2009

Extending substitutability in composite services by allowing asynchronous communication

Zachary James Oster
Iowa State University

Extending substitutability in composite services

by allowing asynchronous communication

by

Zachary James Oster

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Samik Basu, Major Professor
Vasant Honavar
Robyn R. Lutz

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to everyone who has guided and supported me in my research and in writing this thesis. In particular, I thank Dr. Samik Basu for providing me with inspiration, motivation, and wise advice throughout the time he has served as my major professor. I am grateful for the time that he has spent helping me refine my research ideas and improve this thesis. I also thank thank Dr. Vasant Honavar and Dr. Robyn Lutz for serving on my committee and for their help in my research.

I would like to thank Dr. Jyotishman Pathak for leading previous development of the MoSCoE framework and for his work on Web service substitutability, on which this work is based. I would also like to thank Tanmoy Sarkar for his assistance with the initial implementation of the quotienting tool presented in this thesis and Ganesh Ram Santhanam for his help in redesigning the MoSCoE data structures. In addition, I thank the National Science Foundation (grants CNS0709217 and CCF0702758) for providing funding for my research.

Finally, I thank my family for their constant support of my academic pursuits — especially my wife Carrie, without whom I probably would never have come this far.

ABSTRACT

Web services are programs that are self-contained, self-describing, interoperable, platform-independent, and accessible over a network. These properties allow several Web services to be combined together to form a Web service composition. However, when a component service within a Web service composition becomes unavailable or unusable, it is necessary to identify a substitute service that can replace the failed component while preserving the original functionality of the composition. This is the problem of Web service substitution.

Most existing work that addresses this problem requires strict functional equivalence between the original component service and its substitute. In contrast, Pathak et al. have shown in 2007 that it is sufficient for a substitute service to provide the same functionality with respect to the rest of the composition as the component it is replacing. Pathak et al. apply a technique called quotienting to determine the portion of the composition's overall functionality that is satisfied by the original component. The quotienting operation yields the property that must be satisfied by a substitute for that component.

While the use of quotienting allows more possible substitute services to be accepted, it is possible to relax the substitutability condition even further by considering asynchronous communication between component services within the Web service composition model. Our work accomplishes this task by providing a formal framework for representing asynchronous communication within a Web service composition. In our framework, the asynchronous communication is encapsulated in a buffer process, which stores each message until a component is ready to consume it. We prove the correctness of our solution, describe our implementation, and discuss some directions for future research.

CHAPTER 1. INTRODUCTION

1.1 Web Services: An Overview

The use of Web services is rapidly gaining popularity in industry, government, and academia as an effective alternative to traditional software for many applications. A *Web service*, or simply a *service*, is a self-contained and self-describing program that is designed to be interoperable with other Web services, regardless of hardware platform, language, or operating system, and that is made available for use over a network.[1] Web services communicate with each other and with their users by sending and receiving messages according to a common protocol; a group of services can successfully interact with each other, i.e., the services are *interoperable*, if all services in the group communicate using the same protocol. Information describing what each Web service does, how it can be accessed, and other properties of the service is made publicly available in order to allow potential users to discover and make use of the service. Web services are commonly used for tasks such as providing real-time Web-based access to software on a remote system or querying multiple large off-site data stores and then aggregating and presenting the query results. Several popular Web-based software applications, including Google Docs [20] and Microsoft Office Live [29], make extensive use of Web services to provide their functionality to users without requiring the users to install or maintain a complete copy of the software on their own systems.

The use of Web services in software development presents a number of interesting and challenging problems. One of these is the problem of *Web service composition*, which entails finding a way to assemble several existing Web services into a new composite service in order to provide some functionality required by the user. This is made possible by the fact that Web

[1] Throughout this thesis, the terms "Web service" and "service" will be used interchangeably.

services are interoperable, self-contained, and self-describing, which makes them ideal for reuse. Web service composition has the potential to greatly reduce redundant software development efforts and thus provide a way to deliver better software applications more quickly, especially if the process of composition development can be partially or fully automated. Although significant theoretical and practical problems currently prevent the wider use of Web service composition, many techniques have been developed to attempt to deal with these problems during the past decade; please refer to [15, 23] for a survey of some of these techniques. Closely related to the Web service composition problem is the problem of *Web service substitution*, also called *Web service substitutability*, which requires determining the condition under which one service may be replaced by a substitute service without loss or reduction of functionality in the event that it becomes unavailable or unusable.[2] Solving this problem entails determining the substitutability condition for the service with respect to the other services with which it interacts. The *substitutability condition* is defined as the minimum functionality that must be provided by a substitute service in order to preserve the full functionality of the original service with respect to the process in which it is used. Determining a substitutability condition for a Web service is useful because this allows substitutes for a Web service to be identified in advance of failures; in the event that the original service fails, a previously identified substitute service can be quickly deployed to replace it, thus minimizing downtime. Other notable problems, including the lack of consistent standards for specifying various properties of Web services and the difficulty of determining whether services are compatible with one another, also affect the use of Web services; however, these problems are beyond the scope of our work. We will focus primarily on the Web service substitution problem in this thesis.

Much of the existing work on Web service substitution defines the substitutability condition for any Web service in any context to be behavioral equivalence between the original service and the proposed substitute for that service. As a result, the existing work is generally focused on determining whether a proposed substitute service is behaviorally equivalent to the service being replaced. Some of the more notable research efforts along this track are presented in

[2]The terms "Web service substitution" and "Web service substitutability" will be used interchangeably throughout this thesis to refer to this problem.

Section 2.3. While this condition is always sufficient to permit substitution, it is too strong because requiring behavioral equivalence to the original service limits service substitution to instances where a nearly exact match for the original service can be found. In addition, existing work on Web service substitution often assumes that services in a Web service composition always communicate synchronously, i.e., each output message from one component must be consumed immediately by another component as an input message. This assumption is usually made to simplify the process of finding the substitutability condition, but it unintentionally introduces an implicit requirement that a substitute service must perform the required tasks in the same order as the original service. If Web services in a composition can communicate asynchronously with each other, i.e., if there may be some delay between the time when one component sends an output message and the time when another component consumes that message as an input,[3] then it is necessary only to show that all required tasks will be performed and all required properties will hold, regardless of the order in which the tasks are performed.

The rigid requirement of behavioral equivalence as a substitutability condition was first relaxed by Bordeaux et al., who presented the concept of *context-dependent substitutability* in [9] as an alternative to requiring behavioral equivalence. Pathak, Basu, and Honavar further relaxed the behavioral equivalence requirement in [35] by extending the idea of context-dependent substitutability to account for both the composition where the substitution must occur and the properties of the composition that must be preserved after the substitution is completed — i.e., the *environment* in which the substitute service will be deployed — in addition to the behavior of the original service. Pathak et al. employed a technique called *quotienting* to reduce properties that must be satisfied by an entire Web service composition by removing those parts of the required properties that are satisfied by other components of the composition, leaving only the properties that must be satisfied by the component being considered. It was proven in [35] that considering the environment of a proposed substitution results in a significant relaxation of the required substitutability condition, which allows more Web services to

[3] We are not considering real-time systems and/or timeout delays, nor are we explicitly considering the effects of network latency and/or data loss. Our objective is to show that it is not necessary to require "lock-step" movement of services in a composition.

be identified as valid candidates for substitution. The work of Pathak et al. on Web service substitution was an outgrowth of their original work on Web service composition, which resulted in the development of the MoSCoE (Modeling Service Composition and Execution) [36] framework. The MoSCoE framework, which is meant to serve as a test bed for future work in Web services, currently includes designs and partial implementations of tools to assist with composition development and analysis.

Like the majority of previous work on Web service substitution, the context-specific substitutability analysis presented in [35] assumes that the services in the composition being analyzed use only synchronous communication. Our work in this thesis aims to relax this assumption by considering asynchronous communication, making it possible to provide a method to determine the true minimum functionality that must be provided by a substitute for a service in a composition, i.e., the most relaxed substitutability condition for a given service in a given context. We will accomplish this task by providing a method for constructing a buffer process that can be used to treat a composition of asynchronously communicating Web services as if its components communicate synchronously, which will allow us to apply the methods for context-sensitive substitutability analysis presented in [35] to asynchronously communicating Web services. By accomplishing this objective, we aim to increase the number of valid substitute services that are considered during a substitutability analysis (i.e., to decrease the number of false negatives when analyzing potential substitutes for a component of a Web service composition) and to provide a theoretical framework for substitutability analysis of Web service compositions whose components communicate asynchronously.

