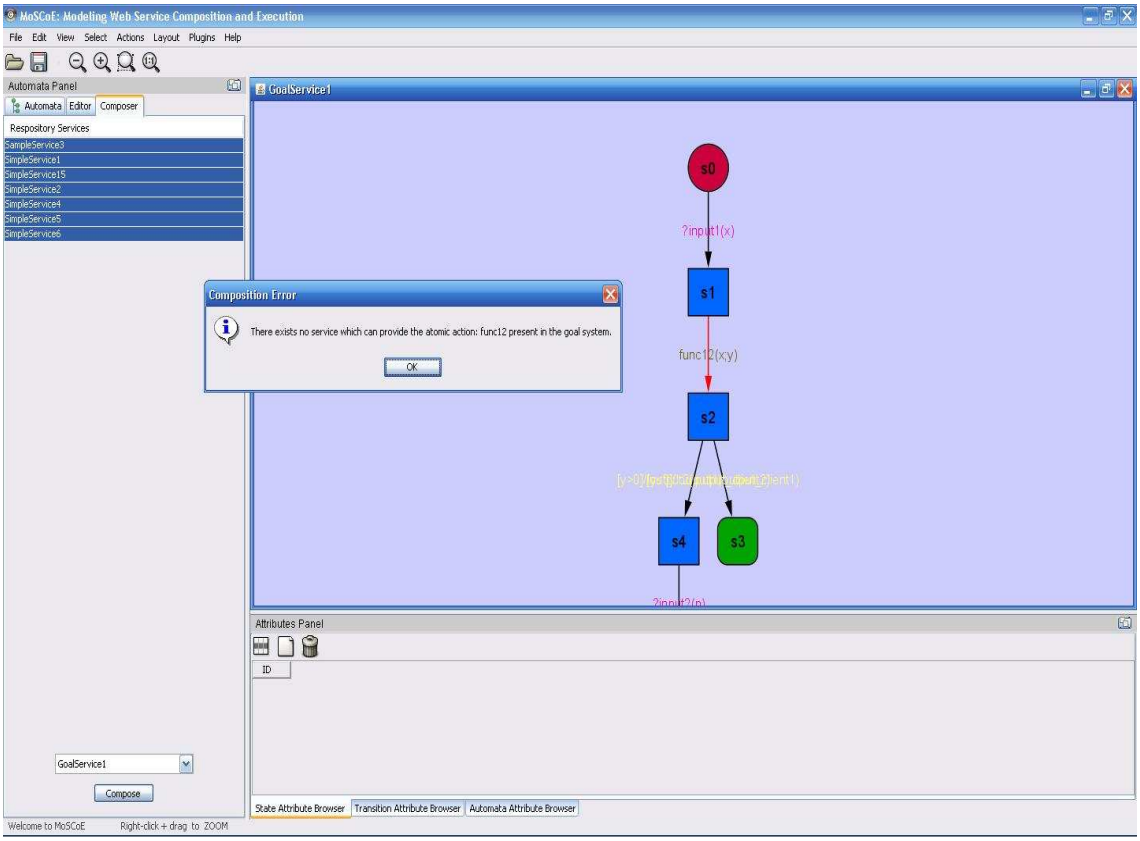


Figure 8.7 Service Composition Error

that the the (unrealizable) transition from s_1 to s_2 is highlighted in red.

8.3 Empirical Evaluation

8.3.1 `Health4U` Case Study

To illustrate the salient aspects of MoSCoE as outlined above, we refer back to the `Health4U` example introduced in Section 4.2 and modify it. In particular, we assume that the service developer is assigned to assemble two different composite services, namely `Health4U`' and `Health4U`", where the former allows patients to search for an appropriate physician based on the ailment that is to be treated and make an appointment, whereas the latter allows patients to search for physicians as well as pediatricians depending not only on the ailment to be treated, but also on the age of the patient. Specifically, if the patient is less than 15 years old,

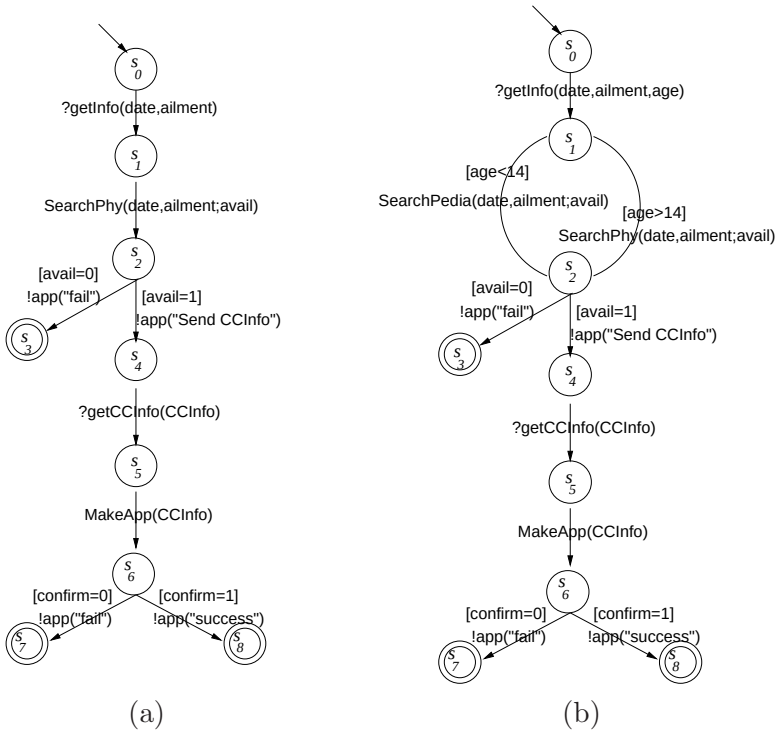


Figure 8.8 LTS representation of (a) `Health4U`' (b) `Health4U`"

`Health4U`" allows to search for (specialist) pediatricians instead of (regular) physicians. The corresponding LTS representation of `Health4U`' and `Health4U`" are shown in Figures 8.8(a) and 8.8(b), respectively. As shown in Section 8.2.2, such a transition system can be either modeled in MoSCoE by the service developer either using the intuitive graphical LTS editor or providing the appropriate BPEL and WSDL and invoking the LTS translator. Figure 8.9, on the other hand, show the available component services that the system can use to realize `Health4U`' and `Health4U`". We assume that such services will be published by the service providers in the MoSCoE repository by providing appropriate BPEL and WSDL descriptions.

The service composition process is initiated by selecting the goal service along with the set of component services that can be analyzed by the composition algorithm (Algorithm 1) to realize the goal service. For our scenario, we begin by selecting `Health4U`' and all the available component services (Figure 8.9). Once the composition starts, an appropriate mediator is determined if it exists. Figure 8.10 shows the corresponding mediator generated

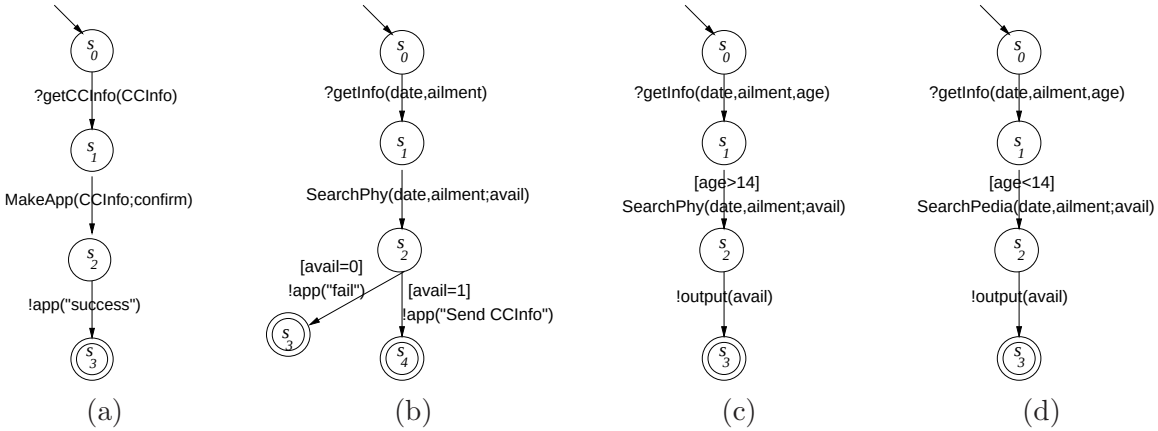


Figure 8.9 LTS representation of (a) `MakeApp` (b) `SearchPhy` (c) `SearchPhy`' (d) `SearchPedia` component services