The problem we aim to solve in this thesis may be stated formally as follows:

Problem Specification: Suppose there exists a composition of services Q comprising n component services $Q_1, Q_2, \ldots, Q_n$ that satisfies some required property φ describing the functionality of the composition, and suppose that one or more components of the composition Q_i (where $i \in \{1, 2, \ldots, n\}$) become unavailable or unusable. Does there exist a service (or composition of services) Q_i' that may be substituted for Q_i such that the new composition Q' resulting from the substitution satisfies the property φ?

1.2 Contributions

The contributions of our work are summarized as follows:

1. We address the service substitution problem, going beyond existing work by taking into consideration the asynchronous communication paradigm and the context in which the substitution takes place.

2. We provide a formal framework for substitutability analysis of asynchronous Web service compositions that includes a transition system-based representation of services and a mu-calculus description of properties. Since the formalism is based on mu-calculus (unlike Linear Temporal Logic as in [12]), our formalism and results can be immediately extended to use other temporal logics such as CTL, CTL*, and LTL.

3. We prove the correctness of our technique and provide a way to design the required buffering of messages to allow for correct handling of asynchronous communication between components of a composition.

4. We present a prototype implementation of our technique, which features XML-based input and output, a robust data structure for representing mu-calculus properties, and compatibility with the MoSCoE composition tool presented in [36].

1.3 Outline of Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2**: A survey of the existing literature in Web services is presented, with special attention to previous approaches to the substitution problem. Because the substitution problem is best understood within the context of the composition problem, formalisms for representing Web services, as well as major paradigms and existing techniques for Web service composition, are also discussed.

- **Chapter 3**: Background information needed to understand the remaining chapters is given, including formal definitions of the major concepts used in our framework and a

summary of previous work on the substitution problem for synchronously communicating services.

- **Chapter 4**: Our formal framework for the substitution problem for asynchronously communicating services is presented, including a proof of correctness and a complexity analysis.

- **Chapter 5**: Information about our implementation of the framework presented in Chapter 4 is given, along with a discussion of the updates and changes made to the existing implementation of the MoSCoE framework for Web service composition.

- **Chapter 6**: The contributions of this thesis are summarized and possible future directions for research are set forth.

CHAPTER 2. RELATED WORK

2.1 Formalisms for Representing Web Services

A large number of formalisms have been proposed for representing the structure and behavior of Web services. These formalisms can generally be categorized as either high-level languages or state machine-based models. This section summarizes the common uses, advantages, and disadvantages of each category of Web service representations and presents examples from each category.

2.1.1 High-Level Languages

The first major category of Web service representations to be covered consists of specification languages that use structured text to describe the structure and function of Web services. These languages resemble high-level programming languages to some degree. Most of these languages use a syntax based on XML [10] and defined in an XML Schema [21] document.

Two of these high-level specification languages that have gained relatively wide acceptance in the Web services community are WSDL and BPEL. WSDL (Web Services Description Language) [13] is a W3C recommendation for describing Web services and is a part of the W3C Web Services Architecture [8]. WSDL allows a service to be described at an abstract level by specifying the types of messages that the service may send and receive, as well as the operations supported by the service. It also supports the specification of concrete details about how to connect to the service. WSDL does not, however, provide a way to represent the internal behavior of a service. BPEL (Business Process Execution Language) [2] is a specification language that was developed by a consortium of industry leaders, including BEA, IBM, and Microsoft. BPEL is built upon the WSDL model of Web services, modeling the given

business process and its partners as Web services using WSDL, but it also provides additional features that can be used to specify both the behavior of individual services and the desired protocols for exchanging messages between services.

WSDL and BPEL are useful for two main reasons. First, they are both accepted as standards for describing Web services. WSDL is a standard because of its status as a W3C recommendation, and BPEL can be considered a *de facto* standard because of its support from IBM, Microsoft, and others. Second, the syntax for both WSDL and BPEL is based on XML, and each language provides its own XML Schema document for validation. This allows service description documents written in either language to be validated and parsed on any computing platform using standard XML document processing tools, which facilitates the development of tool support for Web services specified using these languages.

However, these high-level languages have two serious disadvantages. First, high-level languages, even BPEL, cannot directly represent the internal behaviors of Web services in an easily understandable way; these languages are meant as aids for discovering and accessing Web services, not for analyzing their behavior. Second, formal verification cannot be directly applied to Web service descriptions written in high-level languages, because the semantics of formal methods are generally not compatible with the semantics of these languages.

2.1.2 State Machine-Based Models

The other major category of Web service representations consists of models that are based on the concept of a state machine. State machines can be used to provide low-level, implementation-independent representations of Web services. Any Web service that can be represented using WSDL, BPEL, or any other language can be represented as an equivalent state machine. Models based on state machines are more useful for analyzing Web services than high-level language representations because state machine-based models can represent both the internal behaviors of a Web service and the interactions of a service with its partners in a way that is both machine-readable and intuitively understandable to a human viewing the representation. In addition, many formal logics define their semantics in terms of state ma-

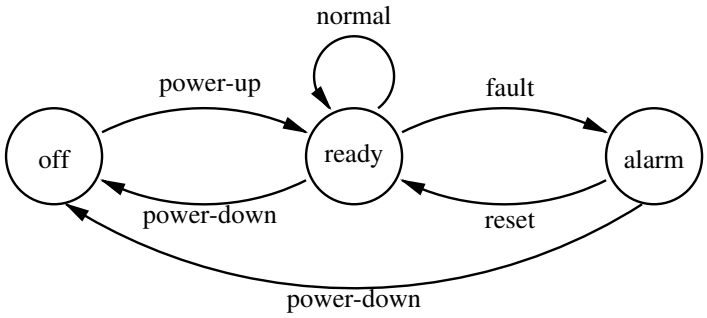


Figure 2.1 Example of a state machine

chines or related formalisms. By modeling Web services as state machines, it becomes possible to rigorously verify desired properties of Web services against the models of those services.

A state machine consists of a set of states, a set of transitions connecting the states in some way, and a set of labels that may be assigned to transitions. Each label generally represents an action that is taken during a transition or a condition that must be true in order to enable a transition, but labels may also be assigned other meanings; for purposes of this thesis, a label will always represent an action that occurs during a transition from one state to another state. State machines are often illustrated as directed graphs, where each node represents a state and each edge represents a transition. Figure 2.1 depicts a state machine that monitors the status of a system and sounds an alarm if a fault occurs in the system. This state machine begins in the *off* state, moving to the *ready* state when the system is powered up. The state machine remains in the ready state as long as the system reports a normal status. If the system reports a fault, then the state machine moves to the *alarm* state; it remains there until a reset command is received from an operator, at which point the state machine returns to the *ready* state. The state machine will also move from either *ready* or *alarm* to *off* when the system is powered down.

Many different state machine-based models exist in the Web services literature. Mecella et al. [28] use the Unified Modeling Language (UML) [18] to represent both the structure and behavior of services: the structure is specified in a UML class diagram and the behavior is given as a UML statechart diagram, which is an alternative visual representation of a state machine.

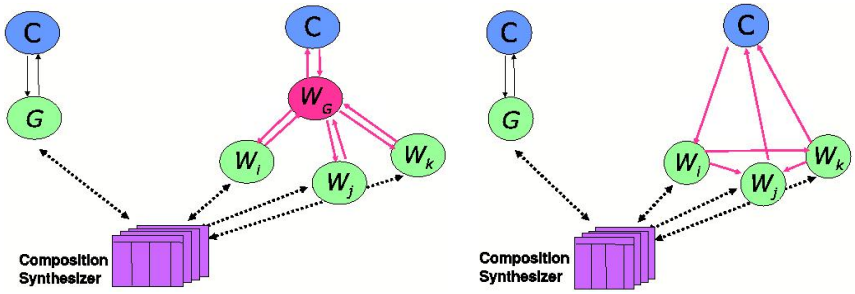
Bultan, Fu, and Su [12] model services as state machines with attached FIFO queues to provide for asynchronous message passing between services; a similar model that uses Büchi automata, which are finite state machines that can accept infinite inputs, is used in [19]. Several groups model Web services using Petri nets [31], which are more powerful than state machines. Petri nets can be viewed as state machines augmented with a well-defined token-passing semantics that may be used for formal verification of properties. Martens et al. [27] transform services specified in BPEL into Petri nets, while Hamadi and Benatallah [22] model services directly as Petri nets. A number of authors, including [9, 24, 35, 37], use labeled transition systems [30] to represent services, as we do in our work; these are essentially finite state machines that designate a start state, although they may also be annotated with additional information. Liu et al. [26] use CCS processes [30], whose semantics are given in terms of labeled transition systems, to represent services. Pathak et al. [36] use symbolic transition systems, which are similar to labeled transition systems but also designate a set of final states.