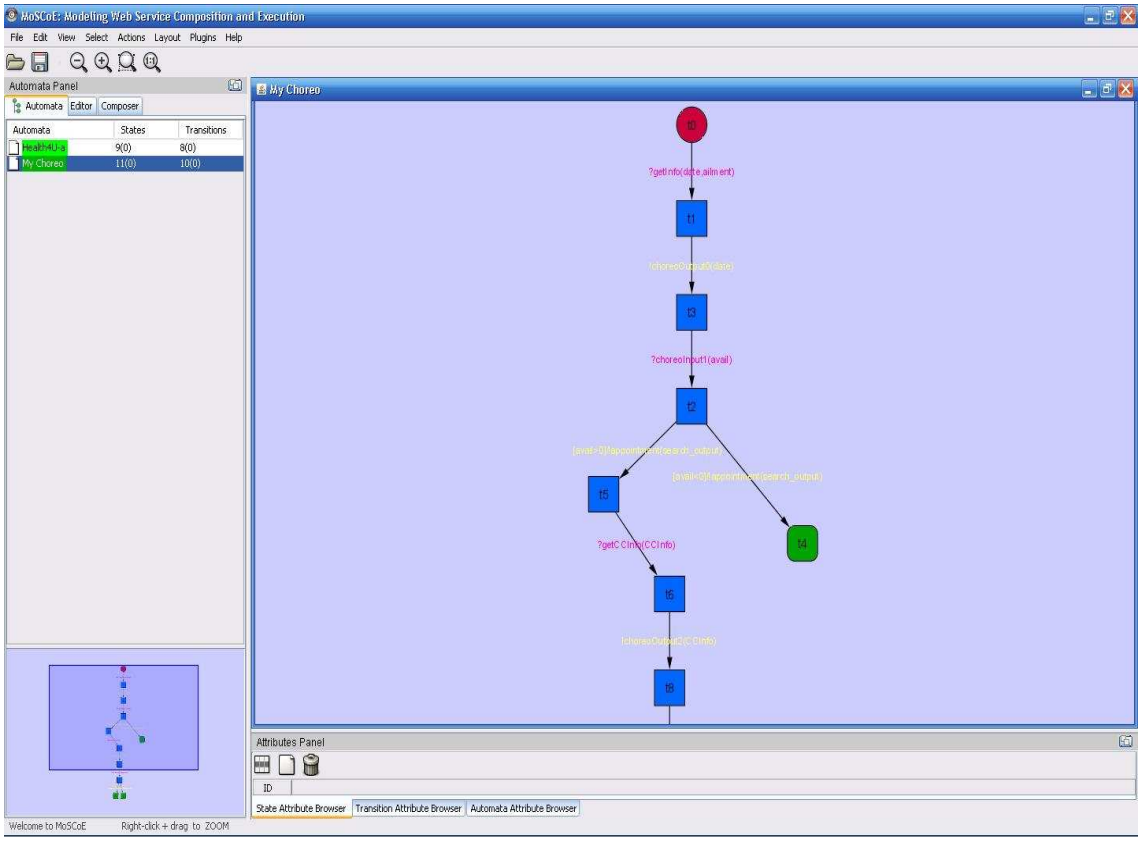


Figure 8.10 LTS representation of `Health4U`' mediator

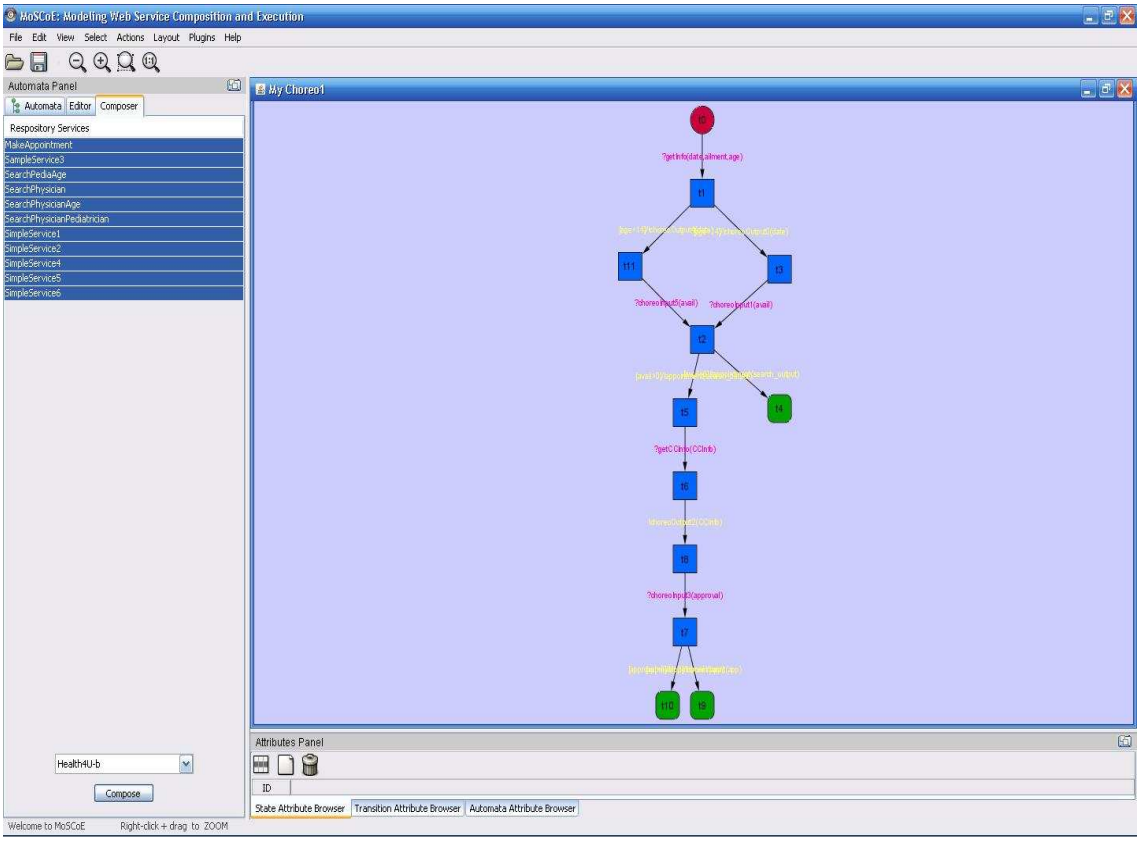


Figure 8.11 LTS representation of `Health4U`" mediator

to realize `Health4U`' using the services shown in Figures 8.9(a) and 8.9(b), whereas Figure 8.11 shows the corresponding mediator generated for `Health4U`" using the services shown in Figures 8.9(a), 8.9(c) and 8.9(d). Note that even though it might be possible that there exists multiple mediators which can realize a given goal service, the current implementation generates only one—work-in-progress is aimed at enhancing the system to generate all the possible alternatives.

However, during this process of composition, it is possible that the goal service cannot be realized from the available component services, thereby resulting in the failure of composition. As explained in Section 4.3.3, an unique aspect of MoSCoE is the ability to identify such failures and present them to the service developer for appropriate reformulation. To illustrate this feature, let us assume that the service developer modifies the `Health4U`' by modifying the atomic action `SearchPhy` (example, changing its input arguments and narrowing the search to