2.2 Web Service Composition Tools and Techniques

Recall from Chapter 1 that the process of combining several component services into a single composite service is known as *Web service composition*. Because Web services are interoperable and reusable, several Web services can be combined into a single composite Web service (a *composition*, for short) that provides the full functionality of its combined components. Web service composition is a difficult problem that involves specifying the desired functionality of the composition, identifying appropriate services to include in the composition, determining whether the chosen component services are compatible with one another, and verifying whether the finished composition actually provides the intended functionality.

Research on Web service composition generally approaches the problem from one of two perspectives: the "orchestration" perspective and the "choreography" perspective.[1] Figure 2.2 illustrates the differences between the two perspectives. In each diagram, the desired goal ser-

[1]Unfortunately, although these terms are used commonly in the Web service composition literature, there is disagreement as to the meaning of each term. For our purposes, we will use the definitions given by Papazoglou et al. in [33], as these definitions seem to reflect a majority opinion in the literature.



(a) (b)

Figure 2.2 Two perspectives on Web service composition [6]: (a) Orchestration, (b) Choreography

vice is shown on the left, and the resulting composition as viewed from the given perspective is shown on the right. The "orchestration" perspective assumes the existence of a supervisory process (a "conductor" or "orchestrator") that exists separately from the components of the composition. This orchestrator process acts as a central point of control for all communication between components; instead of communicating with each other, the components communicate only with the orchestrator process. This approach emphasizes the high-level sequence of actions — in other words, the process — being executed by the composition as a whole. In contrast, the "choreography" perspective treats a composition as a set of processes that interact directly, without any centralized control structure. In this perspective, the component services are viewed as peers in a peer-to-peer system that agree to collaborate by following common interaction rules. This approach focuses on the low-level interactions between components. Although most existing Web service composition research chooses to approach the problem from only one of these perspectives, Papazoglou et al. argue that a complete solution to the composition problem will need to incorporate both perspectives [33].

Many different models, frameworks, and methods for Web service composition have been proposed. Although a full survey of the Web service composition literature is beyond the scope of this chapter, several researchers' work in this area informs our work on Web service substitutability. Pistore et al. [37] apply a technique that they describe as "planning via symbolic model checking" to generate a Web service composition from a set of Web services

represented as labeled transition systems and a set of requirements specified as a formula in the requirements language EAGLE. Hamadi and Benatallah [22] define several composition operators, each with somewhat different behavior, to form what they call a "service algebra." The operators are defined in terms of Petri nets for compatibility with their Petri net-based Web service model, so services modeled according to their framework may be easily composed. One of the most important models of Web service composition is the conversation model presented by Bultan et al. in [11]. According to the conversation model, a Web service composition can be viewed as a set of Mealy machines, or finite state machines with input and output, that communicate asynchronously using message passing. Although [11] focuses on the analysis of Web service compositions rather than their creation, the conversation model has informed a number of other researchers' approaches to various problems in Web services, including our approach to the substitutability problem. The `Colombo` framework developed by Berardi et al. [6] is directly influenced by the conversation model, but its model of Web service composition is much more sophisticated, incorporating a database representing the current world state, atomic processes corresponding to Web services, and various other aspects that make it one of the most detailed representations of Web service composition.

Our work on Web service substitutability is based on the MoSCoE framework for Web service composition presented by Pathak, Basu, and Honavar in [36]. MoSCoE incorporates ideas from each of the approaches to Web service composition mentioned here, as well as several others. The input to the MoSCoE framework consists of a set of symbolic transition systems (STS) corresponding to the available repository of Web services and an STS that models a goal service, which is an abstract and perhaps incomplete specification of the desired functionality to be provided by the composition. Given this input, if it is possible to create a Web service composition that provides the functionality specified by the goal service, the MoSCoE composition algorithm identifies a set of Web services that can form a satisfactory composition and creates a new service that provides the necessary communication links between the component services. If it is impossible to realize a composition that models the goal service, the MoSCoE composition algorithm notifies the user that the composition process

has failed and tells the user exactly which states or transitions in the goal service could not be satisfied. This feature, which is unique to MoSCoE (to the best of our knowledge), is meant to give the user enough information to reformulate the goal specification and attempt to form a composition again; the process can continue iteratively until either a composition is successfully created or the user stops trying. A previous working implementation of the MoSCoE composition algorithm exists, but it is difficult to use because it is tightly coupled to an unstable, non-intuitive graphical user interface and because some of the auxiliary features, including translation between BPEL specifications and STS models, are not fully functional. As part of our work, we have updated this existing MoSCoE implementation to use a simpler set of core data structures, provide support for XML-based input and output, and remove features that were not fully implemented. Our goal in doing so was to establish a solid foundation for future additions and modifications to the MoSCoE framework, providing support for the framework's purpose as a testbed for new Web service research ideas.

2.3 Approaches to the Web Service Substitutability Problem

While the problem of Web service substitutability — identifying conditions that must be satisfied in order for one or more services to be replaced by one or more other services — has been studied by a number of researchers, much of the existing work on this problem has attempted to find ways to establish behavioral equivalence between the service(s) being replaced and the substitute service(s). These approaches differ primarily in the methods employed for computing equivalence between services. To determine equivalence between services, Benatallah et al. [5] and Taher et al. [40] use similarity, Bordeaux et al. [9] use bisimilarity, and Martens et al. [27] use trace equivalence. Other approaches for determining equivalence include subsumption ordering [7] and graph matching [25].

Much of the work on determining substitutability by finding behavioral equivalence between services makes the assumption, implicitly or explicitly, that behavioral equivalence is always a necessary condition for substitutability. However, Bordeaux et al. argue in [9] that while behavioral equivalence is always a sufficient condition for substitutability, it is not necessary.

[9] introduces the idea of *context-dependent substitutability* by stating that given two compatible services Q_1 and Q_2, if another service Q_1' is compatible with Q_2, then Q_1' can substitute for Q_1. Their definition of compatibility implies that Q_1' can substitute for Q_1 if at least one of the following holds: Q_1 and Q_1' are behaviorally equivalent, Q_1' cannot produce any outputs that Q_2 cannot consume and vice versa, or the composition of Q_1' and Q_2 is deadlock-free. This concept represents an important first step in relaxing the behavioral equivalence requirement.

Our approach derives from the work of Pathak et al. in [35], which extends the notion of "context" in context-dependent substitutability to include both the structure of the services in a composition and the specific functionality that each component service provides to the entire composition. In [35], the context-specific substitutability problem is divided into two branches, environment-independent and environment-dependent, based on whether the substitutability condition must apply to all possible compositions (environment-independent) or one particular composition (environment-dependent). To accomplish the environment-dependent analysis, Pathak et al. use a technique called *quotienting* to reduce the functionality that a substitute service is required to satisfy. Consider a simple Web service composition Q that incorporates two services Q_1 and Q_2, and let φ be the property that describes the functionality provided by Q. Suppose that the service Q_1 must be replaced. Quotienting the property φ against the service Q_2, written $(\varphi \diagup Q_2)$, removes the parts of the property φ that are satisfied by the functionality of Q_2 and leaves the parts of φ that must be satisfied by the rest of the composition, i.e., Q_1. The result of this quotienting operation may be used as the substitutability condition for Q_1 with respect to the composition Q. In the quotienting operation, Web services are represented as labeled transition systems and properties are expressed as mu-calculus statements; as a result, environment-dependent substitutability conditions can include detailed requirements that cannot be expressed in other specification languages. A detailed description of the quotienting technique is given in Section 3.5, and an example of the use of quotienting to compute substitutability conditions for Web services appears in Section 3.6.

A drawback of these methods is that they generally assume synchronous communication between all component services; in other words, they assume that each output produced by one component is immediately consumed as an input by another component. While this assumption often simplifies the computation of a substitutability condition, it also significantly limits the extent to which the substitutability condition may be relaxed by requiring that a substitute service must perform certain required actions in exactly the same order as the original service to preserve synchronization of actions. This implicit action-ordering requirement unnecessarily eliminates possible substitute services that accomplish the required tasks in a different order from the original service. An asynchronous communication paradigm, in which each output can be stored until some component requires it, permits further relaxation of the substitutability condition by eliminating this implicit requirement. By considering asynchronous communication in Web service compositions, it is possible to identify additional substitutes for each service, increasing the likelihood that an appropriate substitute can be found.

One model that incorporates asynchronous communication is the conversation model presented by Fu et al. in [11], which focuses on the set of messages generated by the component services (the *conversation* between the components). This model represents each component as a finite state machine with a queue to store unprocessed inputs. Components process input messages and produce output messages independently of other components, so all communication is asynchronous. This model is used by Bultan et al. in [12] to develop the concepts of *synchronizability* (whether a service's conversation set is identical under both synchronous and asynchronous communication) and *realizability* (whether a composition that generates the required conversation set exists). It is shown in [12] that if an asynchronous composition is both synchronizable and realizable, then the asynchronous composition can be safely treated as a synchronous composition for formal verification purposes.

Another Web service model that can account for asynchronous communication is proposed by Kazhamiakin et al. in [24]. In this model, each individual Web service is modeled as a labeled transition system (LTS). A composition of asynchronously communicating Web services is modeled as a composition of the LTSs representing the services, combined with a set of shared

message queues that can store messages for later consumption. The transition function of this composite LTS defines the flow of messages into and out of the queues. Unlike in the conversation model of Fu et al., individual services in this model do not each have their own message queue; instead, the queues exist at the level of the composition. Kazhamiakin et al. use this model in [24] to define a hierarchy of communication models ranging from complete synchronization to unrestricted asynchronous message passing. They also present an algorithm for determining the least general adequate communication model for a given composition, i.e., the model that requires the smallest amount of asynchronous communication to accurately represent the functionality of the composition. They do not, however, use their model to address the problem of Web service substitutability.

The model of asynchronous communication used in our work is inspired by the model used in [12] and is similar to the model used in [24], but the goal of our work is different. We aim to show that any asynchronous composition may be transformed into an equivalent synchronous composition in order to perform formal verification. This will allow us to show, in turn, that there exists some asynchronous composition under which the substitutability condition may be considerably relaxed.

CHAPTER 3. BACKGROUND

This chapter presents the necessary background information for understanding our work on Web service substitutability. The concepts in this chapter are all incorporated into the MoSCoE framework as presented in [35, 36], as our work is meant to complement and extend this existing framework. We begin with a formal definition of labeled transition systems, which we use to represent Web services. A synchronous composition of Web services is then defined as the product of labeled transition systems that model synchronously communicating services. Next, we introduce mu-calculus, the temporal logic that we use to describe properties of Web services. Finally, using the above concepts, we discuss the Web service substitutability problem and explain its solution as proposed by Pathak et al. in [35].

3.1 Labeled Transition Systems

In our work, Web services are represented by Labeled Transition Systems (LTS) [30]. An LTS is a finite state machine with a set of states that are connected by transitions, where each transition is labeled with an action. The formal definition of an LTS, as given in [35], follows:

Definition 1 (Labeled Transition System) *An LTS is defined as* $Q = (S, s_0, A, \Delta)$*, where:*

1. S *is the set of states, representing the configurations of a service.*
2. $s_0 \in S$ *is the start state, representing the initial configuration of a service.*
3. A *is the set of actions of the form* $\{m?, m!, m, \tau\}$*.* $m?$*, denoting an input action, and* $m!$*, denoting an output action, are used by the service to communicate with other services and/or the end-user. An action* m *denotes atomic actions of the service that are observable to the external world, while* τ *represents an internal or unobservable action.*

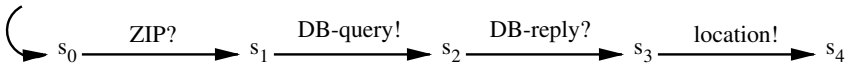


Figure 3.1 Example representation of a Web service as an LTS

4. $\Delta \subseteq S \times A \times S$ *denotes the transition relation.* $(s, a, s') \in \Delta$ *represents that a service in state* s *moves to a state* s' *after performing action* $a \in A$.

From this point forward in the thesis, we will use $s \xrightarrow{a} s'$ for $(s, a, s') \in \Delta$.

Figure 3.1 shows an LTS that corresponds to a simple Web service. The service shown in this figure is designed to receive a ZIP code as input, contact a database, and return the city and state represented by that code as output. The service is initially in the start state s_0 before receiving input. Upon receiving a ZIP code from the user (`ZIP?`), the service moves to state s_1. In state s_1, the service formulates a database query, then moves to state s_2 by sending the query to a database (`DB-query!`). The service waits in state s_2 to receive a response from the database; when a response is received (`DB-reply?`), the service moves to state s_3, in which it parses the response. The service then moves to the final state s_4 by producing an output message (`location!`) that contains the city and state names corresponding to the given ZIP code.

3.2 Web Service Composition

A composition of Web services, as defined in the MoSCoE framework, consists of two or more services that work together to provide some desired functionality. In order for the component services to work together, certain communication links must be defined between the component services in order to ensure that each required input to one service is satisfied by an output from another service or by data received from the user. For example, if component service Q_2 must receive a message of type m as input, then the composition must contain a component service Q_1 that outputs a message of type m; further, a link must be created between the m-output action of Q_1 and the m-input action of Q_2.

Let us formalize this idea by introducing a relation $inv \subseteq A \times A$, where $(a, b) \in inv$ (i.e., a is the inverse action of b) if and only if $a = m?$ and $b = m!$ or vice versa. We will write $inv(a, b)$ to denote $(a, b) \in inv$. This relation, previously defined in [35], captures the idea that if one service performs the input action $a = m?$ and another service performs the output action $b = m!$, then the services communicate by synchronizing on the action pair $(m?, m!)$.

A composite service, or a composition of multiple communicating component services, is then defined as the parallel composition of the LTSs representing the components. In other words, composition of services is performed by establishing synchronous communication links between the components. The formal definition of this operation is given here.

Definition 2 (Synchronous LTS Composition [35]) *Given* $Q_1 = (S_1,\ s_{0,1},\ A_1,\ \Delta_1)$ *and* $Q_2 = (S_2,\ s_{0,2},\ A_2,\ \Delta_2)$*, their composition under a set of "restrictions" R, denoted by* $(Q_1 \ ||\ Q_2)\backslash R$*, is a tuple* $Q = (S, s_0, A, \Delta)$ *where* $S \subseteq S_1 \times S_2$*,* $s_0 = (s_{0,1}, s_{0,2})$*, and* $A \subseteq A_1 \cup A_2 \cup \{\tau\}$*. The transition relation* Δ *is defined as:*

1. $(s_1, s_2) \xrightarrow{a} (t_1, t_2)$ *if* $a \notin R$ *and there exists (i)* $s_1 \xrightarrow{a} t_1$*,* $s_2 = t_2$*, or (ii)* $s_2 \xrightarrow{a} t_2$*,* $s_1 = t_1$

2. $(s_1, s_2) \xrightarrow{\tau} (t_1, t_2)$ *if there exists* $s_1 \xrightarrow{a} t_1$*,* $s_2 \xrightarrow{b} t_2$*,* $\text{inv}(a, b)$*, and* $a, b \in R$

The restriction set $R \subseteq A$ is defined as the set of actions on which Q_1 and Q_2 must make synchronized moves (communicating via the input/output actions) and generate a τ-transition in the composition. For all actions that are not in R, the component services make autonomous moves.

Although this definition provides a simple and effective model of Web service composition, it is worth noting two assumptions made in this definition: that the composition is formed from two services and that the composition's component services communicate synchronously. The first assumption is made to simplify the definition, making it easier to understand. This assumption is easily relaxed, as Definition 2 can be extended to handle more than two services by redefining the components of the composition Q, e.g., by using $S \subseteq S_1 \times \cdots \times S_n$, $s_0 = (s_{01}, \ldots, s_{0n})$, and $A \subseteq A_1 \cup \cdots \cup A_n \cup \{\tau\}$, and redefining Δ accordingly. The second

assumption, that the component services communicate synchronously, is more difficult to relax. Unfortunately, requiring synchronous communication between component services creates problems for two reasons. First, the nature of communications over the Web renders true synchronization costly at best and impossible at worst. Second, and more importantly for our purposes, requiring synchronous communication makes it unnecessarily difficult to identify a group of services that can be successfully composed to provide the required functionality. If a set of interacting services performs the required tasks in a different order than expected, then a satisfactory solution will be wrongly rejected. A representation of Web service composition that allows for asynchronous communication between components is therefore needed. We will propose such a representation in Chapter 4.

3.3 Mu-Calculus

Mu-calculus [17] is an expressive temporal logic that uses explicit least and greatest fixed point operators to represent temporal properties. Because it uses explicit fixed point operators, mu-calculus is more expressive — and therefore more powerful — than more commonly used temporal logics, such as CTL (Computation Tree Logic), CTL* (an extension of CTL), and LTL (Linear Temporal Logic); in fact, any property expressed in these logics can also be expressed in mu-calculus. We use mu-calculus to represent properties of services because the expressivity of mu-calculus makes it flexible enough to represent more types of properties than other temporal logics.

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

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

where $a \in A$, $X \in \mathcal{X}$, and $\sigma \in \{\mu, \nu\}$. The $\langle . \rangle$ and $[.]$ operators are modal operators, which are referred to as diamond and box modalities, respectively. The operator μ is the least fixed point operator, while ν is the greatest fixed point operator. A 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. In this work, we consider only formulas where all variables are bound. We write $def(X) = \sigma X.\varphi$ for $\sigma X.\varphi$.