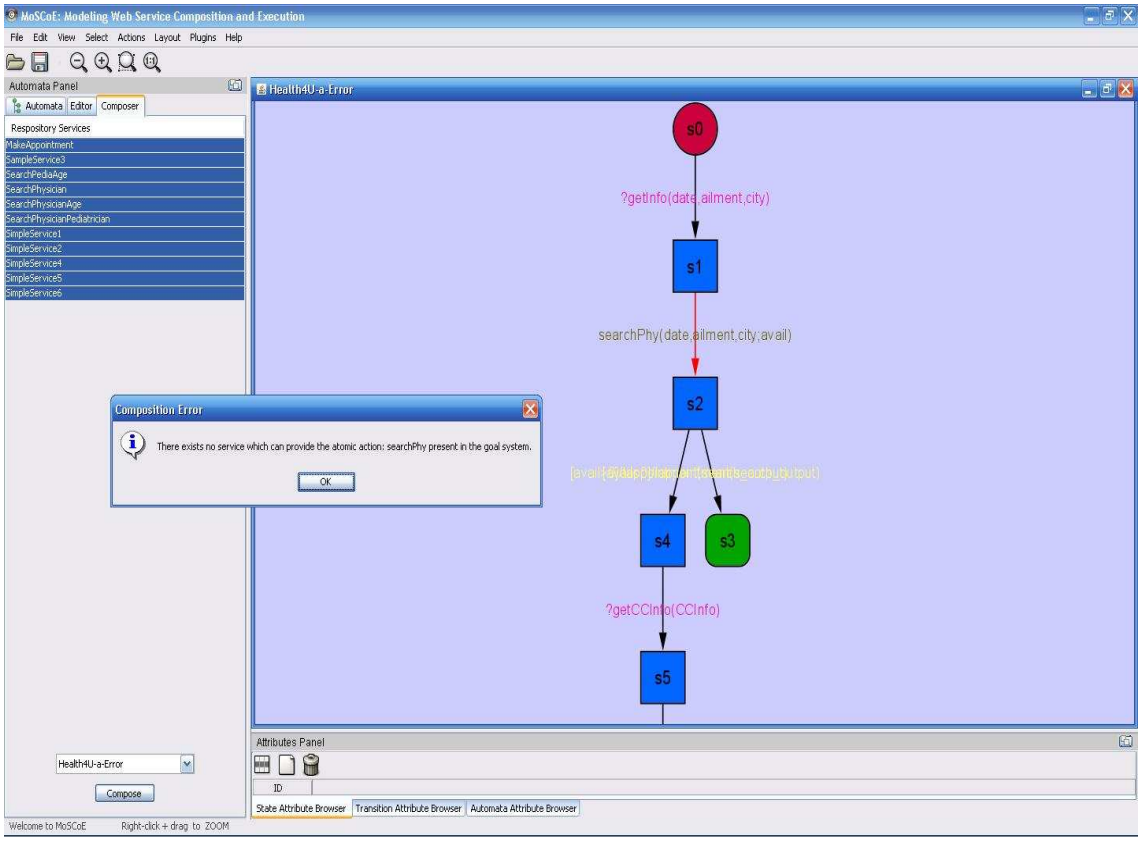


Figure 8.12 Composition Failure for `Health4U`' mediator

only specific cities) and tries to determine a feasible composition using the component services shown in Figures 8.9. As shown in Figure 8.12, this results into a failure of composition because the required atomic action cannot be provided by any of the available services. Consequently, the "failed-transition" is highlighted in the goal LTS, which the service developer can modify/reformulate to achieve a feasible composition. Note that such a process of reformulation can be repeated multiple times until a mediator is generated or the developer decides to abort. Furthermore, in the current implementation we present all the failure causes to the developer for analysis. However, in practice, it is possible that only few of the errors are vital, fixing which could potentially get rid of the remaining errors. Towards this end, we plan to investigate and implement identification of root failure-causes based on techniques such as root-cause analysis [173].

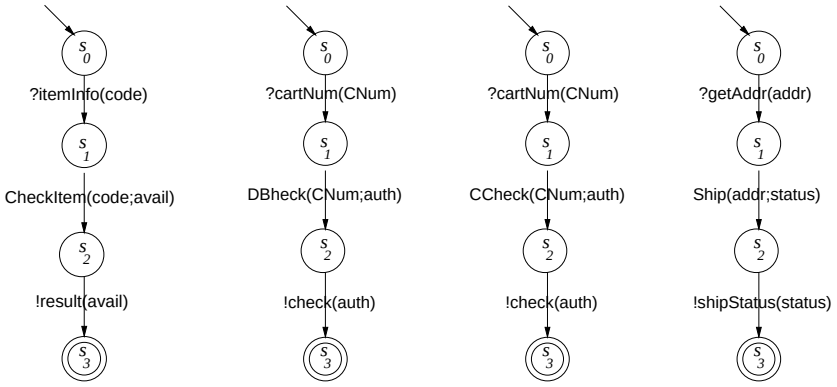


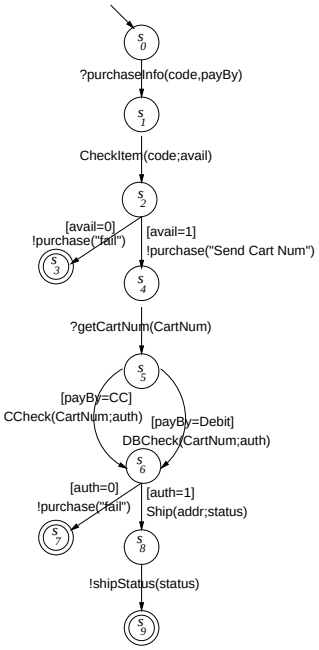
Figure 8.13 LTS representation of (a) `CheckItem` (b) `DebitCheck` (c) `CreditCheck` (d) `Shipping` component services

8.3.2 `e-Warehouse` Case Study

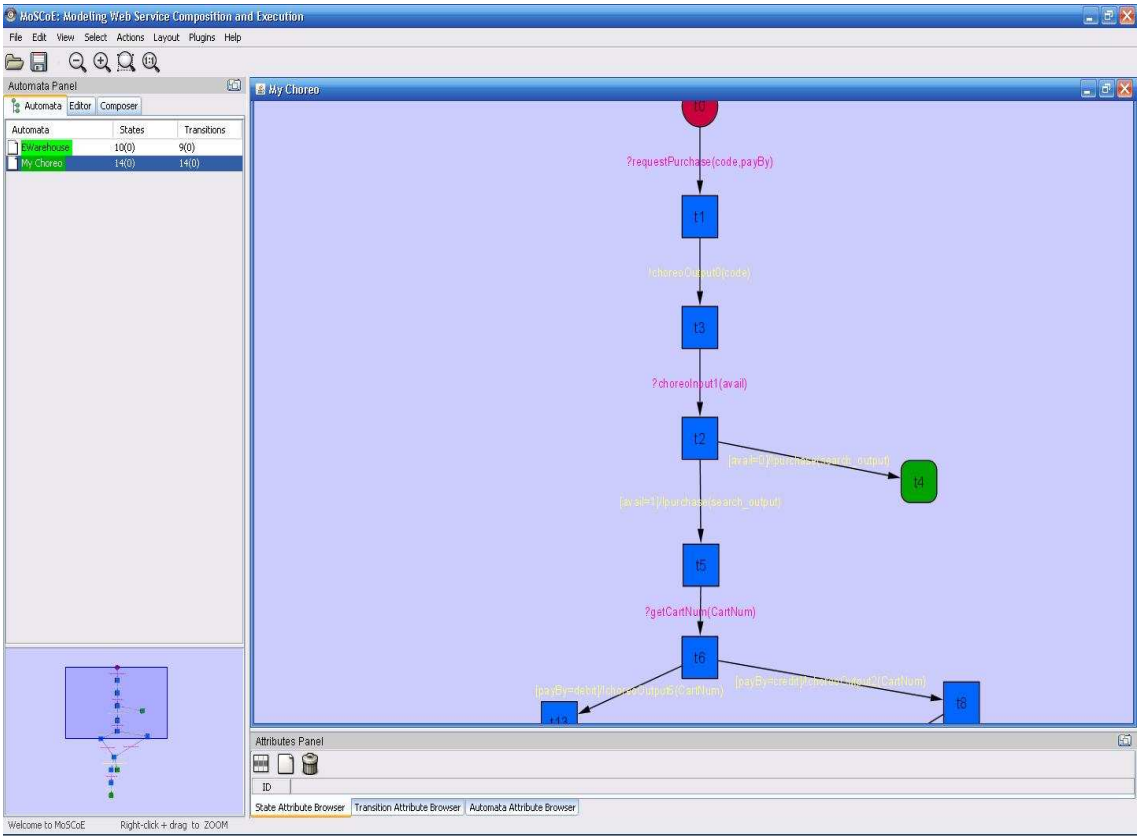
To further illustrate the applicability of our tool, we adopt the `e-Warehouse` case study presented by Berardi et al. [32]. Similar, to the previous case study, we assume that a service developer is assigned to model a composite service that will allow clients to search for a particular item of interest using an item code. If the required item is available, the service allows the client to purchase it using an appropriate payment method (credit or debit card) and also allows the item to be shipped to a particular address (depending on authorization of the form of payment). Figure 8.14(a) shows the LTS representation of the `e-Warehouse` goal service, whereas Figure 8.13 shows the set of available component services that can be analyzed to realize a mediator.

The service composition process, similar to the previous case study, is initiated by importing the BPEL/WSDL files of the corresponding goal and component services or modeling them directly using the LTS editor. In this case, we select all the services shown in Figure 8.13 along with the component services shown in Figure 8.9 for analysis. Figure 8.14(b) shows the mediator generated by the tool that realizes the `e-Warehouse` goal service[3].

[3]Due to higher resolution, Figure 8.14(b) shows the mediator only partially.



(a)



(b)

Figure 8.14 LTS representation of `e-Warehouse` (a) goal (b) mediator services

CHAPTER 9. CONCLUSIONS

9.1 Summary

Recent advances in networks, information and computation grids, and WWW have resulted in the proliferation of physically distributed and autonomously developed software components. These developments allow us to rapidly build new value-added applications from existing ones in various domains such as e-Science, e-Business, and e-Government. In this context, Service-Oriented Architectures (SOAs) based on Web services that offer standardized interface description, discovery and communication mechanisms are becoming an attractive alternative to build software components and to provide seamless application integration.

In this dissertation, we have addressed the problem of realizing a complex service through a composition of a subset of available component services. Specifically, we have presented a theoretically sound and complete approach for constructing a mediator that enables the interactions among component services to realize the behavior of the desired goal service. We use Labeled Transition Systems (LTSs) augmented with state variables over an infinite domain and guards over transitions to model the services. A unique feature of the proposed approach is its ability to work with an abstract (possibly incomplete) specification of a desired goal service. In the event the goal service cannot be realized (either due to incompleteness of the specification provided by the developer or the limited functionality of the available component services), the proposed technique identifies the causes for failure and communicates them to the service developer. The resulting information guides further iterative reformulation of the goal service until a composition that realizes the desired behavior is realized or the user (i.e., the service developer) decides to abort. Furthermore, the approach allows the developer to model services that satisfy non-functional requirements (e.g., Quality of Service) in addition

to the functional requirements.

We propose an approach for enabling Web service composition via automatic reformulation of the desired (or goal) service specifications in the event when the service composition algorithms fail to realize the goal service whenever the available component services cannot be used to "mimic" the structure of the goal service, even if the overall functionality of the goal service can be realized by an alternative formulation of the goal specification. In particular, we model services in our technique using labeled transition systems (LTS) and describe an efficient data structure and algorithms for analyzing data and control flow dependencies implicit in a user-supplied goal LTS specification to automatically generate alternate LTS specifications that capture the same overall functionality without violating the data and control dependencies implicit in the original goal LTS, and determine whether any of the alternatives can lead to a feasible composition. The result is a significant reduction in the need for the tedious manual intervention (by the service developers) in reformulating specifications by limiting such interventions to settings where both the original goal LTS as well as its alternatives cannot be realized using the available component services.

We addressed the problem of context-specific service substitution which requires that some desired property φ of the component being replaced is maintained despite its substitution by another component. We introduce two variants of the context-specific service substitutability problem, namely, environment-dependent and environment-independent substitutability that relax the requirements for substitutability relative to simulation or observational equivalence between services. The proposed solution to these two problems is based on the well-studied notion of "quotienting" which is used to identify the obligation of the environment of a service being replaced in a specific context. We demonstrate that both environment-dependent and environment-independent service substitutability problems can be reduced to quotienting of φ against the service being replaced and the replacement service and hence, to satisfiability of the corresponding mu-calculus formulae. The correctness of our technique follows from the correctness of the individual steps of quotienting and satisfiability.

We proposed a general technique for ontology-based service discovery and composition. In

particular, we introduce the notion of ontology-extended components and mappings between ontologies to facilitate discovery and composition of semantically heterogeneous component services. These results are particularly vital within the context of Service-Oriented Architectures, where services are autonomously developed and maintained, and hence semantic differences between the various messages, message types, action names, and so on are inevitable.

The algorithms and approaches designed through this dissertation are implemented as part of the MoSCoE (http://www.moscoe.org) framework and case studies demonstrating its applicability are presented.

9.2 Contributions

The main contributions of this dissertation include:

- **A General Framework for Iterative Composition of Web Services** [146, 148, 149, 150, 152, 153, 154, 155]

  We have proposed an interactive and verifiable framework Modeling Web Service Composition and Execution (MoSCoE). This framework provides the architectural foundation for incremental development of composite services based on theoretically sound and complete algorithms.

- **An Approach for Web Service Specification Reformulation**

  We have proposed an approach for enabling Web service composition via automatic reformulation of the desired (or goal) service specification by providing the ability to analyze control and data flow dependencies in the specifications to generate alternative models, such that the generated models retain the "overall" desired functionality of the goal service.

- **An Approach for Context-Specific Web Service Substitution** [151]

  We have proposed a general technique for context-specific Web service substitution, where context refers to the overall functionality of the composition that must be preserved after

the substitution. In particular, we introduce two variants of the context-specific service substitutability problem that are based on weaker and flexible requirements compared to existing techniques.

- **A Technique for Ontology-based Service Discovery and Composition** [156, 158]

  We have proposed a technique for enabling semantic interoperability between multiple services. Specifically, we introduce ontology-extended components and mappings between ontologies to facilitate discovery and composition of semantically heterogeneous component services.

- **Open-Source Implementation Framework for Web Service Composition**

  We provide an implementation of the proposed techniques for service composition in the MoSCoE prototype. The software is available under GNU public license at the MoSCoE website: http://www.moscoe.org.

9.3 Further Work

Several future research directions are outlined below:

- **Composition Efficiency**

  The practical feasibility of approaches to automated service composition is ultimately limited by the computational complexity of the service composition algorithms. However, the existing composition techniques run into exponential complexities and become impractical in real-world situations comprising of hundreds, if not thousands, of services. Hence, intelligent approaches and heuristics for reducing the number of candidate compositions that need to be examined are urgently needed in order to scale up service composition techniques sufficiently to make them useful in practice.

- **Execution Models**

Most of the existing implementations for composite Web service execution adopt a centralized architecture, that is, there exists an orchestrator (representing the composite service) in a centralized location responsible for coordinating and forwarding the intermediate results during the execution. Such a design has its limitation in terms of scalability, failure resiliency, and network bottlenecks. Towards this end, we believe that decentralized [39] or Peer-to-Peer (P2P) based architectures such as SELF-SERVE [27] will prove to be more beneficial in practical settings.

- **Failure Handling and Fault Tolerance**

  Web services are by nature autonomous and have an unpredictable behavior. For example, it is possible for a particular Web service W_i that is part of a composition to become inaccessible or updated (furnishing additional functions and/or removing existing ones, thereby altering its original behavior). Consequently, an existing integrated system (or a composite service) which comprises of multiple services including W_i, will require appropriate update in the form of replacing W_i. However, very limited research [25, 45] has been carried out to address this issue which needs further investigation. The problem becomes even more non-trivial when the replacement of the faulty service has to be carried out, while the composite service is being executed, in such a way that it is transparent to the client.

- **Security**

  Addressing security concerns is important for any Web-based system and various researchers have proposed mechanisms for ensuring security in Web services (see [91] for a survey). However, most of techniques build a trust-based framework or assume the existence of an environment, where once a service is identified to be "good" (loosely speaking) based on its security policy etc., it is considered to be trustworthy. However, in certain cases, even though a particular service is trustworthy, it might delegate a part of its functionality to another service which cannot be trusted. For example, an online air ticket reservation service W_x might delegate the process of verifying authenticity of

payment methods (e.g., credit card) required to purchase the air tickets to a third-party service provider W_y (in a manner transparent to the client), which may not follow the same security policy as W_x causing a potential security threat. Unfortunately, it is hard to detect such vulnerabilities. Furthermore, even if W_x claims to be "good", it may not strictly adhere to its own security policy, which makes it even harder to detect whether the integrity of client information has been compromised. We believe that addressing these two issues is a significant and important research challenge for the Web services community.

- **Semantic Mediation**

  Most of the existing and current work on Web service description standards, discovery, and composition techniques have focused on supporting interoperability at the syntactic level. However, the issues regarding structural and semantic heterogeneity between services themselves as well as the messages exchanged between them are quite complex and vital for ensuring smooth interoperability between services. In particular, the ability to mediate messages between semantically heterogeneous services still remains a problem at large that has received relatively less attention from the research community. Some of the interesting work in this direction include [133, 141, 174].

- **Tool Support**

  An important component of making techniques for automatic Web service composition useful for masses is to develop user-friendly tools and platforms that will allow non-experts to model complex services. Towards this end, model-driven based approaches [161] has shown some promise, although a lot of research has to be carried out, in particular by leveraging techniques from human-computer interaction and cognitive science.

- **Experimental Benchmark**

  At present, due to lack of a benchmark (dataset) of Web services, there is no uniform way of comparing, for example, an existing service composition algorithm with another. We

believe that developing a comprehensive benchmark and testbed of Web services will act as a quick aid for testing and ease of prototyping to evaluate different techniques. Such a benchmark should comprise of various hardware platforms and a variety of synthetic and real-world Web services. To the best of our knowledge, WSBen [136] is one of the preliminary efforts in this direction.

- **Applications**

  Web services and Service-Oriented Architectures are getting widely adopted in many domains including e-Science, e-Business and e-Government. In this context, some of the work in progress is aimed at application of the proposed techniques to service composition, substitution and adaptation tasks that arise in bioinformatics [193], electric power systems [157, 159, 122] and information retrieval [160, 167].