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

Figure 3.2 Semantics of a mu-calculus formula

The semantics of a mu-calculus formula φ, denoted by $[\![\varphi]\!]_e$, is given in terms of the set of states of an LTS $Q = (S, s_0, A, \Delta)$ that satisfies the formula φ. Figure 3.2 presents the full set of mu-calculus semantics; these rules and their explanations are taken from [35]. In Figure 3.2, the subscript e in $[\![\varphi]\!]_e$ denotes a mapping function of the form $e : \mathcal{X} \rightarrow 2^S$. This function is used to map fixed point variables to sets of states in the LTS.

The propositional constant *tt* (true) is satisfied by all states, while the propositional constant *ff* (false) is not satisfied by any state. The semantics of conjunctive and disjunctive expressions are the intersection and the union of the semantics of the conjuncts and disjuncts, respectively. The diamond modal expression $\langle a \rangle \varphi$ is satisfied by states having at least one a-successor (i.e., a successor state reachable by an a action) that satisfies φ. The box modal expression $[a]\varphi$, which is the dual of $\langle a \rangle \varphi$, is satisfied by the states whose a-successors, if any, all satisfy φ; note that a state with no a-successors trivially satisfies $[a]\varphi$. The semantics of a fixed point variable X is defined by the mapping function e.

Finally, the 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 e such that $e[X \mapsto S'](Y)$ is equal to S' if $X = Y$ and $e(Y)$ otherwise. It can be immediately shown that $f_{X,e} : 2^S \rightarrow 2^S$ is monotonic over the lattice

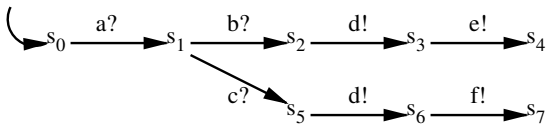


Figure 3.3 LTS for demonstrating the use of mu-calculus

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 the Tarski-Knaster theorem [41], the semantics of the fixed point expression is obtained from $n = |S|$ applications of the function $f_{X,e}(\hat{S})$, where $\hat{S}$ is initially equal to $\emptyset$ (the bottom of the lattice of subsets of S) if computing a least fixed point or S (the top of the lattice) if computing a greatest fixed point.

An LTS $Q = (S, s_0, A, \Delta)$ is said to *satisfy* a mu-calculus formula φ (denoted by $Q \models \varphi$) if and only if $s_0 \in [\![\varphi]\!]_e$. We will use $s_0 \in [\![\varphi]\!]_e$ and $s_0 \models \varphi$ interchangeably. Therefore, if a Web service is represented as an LTS Q and a property that the service must satisfy is expressed as a mu-calculus formula φ, then the service satisfies the required property if and only if $Q \models \varphi$, assuming that the LTS Q accurately represents the service's behavior. This proposition is the basis of the verification technique proposed in [35] and extended in this work.

Example. To demonstrate how the semantics of mu-calculus can be used to determine the set of states that satisfy a given mu-calculus formula, let us consider the LTS Q shown in Figure 3.3 and define the mu-calculus formula

$$\varphi \doteq \langle a? \rangle \mu X.(\langle d! \rangle tt \vee \langle - \rangle X) \tag{3.1}$$

where the "dash" character $(-)$ is a wildcard that stands for any action. In the context of the LTS Q in Figure 3.3, this translates roughly to the following in English:

There is an immediate $a?$-transition, which is followed eventually by a $d!$-transition.

According to the semantics given in Figure 3.2, the first part of φ, $\langle a? \rangle$, is satisfied by all states that have an $a?$-transition to a state that satisfies the remainder of the formula. The only $a?$-transition in Q is from state s_0 to state s_1; therefore, if s_1 satisfies the remaining parts

of φ, then s_0 alone satisfies the entire formula. To determine whether this is the case, the least fixed point $\mu X.(\langle d! \rangle tt \vee \langle - \rangle X)$ must be computed. As before, let $\hat{S} \subseteq S$ be the set of states representing the current status of the fixed point computation. The fixed point function is defined as $f_{X,e}(\hat{S}) = [\![\langle d! \rangle tt \vee \langle - \rangle X]\!]_{e[X \mapsto \hat{S}]}$, meaning "states with an outgoing $d!$-transition to any other state or states with an outgoing transition of any type to a state already in $\hat{S}$." Since we are computing a least fixed point, we must begin with $\hat{S} = \emptyset$. On the first application of the fixed point function, we add to $\hat{S}$ all states that have outgoing $d!$-transitions (s_2 and s_5). On the second application of the function, we add all states that have outgoing transitions to either s_2 or s_5 (only s_1). On the third application, we add all states that have outgoing transitions to s_1 (only s_0). On the fourth application, we add all states that have outgoing transitions to s_0; however, because s_0 is the start state, we cannot add any more states to $\hat{S}$. We have therefore reached a *fixed point*, since further applications of the function cannot change the value of $\hat{S}$. In notation, the fixed point computation proceeds as follows:

$$
\begin{array}{lcll}
f_{X,e}(\emptyset) & & & = \{s_2, s_5\} \\
f^2_{X,e}(\emptyset) & = & f_{X,e}(f_{X,e}(\emptyset)) & = \{s_1, s_2, s_5\} \\
f^3_{X,e}(\emptyset) & = & f_{X,e}(f_{X,e}(f_{X,e}(\emptyset))) & = \{s_0, s_1, s_2, s_5\} \\
f^4_{X,e}(\emptyset) & = & f_{X,e}(f_{X,e}(f_{X,e}(f_{X,e}(\emptyset)))) & = \{s_0, s_1, s_2, s_5\}
\end{array}
$$

So $\{s_0, s_1, s_2, s_5\}$ is the set of states that satisfy the subformula $\mu X.(\langle d! \rangle tt \vee \langle - \rangle X)$. Because s_1 satisfies this subformula and s_0 has an $a?$-transition to s_1, we know that s_0 satisfies Equation 3.1. Using this proposition and the fact that $s_0 \models \varphi$, we conclude that $Q \models \varphi$, i.e., Q provides the required functionality specified by φ.

Let us briefly revisit the Web service composition problem. Given a set of LTS representations of services $\mathcal{Q}$ and a desired functionality φ expressed in mu-calculus logic, the problem of Web service composition is to identify the existence of a composition $Q_1 \,||\, Q_2 \,||\, \cdots \,||\, Q_n \models \varphi$, where for all $i \leq n$, $Q_i \in \mathcal{Q}$. A number of techniques, including [6, 11, 22, 36, 37], have been proposed and developed to address this problem. In this work, we do not focus on the composition problem; instead, we focus on the related problem of Web service substitutability, which will be discussed in the following section.

3.4 The Web Service Substitutability Problem

The problem of Web service substitutability is concisely expressed by Bordeaux et al. in [9] as a question: "when can one service be replaced by another without introducing some flaws into the whole system?" Although the substitutability problem applies to any application of Web services, we will restrict our attention to substitution within the context of Web service compositions; this allows us to characterize Web service substitution in the context of several interacting services while directly addressing the problem of replacing unusable components of Web service compositions. Our view of the Web service substitutability problem, which is derived from the view taken by Pathak et al. in [35], involves determining whether a service Q_1 that is part of a composition with one or more services Q_2 can be replaced by a substitute service Q_1' while preserving the functionality of the original composition. Generalizing this problem to compositions of any size produces the following definition of the general Web service substitutability problem:

Definition 3 (General Service Substitutability Problem [35]) *Let Q be a Web service composition that comprises n component services $Q_1, Q_2, \ldots, Q_n$ and that satisfies some required property φ describing the functionality of the composition. Suppose that one or more components of the composition Q_i (where $i \in 1, 2, \ldots, n$) become unavailable or unusable. Does there exist a service (or composition of services) Q_i' that may be substituted for Q_i such that the new composition Q' resulting from the substitution satisfies the property φ?*

The substitutability problem is usually solved by finding an appropriate *substitutability condition* that will allow the service(s) to be replaced without loss or reduction of functionality. Any service that satisfies the substitutability condition for a service Q_i in a composition Q is then an acceptable substitute for Q_i in Q. A formal definition of this concept is:

Definition 4 (Substitutability Condition) *Let $Q = Q_1 \,||\, \cdots \,||\, Q_n$ be a Web service composition that realizes a desired functionality φ. The* substitutability condition *for a service $Q_i \in Q$ is the property or functionality that a substitute service Q_i' must satisfy or provide in order to preserve the full functionality φ of the composition Q when Q_i' is substituted for Q_i.*