APPENDIX A. BPEL process description of e-Auction service

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns:bpws="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
         xmlns:ns="http://SampleProject/SampleProcessArtifacts"
         xmlns:ns0="http://SampleProject/SampleProcessInterface"
         expressionLanguage="http://www.w3.org/TR/1999/REC-xpath-19991116"
         name="SampleProcess"
         suppressJoinFailure="yes"
         targetNamespace="http://SampleProject">
  <import importType="http://schemas.xmlsoap.org/wsdl/" location="SampleProcessArtifacts.wsdl"
          namespace="http://SampleProject/SampleProcessArtifacts"/>
  <import importType="http://schemas.xmlsoap.org/wsdl/" location="SampleProcess.wsdl"
          namespace="http://SampleProject/SampleProcessInterface"/>
  <partners>
     <partner name="client" serviceLinkType="ns:SampleProcessPartnerLinkType" myRole="seller"/>
     <partner name="seller" serviceLinkType="ns:SampleProcessPartnerLinkType" partnerRole="seller"/>
  </partners>
 <variables>
    <variable name="Input" messageType="ns0:sellerRequest"/>
    <variable name="sellerSendData" messageType="ns0:sellerAnswerData"/>
    <variable name="sellerReplyData" messageType="ns0:sellerAnswerData"/>
    <variable name="Output" messageType="ns0:sellerResponse"/>
 </variables>
 <sequence name="Sequence">
    <receive createInstance="yes" name="receive1" operation="check" partnerLink="client"
            portType="ns0:SampleProcess" variable="Input"/>
    <assign name="AssignInputToSeller">
      <copy>
        <from variable="Input" part="requestParameters"/>
        <to variable="sellerSendData" part="sellerAnswerDataParameters"/>
      </copy>
    </assign>
    <invoke name="invokeSeller" partnerLink="seller" portType="as:sellerAnswerPT"
```

```
                    operation="check" inputVariable="sellerSendData"
                    outputVariable="sellerReplyData"/>
    <assign name="AssignSellerOutputToOutput">
          <copy>
            <from variable="sellerReplyData" part="sellerAnswerDataParameters"/>
            <to variable="Output" part="responseParameters"/>
          </copy>
    </assign>
    <reply name="reply1" operation="check" partnerLink="client" portType="ns0:SampleProcess" variable="Output"/>
  </sequence>
</process>
```

APPENDIX B. WSDL description of e-Auction service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:tns="http://SampleProject/SampleProcessInterface"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/" name="SampleProcess"
             targetNamespace="http://SampleProject/SampleProcessInterface">
  <types>
    <schema targetNamespace="http://SampleProject/SampleProcessInterface"
            xmlns:tns="http://SampleProject/SampleProcessInterface"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <element name="requestParameters">
        <complexType>
          <sequence>
            <element name="input" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="responseParameters">
        <complexType>
          <sequence>
            <element name="output" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="sellerAnswerData">
        <complexType>
          <sequence>
            <element name="sellerText" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
```

```
  <message name="sellerRequest">
    <part name="inputParameters" element="tns:requestParameters"/>
  </message>
  <message name="sellerResponse">
    <part name="operation1Result" element="tns:responseParameters"/>
  </message>
  <message name="sellerAnswerData">
    <part name="sellerAnswerDataParameters" element="tns:sellerAnswerData"/>
  </message>
  <portType name="sellerAnswerPT">
      <operation name="check">
        <input message="tns:sellerAnswerData" name="sellerAnswerData"/>
        <output message="tns:sellerAnswerData" name="sellerAnswerData"/>
      </operation>
  </portType>
</definitions>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
        xmlns:plnk="http://schemas.xmlsoap.org/ws/2004/03/partner-link/"
        xmlns:tns="http://SampleProject/SampleProcessArtifacts"
        xmlns:wsdl0="http://SampleProject/SampleProcessInterface"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns="http://schemas.xmlsoap.org/wsdl/" name="SampleProcess"
        targetNamespace="http://SampleProject/SampleProcessArtifacts">
        <plnk:partnerLinkType name="SampleProcessPartnerLinkType">
                <plnk:role name="seller" portType="wsdl0:SampleProcess" />
        </plnk:partnerLinkType>
        <import location="SampleProcess.wsdl" namespace="http://SampleProject/SampleProcessInterface" />
</definitions>
```

BIBLIOGRAPHY

[1] OASIS SOA Reference Model Technical Committee, Last accessed: 2nd June, 2007. URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.

[2] OMG Object Construct Language, Version 2.0, Last accessed: 7th June, 2007. URL ttp://www.omg.org/docs/ptc/05-06-06.pdf.

[3] W3C Web Ontology Language, http://www.w3.org/tr/owl-features/. URL http://www.w3.org/TR/owl-features/.

[4] Web Ontology Language for Web Services, http://www.daml.org/services/owl-s. URL http://www.daml.org/services/owl-s.

[5] V. Agarwal, K. Dasgupta, and et al. A Service Creation Environment Based on End to End Composition of Web Services. In *14th International Conference on World Wide Web*, pages 128–137. ACM Press, 2005.

[6] R. Akkiraju, B. Srivastava, A.-A. Ivan, R. Goodwin, and T. F. Syeda-Mahmood. SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition. In *4th IEEE International Conference on Web Services*, pages 37–44. IEEE CS Press, 2006.

[7] J. Alexander, D. Box, L. F. Cabrera, and et al. Web Services Enumeration. In *http://www.w3.org/Submission/WS-Enumeration/*, 2006.

[8] J. Alexander, D. Box, L. F. Cabrera, and et al. Web Services Transfer. In *http://www.w3.org/Submission/WS-Transfer/*, 2006.

[9] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[10] G. Alonso, F. Casati, H. Kuna, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[11] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 23–60. Springer-Verlag, London, UK, 1998. ISBN 3-540-65493-3.

[12] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002. ISSN 0004-5411.

[13] H. Andersen. Partial Model Checking (extended abstract). In *Logic in Computer Science*, 1995.

[14] S. Anderson, J. Bohren, and et al. Web Services Secure Conversation Language. In *http://www.ibm.com/developerworks/library/ws-secon/*, 2005.

[15] S. Anderson, J. Bohren, and et al. Web Services Trust Language. In *http://www.ibm.com/developerworks/library/ws-trust/*, 2005.

[16] T. Andrews, F. Curbera, and et al. Business Process Execution Language for Web Services, Version 1.1. In *http://www.ibm.com/developerworks/library/ws-bpel/*, 2003.

[17] G. Antoniou and F. van Harmelen. Web ontology language: OWL. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*. Springer-Verlag, 2003.

[18] A. Arnold, A. Vincent, and I. Walukiewicz. Games for Synthesis of Controllers with Partial Observation. *Theoretical Computer Science*, pages 7–34, 2003.

[19] S. Bajaj, D. Box, and et al. Web Services Policy 1.2 - Framework. In *http://www.w3.org/Submission/WS-Policy/*, 2005.

[20] K. Ballinger, B. Bissett, and et al. Web Services Metadata Exchange, Version 1.1. In *http://www.ibm.com/developerworks/webservices/library/specification/ws-mex/*, 2006.

[21] K. Ballinger, P. Brittenham, and et al. Web Services Inspection Language, Version 1.0. In *http://www.ibm.com/developerworks/webservices/library/ws-wsilspec.html*, 2001.

[22] S. Basu and R. Kumar. Quotient-based Control Synthesis for Non-Deterministic Plants with Mu-Calculus Specifications. In *45th IEEE Conference on Decision and Control*, 2006.

[23] S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and Symbolic Bisimulation Using Tabled Constraint Logic Programming. In *Intl. Conference on Logic Programming*, volume 2237, pages 166–180. Springer-Verlag, 2001.

[24] S. Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. *Theoretical Computer Science*, 354(2):211–229, 2006.

[25] B. Benatallah, F. Casati, and F. Toumani. Representing, Analysing and Managing Web Service Protocols. *Data and Knowledge Engineering*, 58(3):327–357, 2006.

[26] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Service Conversations. In *15th International Conference on Advanced Information Systems Engineering*, pages 449–467. LNCS 2681, Springer-Verlag, 2003.

[27] B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[28] B. Benatallah, Q. Z. Sheng, A. H. H. Ngu, and M. Dumas. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *18th International Conference on Data Engineering*, pages 297–308. IEEE Computer Society, 2002. ISBN 0-7695-1531-2.

[29] D. Berardi. *Automatic Service Composition: Models, Techniques and Tools*. PhD thesis, Università di Roma, La Sapienza, Italy, 2005.

[30] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic Composition of Web Services in Colombo. In *13th Italian Symposium on Advanced Database Systems*, pages 8–15, 2005.

[31] D. Berardi, D. Calvanese, D. G. Giuseppe, R. Hull, M. Lenzerini, and M. Mecella. Modeling Data and Processes for Service Specifications in `Colombo`. In *Workshop on Enterprise Modelling and Ontologies for Interoperability*, 2005.

[32] D. Berardi, D. Calvanese, D. G. Giuseppe, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *31st International Conference on Very Large Databases*, pages 613–624, 2005.