In [35], Pathak et al. divide the substitutability problem into two variants based on the potential substitute service's dependence on its *environment* (i.e., the other services in the composition). Both variants assume that a property φ and services Q_1 and Q_1' are given. The *environment-independent substitutability problem* asks whether Q_1' can substitute for Q_1 in any composition, regardless of the environment of Q_1. The *environment-dependent substitutability problem*, which is a relaxation of the environment-independent problem, asks whether Q_1' can substitute for Q_1 in a particular composition. Defining Q_2 as the environment of a composition (i.e., all components of a composition except for Q_1) yields the following formal definitions for each variant of the problem, taken from [35]:

- Environment-independent: $\forall Q_2 : (Q_1 \mid\mid Q_2 \models \varphi) \stackrel{?}{\Rightarrow} (Q_1' \mid\mid Q_2 \models \varphi)$
- Environment-dependent: $\exists Q_2 : (Q_1 \mid\mid Q_2 \models \varphi) \stackrel{?}{\Rightarrow} (Q_1' \mid\mid Q_2 \models \varphi)$

It follows from these definitions that an environment-independent substitutability condition is generally stricter than the corresponding environment-dependent condition. A service that satisfies the environment-independent version is valuable because it can be used to replace the original component in any composition. However, in any given case, more services will satisfy environment-dependent substitutability, which gives a composition developer flexibility to choose the best substitute service for the situation based on factors besides simple correctness. Even if a substitute service can replace the original service in all compositions, it may not be the best choice for a particular composition. Further, using an environment-dependent substitutability condition may allow the consideration of one or more possible substitute services that would be rejected if an environment-independent condition was used.

Note that both variants of the substitutability problem defined thus far assume that all services within the composition communicate synchronously. We will show in Chapter 4 that this assumption significantly limits the likelihood of finding a suitable substitute service. Nonetheless, for simplicity, we will continue to assume that all components of compositions communicate synchronously until the end of this chapter.

3.5 The Quotienting Technique

The formal analysis necessary to compute a substitutability condition for a component of a Web service composition is done in [35] using a technique called *quotienting*. Quotienting provides a method for reducing the size and complexity of the property that a substitute service must satisfy by removing the portions of the required property that are satisfied by other components of the composition (i.e., by the environment of the service being replaced). The result of the quotienting operation is an environment-dependent substitutability condition for the service in question. In addition to its use in Web service substitutability analysis, quotienting has been used to solve problems in a variety of other settings, including model checking of ring protocols [1], verification of parameterized systems [4], and controller synthesis of discrete event systems [3].

Given that a composition $Q_1 \mid\mid Q_2$ satisfies a desired functionality φ, the quotienting operation aims to obtain the property ψ that Q_2 (the *environment* of Q_1) must satisfy. In terms of the start states $s_{0,1}$ and $s_{0,2}$ of Q_1 and Q_2 respectively, this can be restated as: given $(s_{0,1}, s_{0,2}) \models \varphi$, obtain the property ψ such that $s_{0,2} \models \psi$ must hold. This is realized by defining the quotienting function as follows: $(\varphi \diagup_{T,R} s) : \Phi \times S \times \mathcal{R} \times \mathcal{T} \rightarrow \Phi$, where $\varphi \in \Phi$, $s \in S$ of an LTS Q, $R \in \mathcal{R}$ is the restricted action set (i.e., 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 definition of quotienting. The result of $(\varphi \diagup_{T,R} s)$ is another mu-calculus formula that must be satisfied by the environment state t such that $(s, t) \models \varphi$ under the restriction set R.

Figure 3.4 presents the quotienting function. Rule 1 states that any environment state when composed with s can satisfy *tt*, while Rule 2 states that no environment state can be composed with s to satisfy *ff*. Rules 3 and 4 follow from the fact that the semantics of conjunctive and disjunctive formulas are the 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

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

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

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

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

Figure 3.4 Quotienting rules

s, can satisfy $\langle a \rangle \varphi$. The cases depend on whether the actions from s and t synchronize. The first disjunct is satisfied if some action from state t is responsible to satisfy the diamond modal obligation, i.e., if t has an a action after which it reaches a state t' such that (s, t') satisfies φ. The second disjunct corresponds to the case when $a = \tau$ and the diamond modal obligation on τ is satisfied by a synchronous move from s and t. As a result, if there is a restricted action[1] a from s, then the environment state t must have the action b such that $inv(a, b)$. The third disjunct corresponds to the case where s makes an autonomous move to satisfy the diamond modal obligation. Rule 6 is the dual of Rule 5 and can be similarly explained.

Rule 7 denotes the quotienting of a fixed point formula. It can be proven that given a fixed point formula φ containing sub-formulas of the form $\sigma X.\varphi_x$, the sub-formulas can be quotiented $|S|^{nd}$ times in the worst case [3]. Here nd denotes the nesting depth of the formula under consideration. For each repeated quotienting, a new fixed point formula is generated over a fixed point variable X_i^s, where i denotes the ith time the formula $\sigma X.\varphi_x$ is quotiented using the state s. The tag set T is used to keep track of the repeated quotienting of the same fixed point formula against the same state. In case 2 of Rule 7, $T[X_i^s \backslash X_{i+1}^s]$ denotes the replacement of X_i^s with X_{i+1}^s in the tag set T.

Finally, Rule 8 case 1 applies to the situation where quotienting $\sigma X.\varphi_x$ against s leads to quotienting of X in φ_x against s. The result is X_i^s, the variable resulting from the last quotienting of $\sigma X.\varphi_x$ against s as per Rule 7. Case 2 in Rule 8 considers the quotienting of X against s for the first time. In this case, X is expanded to its definition, $\sigma X.\varphi_x$.

The following theorem asserts the correctness of the quotienting rules.

Theorem 1 (Soundness and Completeness of Quotienting) *Given any two LTSs* $Q_1 = (S_1, s_{01}, A_1, \Delta_1)$ *and* $Q_2 = (S_2, s_{02}, A_2, \Delta_2)$*, a restriction set* R*, and a mu-calculus formula* φ*, the following holds:* $((Q_1 \mid\mid Q_2) \backslash R \models \varphi) \Leftrightarrow (Q_2 \models (\varphi \diagup_{\emptyset,R} Q_1))$.

For a detailed explanation of the rules, their correspondence to the mu-calculus semantics (Figure 3.2), and the proof of the above theorem, refer to [3, 4].

[1] Recall that restricted actions are the ones on which LTSs communicate via synchronization (see Definition 2).

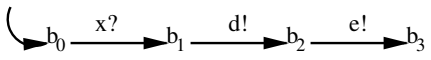


Figure 3.5 Sample service Q_2 for quotienting example

$$
\begin{array}{lcll}
(\varphi \mathbin{/}_{\emptyset,\emptyset} Q_2) & = & (\nu Y.([b?]\mu Z.(\langle e! \rangle tt \vee \langle - \rangle Z) \wedge [-]Y) \mathbin{/}_{\emptyset,\emptyset} b_0) & \\
 & = & \nu Y_1^{b_0}.([b?](\mu Z.(\langle e! \rangle tt \vee \langle - \rangle Z) / b_0) \wedge [-]Y_1^{b_0} \wedge \varphi_{Y_1^{b_1}}) & \text{Rules 7, 3, 6} \\
 & = & \nu Y_1^{b_0}.([b?]\mu Z_1^{b_0}.(\langle e! \rangle tt \vee \langle - \rangle Z_1^{b_0} \vee \varphi_{Z_1^{b_1}}) \wedge [-]Y_1^{b_0} \wedge \varphi_{Y_1^{b_1}}) & \text{Rules 7, 4, 5, 1} \\
 & & & \\
\varphi_{Y_1^{b_1}} & = & \nu Y_1^{b_1}.([b?]\varphi_{Z_1^{b_1}} \wedge [-]Y_1^{b_1} \wedge \varphi_{Y_1^{b_2}}) & \text{Rules 7, 3, 6} \\
\varphi_{Y_1^{b_2}} & = & \nu Y_1^{b_2}.([b?]\varphi_{Z_1^{b_2}} \wedge [-]Y_1^{b_2} \wedge \varphi_{Y_1^{b_3}}) & \text{Rules 7, 3, 6} \\
\varphi_{Y_1^{b_3}} & = & \nu Y_1^{b_3}.([b?](\mu Z.(\langle e! \rangle tt \vee \langle - \rangle Z) / b_3) \wedge [-]Y_1^{b_3}) & \text{Rules 7, 3, 6} \\
 & = & \nu Y_1^{b_3}.([b?]\mu Z_1^{b_3}.(\langle e! \rangle tt \vee \langle - \rangle Z_1^{b_3}) \wedge [-]Y_1^{b_3}) & \text{Rules 7, 4, 5, 1} \\
 & & & \\
\varphi_{Z_1^{b_1}} & = & \mu Z_1^{b_1}.(\langle e! \rangle tt \vee \langle - \rangle Z_1^{b_1} \vee \varphi_{Z_1^{b_2}} & \text{Rules 7, 4, 5, 1} \\
\varphi_{Z_1^{b_2}} & = & \mu Z_1^{b_1}.(\langle e! \rangle tt \vee tt \vee \langle - \rangle Z_1^{b_2} \vee \mu Z_1^{b_3}.(\langle e! \rangle tt \vee \langle - \rangle Z_1^{b_3})) & \text{Rules 7, 4, 5, 1} \\
 & = & \mu Z_1^{b_2}.(tt) & \forall P : tt \vee P = tt
\end{array}
$$

Figure 3.6 Results of quotienting φ (Equation 3.2) by Q_2 (Figure 3.5)

Example. Suppose that the service Q_2, shown in Figure 3.5, provides a portion of the functionality specified by the following mu-calculus formula:

$$\varphi = \nu Y.([b?]\mu Z.(\langle e! \rangle tt \vee \langle - \rangle Z) \wedge [-]Y) \tag{3.2}$$

which means that every $b?$-transition must be followed eventually by an $e!$-transition. We want to know what portion of this property is satisfied by Q_2, so we will quotient φ against the start state of Q_2.

The results of this operation are shown in Figure 3.6. Because Q_2 contains an $e!$-transition, the inner fixed point formula is satisfied if a $b?$ action occurs in the environment of Q_2 before Q_2 reaches state b_2.

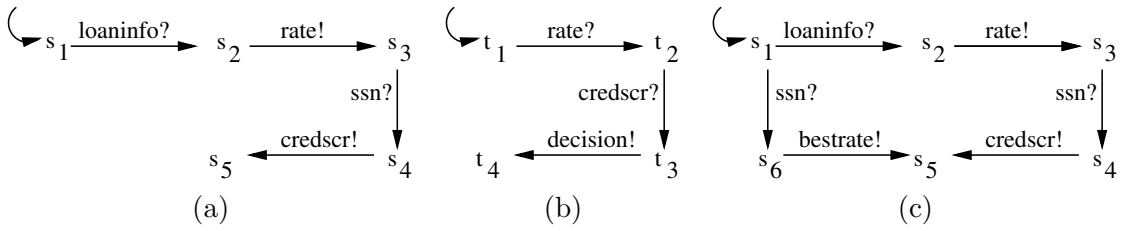


Figure 3.7 LTS representations for (a) Q_1, (b) Q_2, (c) Q_1'

3.6 Applying Quotienting to Determine Web Service Substitutability

The quotienting technique presented in the previous section is used by Pathak et al. in [35] to reduce the Web service substitutability problem to the problem of mu-calculus satisfiability. Recall the two variants of the context-specific substitutability problem defined in Section 3.4, and suppose that the composition $Q_1 \parallel Q_2$ satisfies the functionality or property φ. For the environment-independent substitutability problem, a service Q_1' can replace Q_1 in all possible environments Q_2 where $Q_1 \parallel Q_2 \models \varphi$ if it holds that $(\varphi \mathbin{/_{\emptyset,R}} Q_1) \Rightarrow (\varphi \mathbin{/_{\emptyset,R}} Q_1')$; however, we are not addressing environment-independent substitutability in this work. For the environment-dependent substitutability problem, a service Q_1' can replace Q_1 with respect to a given environment Q_2 if it holds that $Q_1' \models (\varphi \mathbin{/_{\emptyset,R}} Q_2)$. Therefore, the environment-dependent substitutability condition for Q_1 can be determined by simply quotienting the required functionality φ against the environment Q_2. A mu-calculus model checking tool can then be used to determine whether a possible substitute service Q_1' satisfies the substitutability condition.

Let us demonstrate this process with the following example. Suppose that a bank has decided to automate parts of its loan approval process using a Web service composition, which it calls `LoanApproval`. The inputs to the service are the amount and length of the loan (denoted by `loaninfo?`), as well as the Social Security number of the user (denoted by `ssn?`); the output from the service is the approval decision and the monthly payment amount (denoted by `decision!`). `LoanApproval` is composed of two services, `LoanCompute` (Q_1) and `DecisionMaker` (Q_2), shown in Figure 3.7(a, b). `LoanCompute` takes the user inputs and then outputs the interest rate of the loan (denoted by `rate!`) followed by the credit score of the user (denoted by

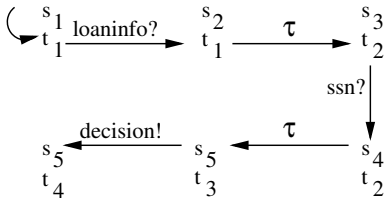


Figure 3.8 The `LoanApproval` service composition $Q_1 \,||\, Q_2$

`credscr!`). `DecisionMaker` takes as input the interest rate (`rate?`) followed by the credit score (`credscr?`) and outputs the decision (`decision!`). Notationally, if the composition is represented by $Q_1 \,||\, Q_2$ (capturing the communication between Q_1 and Q_2) and the functionality of the composition is represented by φ, then the composition satisfies the required functionality, i.e., $Q_1 \,||\, Q_2 \models \varphi$. Figure 3.8 illustrates this composition.

In the event that Q_1 becomes unavailable, it can be replaced with Q_1' (Figure 3.7(c)) such that the composition still realizes the same functionality, i.e., $Q_1' \,||\, Q_2 \models \varphi$. Note that Q_1 and Q_1' are not functionally equivalent ($Q_1 \not\equiv Q_1'$), as is typically required for substitution in the existing literature; however, with respect to the desired functionality of the composition (referred to as the context in [35]), their behavior is equivalent, as their communication patterns with Q_2 are identical. Therefore, the composition $Q_1' \,||\, Q_2$ can be realized and will satisfy φ, meaning that Q_1' is a feasible substitute for Q_1 in `LoanApproval`.

In this example, the property φ that the `LoanApproval` service must satisfy is that after user inputs of `loaninfo?` and `ssn?` are received, eventually an output `decision!` is obtained. This property is represented formally by the following least fixed point mu-calculus formula:

$$\varphi \doteq \langle \texttt{loaninfo?} \rangle \mu X.(\langle \texttt{ssn?} \rangle \mu Y.(\langle \texttt{decision!} \rangle tt \vee \langle \tau \rangle Y) \vee \langle \tau \rangle X)$$

This states that a `loaninfo?` input is followed eventually by an `ssn?` input, which is followed eventually by a `decision!` output. For better readability, in the rest of the paper we will use φ_1 and φ_2 to represent sub-formulas of φ, where:

$$\varphi \doteq \langle \texttt{loaninfo?} \rangle \varphi_1,\ \varphi_1 \doteq \mu X.(\langle \texttt{ssn?} \rangle \varphi_2 \vee \langle \tau \rangle X),\ \varphi_2 \doteq \mu Y.(\langle \texttt{decision!} \rangle tt \vee \langle \tau \rangle Y) \quad (3.3)$$

By inspection of Figure 3.8, it is clear that the composition $Q_1 \,||\, Q_2$ satisfies φ, i.e., $(s_1, t_1) \models \varphi$.

$$
\begin{array}{rcl}
\varphi / t_1 & \doteq & \underline{\langle \texttt{loaninfo?} \rangle (\varphi_1 / t_1)} \\
\varphi_1 / t_1 & \doteq & \mu X_1^{t_1}.(\langle \texttt{ssn?} \rangle (\varphi_2 / t_1) \vee \langle \tau \rangle X_1^{t_1} \vee \underline{\langle \texttt{rate!} \rangle (X / t_2)}) \\
X / t_2 & \doteq & \mu X_1^{t_2}.(\underline{\langle \texttt{ssn?} \rangle (\varphi_2 / t_2)} \vee \langle \tau \rangle X_1^{t_2} \vee \langle \texttt{credscr!} \rangle (X / t_3) \\
X / t_3 & \doteq & \mu X_1^{t_3}.(\langle \texttt{ssn?} \rangle (\varphi_2 / t_3) \vee \langle \tau \rangle X_1^{t_3} \\
\varphi_2 / t_1 & \doteq & \mu Y_1^{t_1}.(\langle \texttt{decision!} \rangle tt \vee \langle \tau \rangle Y_1^{t_1} \vee \langle \texttt{rate!} \rangle (Y / t_2)) \\
Y / t_2 & \doteq & \mu Y_1^{t_2}.(\langle \texttt{decision!} \rangle tt \vee \langle \tau \rangle Y_1^{t_2} \vee \underline{\langle \texttt{credscr!} \rangle (Y / t_3)}) \\
Y / t_3 & \doteq & \underline{\mu Y_1^{t_3}.(tt)} \\
\varphi_2 / t_2 & \equiv & Y / t_2 \\
\varphi_2 / t_3 & \equiv & Y / t_3
\end{array}
$$

Figure 3.9 Result of quotienting $(\varphi /_{\emptyset,R} t_1)$

Having specified the composition $Q_1 \parallel Q_2$ and the property φ that describes its essential functionality, we are now ready to compute the substitutability condition for Q_1. Recall from Theorem 1 that if $Q_1 \parallel Q_2$ satisfies the original property φ, then Q_1 satisfies the quotiented property $(\varphi /_{\emptyset,R} Q_2)$; therefore, if Q_1' also satisfies $(\varphi /_{\emptyset,R} Q_2)$, then $Q_1' \parallel Q_2$ must satisfy φ. We proceed by first quotienting the property φ against the environment process Q_2 and then determining whether Q_1' satisfies the resulting quotiented property.