[33] D. Berardi, D. Calvanese, D. G. Giuseppe, M. Lenzerini, and M. Mecella. Automatic Service Composition based on Behavioral Descriptions. *International Journal on Cooperative Information Systems*, 14(4):333–376, 2005.

[34] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 2001.

[35] A. Betin-Can, T. Bultan, and X. Fu. Design for Verification for Asynchronously Communicating Web Services. In *14th International World Wide Web Conference*, pages 750–759, 2005.

[36] D. Beyer, A. Chakrabarti, and T. Henzinger. Web Services Interfaces. In *15th World Wide Web Conference*, pages 148–159. ACM Press, 2005.

[37] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory 1736-1936*. Oxford University Press, 1986. ISBN 0198539169.

[38] R. Bilorusets, D. Box, and et al. Web Services Reliable Messaging Protocol. In *http://www.w3.org/Submission/ws-messagedelivery/*, 2005.

[39] W. Binder, I. Constantinescu, and B. Faltings. Decentralized Orchestration of Composite Web Services. In *4th IEEE International Conference on Web Services*, pages 869–876. IEEE Computer Society, 2006.

[40] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[41] J. Bloomer. *Power Programming with RPC*. O'Reilly Media, Inc., 1992.

[42] F. Bolton. *Pure CORBA*. Sams Publishing, 2001.

[43] P. Bonatti, Y. Deng, and V. Subrahmanian. An Ontology-Extended Relational Algebra. In *IEEE International Conference on Information Integration and Reuse*, pages 192–199. IEEE CS Press, 2003.

[44] D. Booth, H. Haas, F. McCabe, and et al. Web Services Architecture, W3C Working Group Note 11. In *http://www.w3.org/TR/ws-arch/*, 2004.

[45] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *5th International Workshop on Technologies for E-Services*, pages 15–28. LNCS 3324, Springer-Verlag, 2004.

[46] A. Borgida and L. Serafini. Distributed description logics: Assimilating information from peer sources. *Journal of Data Semantics*, pages 153–184, 2003.

[47] J. P. Bowen and M. G. Hinchey. *Applications of Formal Methods*. Prentice Hall, 1995. ISBN 0133669491.

[48] D. Box, L. F. Cabrera, and et al. Web Services Eventing. In *http://www.w3.org/Submission/WS-Eventing/*, 2006.

[49] D. Box, E. Christensen, F. Curbera, and et al. Web Services Addressing, W3C Member Submission 10 August 2004, 2004.

[50] D. Box, M. Hondo, and et al. Web Services Policy Assertions Language, Version 1.0. In *http://www.ibm.com/developerworks/library/ws-polas/*, 2002.

[51] T. Bray, J. Paoli, and et al. Extensible Markup Language, Version 1.1. In *http://www.w3.org/TR/2004/REC-xml11-20040204/*, 2004.

[52] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *4th International Conference on Service-Oriented Computing*, pages 27–39. LNCS 4294, 2006.

[53] D. Bryan, V. Draluk, D. Ehnebuske, and et. al. Universal description discovery and integration, version 2.04. In *http://www.uddi.org*, 2002.

[54] V. Bullard and W. Vambenepe. Web Services Distributed Management. In *http://www.oasis-open.org/committees/wsdm/*, 2006.

[55] T. Bultan, J. Su, and X. Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[56] L. F. Cabrera, G. Copeland, and et al. Web Services Atomic Transaction, Version 1.0. In *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx*, 2005.

[57] L. F. Cabrera, G. Copeland, and et al. Web Services Coordination, Version 1.0. In *http://www.ibm.com/developerworks/library/ws-coor/*, 2005.

[58] D. Caragea, J. Pathak, , and V. Honavar. Learning classifiers from semantically heterogeneous data. In *Proceedings of the International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE 2004)*, 2004.

[59] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks*, 33(1-6):137–157, 2000.

[60] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava. Adaptation in Web Service Composition and Execution. In *4th IEEE International Conference on Web Services*, pages 549–557. IEEE CS Press, 2006.

[61] G. Chafle, P. Doshi, J. Harney, S. Mittal, and B. Srivastava. Improved Adaptation of Web Service Compositions using Value of Changed Information. In *5th IEEE International Conference on Web Services*, pages 784–791. IEEE CS Press, 2007.

[62] E. Christensen, F. Curbera, and et al. Web Services Description Language, Version 1.1. In *http://www.w3.org/TR/wsdl*, 2001.

[63] J. Clark and S. DeRose. XML Path Language, Version 1.0. In *http://www.w3.org/TR/xpath*, 1999.

[64] M. Crane and J. Dingel. On the Semantics of UML State Machines: Categorization and Comparision. In *Technical Report 2005-501, School of Computing, Queen's University, Canada*, 2005.

[65] L. de Alfaro and T. A. Henzinger. Interface Automata. In *9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[66] G. Della-Libera, M. Gudgin, and et al. Web Services Security Policy Language 1.1. In *http://www.ibm.com/developerworks/library/ws-secpol/*, 2005.

[67] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1):1–30, 2005.

[68] P. V. Eijk and M. Diaz, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., 1989. ISBN 0444872671.

[69] E. A. Emerson. Temporal And Modal Logic. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 995–1072. MIT Press, 1990. ISBN 0-444-88074-7.

[70] E. A. Emerson. Model Checking and the Mu-Calculus. In *Symposium on Descriptive Complexity and Finite Model*, pages 185–214. American Mathematical Society Press, 1997.

[71] E. A. Emerson and C. S. Jutla. The Complexity of Tree Automata and Logics of Programs. *SIAM Journal of Computing*, 29(1):132–158, 1999.

[72] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, New Jersey, 2004.

[73] D. Ferguson and M. Stockton. Service-Oriented Architecture: Programming Model and Product Architecture. *IBM Systems Journal*, 44(4):753–780, 2005.

[74] A. Ferrara. Web Services: A Process Algebra Approach. In *2nd International Conference on Service Oriented Computing*, pages 242–251. ACM Press, 2004.

[75] R. T. Fielding and R. N. Taylor. Principled Design of The Modern Web Architecture. In *22nd International Conference on Software Engineering*, pages 407–416. ACM Press, New York, NY, USA, 2000. ISBN 1-58113-206-9.

[76] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2003. ISBN 0321193687.

[77] I. R. Frank E. *DCOM: Microsoft Distributed Component Object Model*. John Wiley & Sons Inc., 1997.

[78] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *13th Intl. conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[79] X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[80] J. Fuller, M. Krishnan, and et al. Asynchronous Service Access Protocol, Version 1.0. In *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=asap*, 2006.

[81] G. C. Gannod, J. Timm, and R. J. Brodie. Facilitating the Specification of Semantic Web Services Using Model-Driven Development. *International Journal of Web Services Research*, 3(3):61–81, 2006.

[82] J. Gekas and M. Fasli. Automatic Web Service Composition Based on Graph Network Analysis Metrics. In *International Conference on Cooperative Information Systems*, pages 1571–1587. LNCS 3761, Springer-Verlag, 2005. ISBN 3-540-29738-3.

[83] A. Gerevini and D. Long. Preferences and Soft Constraints in PDDL3. In *ICAPS Workshop on Preferences and Soft Constraints in Planning*, 2006.

[84] C. Ghidini and L. Serafini. Distributed first order logics. In *Frontiers of Combining Systems 2*, volume 7, pages 121–139, 2000.

[85] F. Giannotti and G. Manco. Specifying Mining Algorithms with Iterative User-Defined Aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1232–1246, 2004. ISSN 1041-4347.

[86] S. Graham, D. Hull, B. Muray, and et al. Web Services Notification, Version 1.3 . In *http://www.oasis-open.org/committees/wsn*, 2005.

[87] R. Grønmo and I. Solheim. Towards Modeling Web Service Composition in UML. In *2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 72–86, 2004.

[88] T. Gruber. Ontolingua: A Mechanism to Support Portable Ontologies. In *Technical Report, KSL-91-66, Stanford University, Knowledge Systems Laboratory*, 1992.

[89] M. Gudgin, N. Mendelsohn, and et al. SOAP Message Transmission Optimization Mechanism. In *http://www.w3.org/TR/soap12-mtom/*, 2005.

[90] M. Gudin, M. Hadley, and et al. Simple Object Access Protocol. In *http://www.w3.org/TR/soap/*, 2003.

[91] C. Gutiérrez, E. Fernández-Medina, and M. Piattini. A Survey of Web Services Security. In *International Conference on Computational Science and Its Applications*, pages 968–977. LNCS 3043, Springer-Verlag, 2004.

[92] R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. In *14th Australasian Database Conference*, pages 191–200. Australian Computer Society, Inc., 2003.

[93] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000. ISBN 0262082896.

[94] J. Harney and P. Doshi. Adaptive Web Processes Using Value of Changed Information. In *4th International Conference on Service-Oriented Computing*, pages 116–128. LNCS 4294, 2006.

[95] J. Harney and P. Doshi. Speeding Up Adaptation of Web Service Compositions Using Expiration Times. In *16th World Wide Web Conference*, pages 1023–1032. ACM Press, 2007.

[96] S. V. Hashemian and F. Mavaddat. A Graph-Based Approach to Web Services Composition. In *IEEE/IPSJ International Symposium on Applications and the Internet*, pages 183–189. IEEE Computer Society, 2005. ISBN 0-7695-2262-9.

[97] S. V. Hashemian and F. Mavaddat. A Graph-Based Framework for Composition of Stateless Web Services. In *4th IEEE European Conference on Web Services*, pages 75–86. IEEE Computer Society, 2006. ISBN 0-7695-2737-X.

[98] H. Hass. Web Services Message Exchange Patterns. In *http://www.w3.org/2002/ws/cg/2/07/meps.html*, 2002.

[99] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988. ISBN 0262081717.

[100] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Welsey, 1979.

[101] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: A Look Behind the Curtain. In *22nd ACM Symposium on Principles of Database Systems*, pages 1–14. ACM Press, 2003.

[102] R. Hull and J. Su. Tools for Design of Composite Web Services. In *ACM SIGMOD Intl. Conference on Management of Data*, pages 958–961, 2004.

[103] R. Hull and J. Su. Tools for Composite Web Services: A Short Overview. *SIGMOD Record*, 34(2):86–95, 2005.

[104] K. Iwasa, J. Durand, and et al. Web Services Reliable Messaging, Version 1.1. In *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm*, 2004.

[105] E. M. C. Jr. and D. A. P. Orna Grumberg and. *Model Checking*. MIT press, 1999. ISBN 0262032708.

[106] A. Karmarkar, M. Hapner, and et al. Web Services Message Delivery, Version 1.0. In *http://www.w3.org/Submission/ws-messagedelivery/*, 2004.

[107] N. Kavantzas, D. Burdett, and et al. Web Services Choreography Description Language, Version 1.0. In *http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/*, 2005.

[108] A. Kleppe, J. Warmer, and W. Bast. *AMDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, 2003. ISBN 0132119442X.

[109] R. Kumar, C. Zhou, and S. Basu. Finite Bisimulation of Reactive Untimed Infinite State Systems Modeled as Automata with Variables. In *American Control Conference*, 2006.

[110] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994. ISBN 0-691-03436-2.

[111] V.-M. Kwan, F. C.-M. Lau, and C.-L. Wang. Functionality Adaptation: A Context-Aware Service Code Adaptation for Pervasive Computing Environments. In *IEEE/WIC International Conference on Web Intelligence*, pages 358–364. IEEE CS Press, 2003.

[112] S. Lam and A. Shankar. Protocol Verification via Projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, 1984.

[113] F. Leymann. Web Services Flow Language, Version 1.0. In *http://www.ebpml.org/wsfl.htm*, 2001.

[114] F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. In *3rd International Conference on Service-Oriented Computing*, pages 12–20. LNCS 3826, Springer-Verlag, 2005.

[115] L. Li and I. Horrocks. A Software Framework for Matchmaking based on Semantic Web Technology. In *12th Intl. Conference on World Wide Web*, 2003.

[116] Q. Liang and S. Su. AND/OR Graph and Search Algorithm for Discovering Composite Web Services. *International Journal of Web Services Research*, 4(2):48–67, 2005.

[117] F. Liu, L. Zhang, Y. Shi, L. Lin, and B. Shi. Formal Analysis of Compatibility of Web Services via CCS. In *1st International Conference on Next Generation Web Services Practices*, pages 143–148. IEEE Computer Society, 2005.

[118] H. Lockhart, J. Andersen, and et al. Web Services Federation Language, Version 1.1. In *http://www.ibm.com/developerworks/webservices/library/ws-fed/*, 2006.

[119] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Transactions on Internet Technology*, 5(3):439–479, 2005.

[120] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing Compatibility of BPEL Processes. In *International Conference on Internet and Web Applications and Services*, pages 147–155. IEEE CS Press, 2006.

[121] D. Martin, M. Burstein, J. Hobbs, and et al. OWL-S: Semantic Markup for Web Services, Version 1.1. In *http://www.daml.org/services/owl-s*, 2004.

[122] J. D. McCalley, V. Honavar, S. M. Ryan, W. Q. Meeker, D. Qiao, R. A. Roberts, Y. Li, J. Pathak, M. Ye, and Y. Hong. Integrated Decision Algorithms for Auto-steered Electric Transmission System Asset Management. In *7th International Conference on Computational Science*, pages 1066–1073. LNCS 4487, Springer-Verlag, 2007. ISBN 978-3-540-72583-1.

[123] D. V. McDermott. Estimated-Regression Planning for Interactions with Web Services. In *6th International Conference on Artificial Intelligence Planning Systems*. AAAI Press, 2002. ISBN 1-57735-142-8.

[124] S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[125] M. Mecella, B. Pernici, and P. Craca. Compatibility of e-Services in a Cooperative Multi-platform Environment. In *1st International Workshop on Technologies for e-Services*, pages 44–57. LNCS 2193, 2001.

[126] B. Medjahed and A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):954–968, 2005.

[127] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web services on the Semantic Web. *The Very Large Databases Journal*, 12(4):333–351, 2003.

[128] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[129] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[130] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[131] R. Murch. *Autonomic Computing*. IBM Press, 2004. ISBN 013144025X.

[132] A. Nadalin, C. Kaler, and et al. Web Services Security: SOAP Message Security 1.1. In *http://www.oasis-open.org/committees/wss/*, 2004.

[133] M. Nagarajan, K. Verma, A. P. Sheth, J. A. Miller, and J. Lathem. Semantic Interoperability of Web Services - Challenges and Experiences. In *4th IEEE International Conference on Web Services*, pages 373–382. IEEE CS Press, 2006.

[134] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[135] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *16th World Wide Web Conference*, pages 993–1002. ACM Press, 2007.

[136] S.-C. Oh, H. Kil, D. Lee, and S. R. T. Kumara. WSBen: A Web Services Discovery and Composition Benchmark. In *4th International Conference on Web Services*, pages 239–246. IEEE Press, 2006.

[137] S.-C. Oh, D. Lee, and S. Kumara. A Comparative Illustration of AI Planning-based Web Services Composition. *ACM SIGecom Exchanges*, 5(5):1–10, 2005.

[138] S.-C. Oh, B.-W. On, E. J. Larson, and D. Lee. Bf*: Web services discovery and composition as graph search problem. In *IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 784–786. IEEE Computer Society, 2005. ISBN 0-7695-2073-1.

[139] B. Orriëns, J. Yang, and M. P. Papazoglou. Model Driven Service Composition. In *1st International Conference on Service Oriented Computing*, pages 75–90. LNCS 2910, Springer-Verlag, 2003.

[140] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *1st Intl. Semantic Web Conference*, pages 333–347. Springer-Verlag, 2002.

[141] M. Paolucci, N. Srinivasan, and K. Sycara. Expressing WSMO Mediators in OWL-S. In *Semantic Web Services Workshop at International Semantic Web Conference*, 2004.

[142] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering*, pages 3–12. IEEE CS Press, 2003.

[143] M. P. Papazoglou. Web Services Technologies and Standards. *ACM Computing Surveys*, 2006 (under review). Available at: http://infolab.uvt.nl/pub/papazogloump-2006-97.pdf. Last accessed: 4th June, 2007.

[144] M. P. Papazoglou and D. Georgakopoulos. Introduction to Special Issue on Service-Oriented Computing. *Communications of ACM*, 46(10):24–28, 2003. ISSN 0001-0782.

[145] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-Oriented Computing: A Research Roadmap. In F. Cubera, B. J. Krämer, and M. P. Papazoglou, editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. ISSN 1862-4405.

[146] J. Pathak. MoSCoE: A Specification-Driven Framework for Modeling Web Services using Abstraction, Composition, and Reformulation. In *2nd IBM Ph.D. Symposium at 4th International Service Oriented Computing Conference*, pages 1–6. IBM Research Technical Report, RC24118, 2006.

[147] J. Pathak, J. Bao, D. Caragea, A. Silvescu, C. Andorf, C. Yan, D. Dobbs, and V. Honavar. INDUS: A System for Information Integration and Knowledge Acquisition from Autonomous, Distributed, and Semantically Heterogeneous Data Sources. In *13th Annual International Conference on Intelligent Systems for Molecular Biology, Demo Program*, 2005.

[148] J. Pathak, S. Basu, and V. Honavar. Modeling Web Service Composition using Symbolic Transition Systems. In *AAAI Workshop on AI-Driven Technologies for Service-Oriented Computing*, pages 44–51. AAAI Press Technical Report WS-06-01, 2006.

[149] J. Pathak, S. Basu, and V. Honavar. Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements. In *4th International Conference on Service Oriented Computing*, pages 314–326. LNCS 4294, Springer-Verlag, 2006.

[150] J. Pathak, S. Basu, and V. Honavar. Assembling Composite Web Services from Autonomous Components. In J. Soldatos and I. Maglogiannis, editors, *Emerging Artificial Intelligence Applications in Computer Engineering*, Frontiers in Artificial Intelligences and Applications, page xxx. IOS Press, 2007.

[151] J. Pathak, S. Basu, and V. Honavar. On Context-Sensitive Substitutability of Web Services. In *5th IEEE International Conference on Web Services*, pages 192–199. IEEE CS Press, 2007.

[152] J. Pathak, S. Basu, R. Lutz, and V. Honavar. MoSCoE: A Framework for Modeling Web Service Composition and Execution. In *IEEE 22nd Intl. Conference on Data Engineering Ph.D. Workshop*, page x143. IEEE CS Press, 2006. ISBN 0-7695-2571-7.

[153] J. Pathak, S. Basu, R. Lutz, and V. Honavar. Parallel Web Service Composition in MoSCoE: A Choreography-based Approach. In *4th IEEE European Conference on Web Services*, pages 3–12. IEEE CS Press, 2006.

[154] J. Pathak, S. Basu, R. Lutz, and V. Honavar. Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications. In *18th IEEE International Conference on Tools with Artificial Intelligence*, pages 445–454. IEEE CS Press, 2006.

[155] J. Pathak, S. Basu, R. Lutz, and V. Honavar. MoSCoE: An Approach for Composing Web Services through Iterative Reformulation of Functional Specifications. *International Journal on Artificial Intelligence Tools*, xxx(xxx):xxx, 2007.

[156] J. Pathak, D. Caragea, and V. Honavar. Ontology-Extended Component-Based Workflows-A Framework for Constructing Complex Workflows from Semantically Heterogeneous Software Components. In *2nd International Workshop on Semantic Web and Databases*, pages 41–56. LNCS 3372, Springer-Verlag, 2004.

[157] J. Pathak, Y. Jiang, V. Honavar, and J. McCalley. Condition Data Aggregation with Application to Failure Rate Calculation of Power Transformers. In *39th Annual Hawaii Intl. Conference on System Sciences*. IEEE Press, 2006.

[158] J. Pathak, N. Koul, D. Caragea, and V. Honavar. A Framework for Semantic Web Services Discovery. In *7th ACM Intl. Workshop on Web Information and Data Management*, pages 45–50. ACM press, 2005.

[159] J. Pathak, Y. Li, V. Honavar, and J. McCalley. A Service-Oriented Architecture for Electric Power System Asset Management. In *2nd International Workshop on Engineering Service-Oriented Applications: Design and Composition*, pages 24–35. LNCS 4652, Springer-Verlag, 2006.

[160] J. Pathak, N. Rajamani, W. Zadrozny, Y. Deng, M. Devarakonda, and H. Sachar. An Information Management Framework for Improving Engagement Efficiency in Services Business. In *IBM Research Technical Report*, 2006.

[161] K. Pfadenhauer, S. Dustdar, and B. Kittl. Challenges and Solutions for Model Driven Web Service Composition. In *14th IEEE Intl. Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pages 126–131. IEEE Press, 2005.

[162] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *19th Internatinal Joint Conferences on Artificial Intelligence*, pages 1252–1259, 2005.

[163] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *15th Intl. Conference on Automated Planning and Scheduling*, pages 2–11, 2005.

[164] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *3rd IEEE International Conference on Web Services*, pages 293–301. IEEE Press, 2005.

[165] M. Potts, I. Sedukhin, and et al. Web Service Manageability Specification, Version 1.0. In *http://www.ibm.com/developerworks/webservices/library/ws-manage/*, 2003.

[166] J. Radatz and M. S. Sloman. A Standard Dictionary for Computer Terminology: Project 610. *IEEE Computer*, 21(2), 1988.

[167] N. Rajamani, M. Devarakonda, Y. Deng, W. Zadrozny, and J. Pathak. Business-Activity Driven Search: Addressing the Information Needs of Services Professionals. In *5th IEEE International Conference on Services Computing*, pages 644–651, 2007.

[168] C. R. Ramakrishnan. A Model Checker for Value-Passing Mu-Calculus Using Logic Programming. In *3rd International Symposium on Practical Aspects of Declarative Languages*, pages 1–13. Springer-Verlag, London, UK, 2001. ISBN 3-540-41768-0.

[169] J. Rao, P. Kungas, and M. Matskin. Logic-based Web Services Composition: From Service Description to Process Model. In *2nd IEEE International Conference on Web Services*, pages 446–453. IEEE CS Paper, 2004.

[170] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *1st Intl. Workshop on Semantic Web Services and Web Process Composition*, pages 43–54, 2004.

[171] J. Reinoso-Castillo, A. Silvescu, D. Caragea, J. Pathak, and V. Honavar. Information Extraction and Integration from Heterogeneous, Distributed, Autonomous Information Sources: A Federated, Query-Centric Approach. In *IEEE Intl. Conference on Information Integration and Reuse*, pages 183–191, 2003.

[172] D. Robins. Interactive Information Retrieval: Context and Basic Notions. *Informing Science*, 3(2):57–62, 2000.

[173] D. Robitaille. *Root Cause Analysis: Basic Tools and Techniques.* Paton Press, 2004. ISBN 1932828028.

[174] D. Roman, U. Keller, H. Lausen, and et al. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[175] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[176] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *2nd IEEE International Conference on Web Services*, pages 43–50. IEEE Computer Society, 2004.

[177] F. O. Silva and P. F. Rosa. The Quest for the Web Services Stack: A Fast Trip. In *6th International Conference on Web Engineering*, pages 93–94. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-352-2.

[178] R. Sinha, P. Roop, and S. Basu. A Model Checking Approach to Protocol Conversion. In *Workshop on Model-driven High-level Programming of Embedded Systems*, 2007.

[179] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition using SHOP. *Journal of Web Semantics*, 1(4):377–396, 2004.

[180] D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *8th IEEE Intl. Enterprise Distributed Object Computing Conference*, pages 47–57. IEEE Press, 2004.

[181] B. Srivastava, J. P. Bigus, and D. A. Schlosnagle. Bringing Planning to Autonomic Applications with ABLE. In *1st International Conference on Autonomic Computing*, pages 154–161. IEEE Computer Society, 2004. ISBN 0-7695-2114-2.

[182] C. Stirling. Games and Modal Mu-Calculus. In *Second International Workshop Tools and Algorithms for Construction and Analysis of Systems*, pages 298–312, 1996.

[183] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[184] M. ter Beek, A. Bucchiarone, and S. Gnesi. Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *2nd International Conference on Internet and Web Applications and Services*, pages 15–20. IEEE CS Press, 2007.

[185] R. Thompson. Web Services for Remote Portlets Specification, Version 2.0. In *http://www.oasis-open.org/committees/wsrp/*, 2007.

[186] J. Timm and G. Gannod. A Model-Driven Approach for Specifying Semantic Web Services. In *3rd International Conference on Web Services*, pages 313–320. IEEE press, 2005.

[187] J. Timm and G. Gannod. Specifying Semantic Web Service Compositions using UML and OCL. In *5th International Conference on Web Services*, page xxx. IEEE press, 2007.

[188] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *3rd International Semantic Web Conference*, pages 380–394. Springer-Verlag, 2004.

[189] A. Tsalgatidou and T. Pilioura. An Overview of Standards and Related Technology in Web Services. *Distributed and Parallel Databases*, 12(2-3):135–162, 2002. ISSN 0926-8782.

[190] S. Vinoski. WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):94–96, 2004. ISSN 1089-7801.

[191] M. Weske. *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects (Habilitationsschrift)*. PhD thesis, Fachbereich Mathematik und Informatik, UniversitMnster, Germany, 2000.

[192] G. Woods and T. Gullotta. Web Services Provisioning, Draft Version 0.7. In *http://www.ibm.com/developerworks/webservices/library/ws-provis/*, 2003.

[193] F. Wu, P. Zaback, J. Pathak, C. Yan, N. Koul, X. Li, D. Dobbs, and V. Honavar. PPIDB: A Comprehensive Database of Protein-Protein Interfaces. In *Nucleic Acids Research, Database Issue*, Submitted, 2008.

[194] P. Yang, S. Basu, and C. Ramakrishnan. Parameterized Verification of Pi-Calculus Systems. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 42–57. LNCS 3920, Springer-Verlag, 2006.