The process of quotienting $(\varphi /_{\emptyset,R} Q_2)$, where Q_2 is the service LTS (Figure 3.7(b)) and R is the restriction set {`rate!, rate?, credscr!, credscr?`}, begins as follows:

$$(\varphi /_{\emptyset,R} t_1) = \langle \texttt{loaninfo?} \rangle (\mu X_1^{t_1}.\langle \texttt{ssn?} \rangle (\varphi_2 /_{\{X_1^{t_1}\},R} t_1) \vee \langle \texttt{rate!} \rangle (\varphi_1 /_{\{X_1^{t_1}\},R} t_2))$$

where φ_1 and φ_2 are defined in Equation 3.3. Note that `loaninfo?` and `ssn?` are left as the obligation of the environment of Q_2 (i.e., Q_1 or Q_1' must provide these actions), as Q_2 cannot satisfy these modal action obligations. Also note that Q_2 at state t_1 can perform a restricted action `rate?`, which satisfies $\langle \tau \rangle$ in the definition of φ_1 by leaving the obligation of $\langle$`rate!`$\rangle$ for the environment (see Rule 5, second disjunct, Figure 3.4). The result of this quotienting operation is shown in Figure 3.9; the underlined subformulas are satisfied by Q_1'.

To prove that Q_1' can substitute for Q_1, it suffices to show that $Q_1' \models (\varphi /_{\emptyset,R} Q_2)$ holds. Let

s_1 be the start state of Q_1' and t_1 be the start state of Q_2. The proposition is proven as follows:

$$
\begin{aligned}
Q_1' \models (\varphi \diagup Q_2) \quad &\Leftrightarrow \quad s_1 \models (\varphi \diagup t_1) \doteq \langle \texttt{loaninfo?} \rangle (\varphi_1 \diagup t_1) \\
&\Leftrightarrow \quad s_2 \models (\varphi_1 \diagup t_1) \doteq \mu X_1^{t_1}.(\langle \texttt{ssn?} \rangle (\varphi_2 \diagup t_1) \vee \langle \tau \rangle X_1^{t_1} \vee \langle \texttt{rate!} \rangle (X \diagup t_2)) \\
&\Leftrightarrow \quad s_3 \models (X \diagup t_2) \doteq \mu X_1^{t_2}.(\langle \texttt{ssn?} \rangle (\varphi_2 \diagup t_2) \vee \langle \tau \rangle X_1^{t_2} \vee \langle \texttt{credscr!} \rangle (X \diagup t_3)) \\
&\Leftrightarrow \quad s_4 \models (\varphi_2 \diagup t_2) \doteq \mu Y_1^{t_2}.(\langle \texttt{decision!} \rangle tt \vee \langle \tau \rangle Y_1^{t_2} \vee \langle \texttt{credscr!} \rangle (Y \diagup t_3)) \\
&\Leftrightarrow \quad s_5 \models (Y \diagup t_3) \doteq \mu Y_1^{t_3}.(tt) \\
&\Leftrightarrow \quad s_5 \models tt
\end{aligned}
$$

According to the semantics of mu-calculus, every state of every LTS always satisfies tt, implying that $s_1 \models (\varphi \diagup t_1)$; therefore, $Q_1' \models (\varphi \diagup Q_2)$ holds as desired. Thus we have successfully proven that Q_1' can substitute for Q_1 in the composition $Q_1 \,||\, Q_2$ in the event that Q_1 becomes unusable or unavailable. Knowing that a substitute is available for the component Q_1 may help convince the bank to adopt the proposed Web service composition to provide automated loan approval services.

CHAPTER 4. SUBSTITUTABILITY IN ASYNCHRONOUS COMPOSITIONS

In this chapter, we will discuss how the substitutability condition can be relaxed by allowing asynchronous communication between participating services in a composition. The central theme of our technique will rely on formally modeling an asynchronous communication paradigm using synchronous communication of services with appropriate communication buffers. After formalizing the notion of an asynchronous Web service composition and presenting a technique for treating such a composition as if its components communicate synchronously, we will show how the substitutability condition can be further relaxed.

To motivate the contents of this chapter, we proceed by presenting the drawbacks of considering only synchronous communication when obtaining a substitutability condition. Let us revisit the `LoanApproval` composite service introduced in Section 3.6. Suppose that the bank in our example has developed a new version of the `LoanCompute` component service, which we will denote by Q_1''. The bank needs to determine whether Q_1'' can substitute for the original `LoanCompute` component Q_1 within the `LoanApproval` composition. To that end, our objective is to determine whether the proposed substitute service Q_1'' can provide the same functionality as Q_1 within the existing composition $Q_1 \mid\mid Q_2$. The original `LoanCompute` component Q_1 and its previously identified substitute Q_1' are shown in Figures 3.7(a) and 3.7(c), respectively, while Q_1'' is presented in Figure 4.1.

Under the existing setting for substitutability analysis, which requires synchronous communication between component services, it is not possible for Q_1'' to replace Q_1 in the existing composition. However, if support for asynchronous communication can be introduced into the formal Web service composition model, then it may be possible to identify Q_1'' as a valid sub-

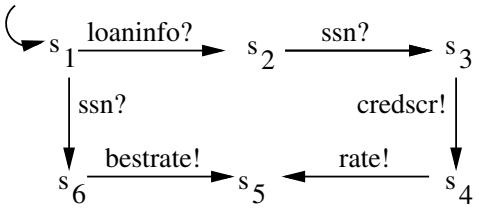


Figure 4.1 LTS representation for Q_1''

stitute for Q_1. Our intent is to provide a formal model of asynchronous communication within Web service compositions and then apply this model to correctly determine a substitutability condition for a component of a Web service composition under an asynchronous communication paradigm.

4.1 Existing Substitutability Analysis Technique Cannot Be Applied

To clearly illustrate that the existing quotienting-based substitutability analysis technique cannot be applied to Web service compositions where the components communicate asynchronously, we will begin by demonstrating informally that Q_1'' can substitute for Q_1 in the existing composition with no loss of functionality. We will then show that naive application of the technique used in Section 3.6 fails to conclude that Q_1'' can substitute for Q_1, after which we will present prior results implying that applying this technique naively to asynchronously communicating Web service compositions is undecidable in general. We will then present our technique in the remaining sections of this chapter.

First, let us observe the functionality of the original `LoanApproval` composite service $Q_1 \parallel Q_2$ that was defined in Section 3.6 and shown in Figure 3.8. In $Q_1 \parallel Q_2$, execution begins when Q_1 receives a `loaninfo` input from the user. The first τ-action represents a `rate` output being produced by Q_1 and immediately consumed by Q_2. At this point, Q_1 receives an `ssn` input from the user; this leads to the second τ-action, in which Q_1 produces a `credscr` output that is immediately consumed by Q_2. Since Q_1 has completed execution, all that remains is for Q_2 to produce its `decision` output for the user.

Now let us determine by inspection whether substituting Q_1'' for Q_1 in this composition preserves the original functionality of $Q_1 \,||\, Q_2$. Q_1'' can receive either a `loaninfo` or `ssn` input from the user as it begins executing. If a `loaninfo` input is received first, then Q_1'' waits to receive an `ssn` input from the user. After receiving both inputs, Q_1'' produces two outputs: first `credscr` and then `rate`. Because Q_2 blocks until it receives a `rate` input, it cannot do anything until Q_1'' has finished; further, the `credscr` output must be stored in order for Q_2 to use it when needed. Clearly, the change in output order from Q_1 to Q_1'' renders synchronous communication between Q_1'' and Q_2 impractical; as a result, $Q_1'' \,||\, Q_2$ cannot provide the functionality of $Q_1 \,||\, Q_2$. However, if we allow asynchronous communication between Q_1'' and Q_2 by providing a common message store accessible to both component services, then the out-of-order outputs produced by Q_1'' can be consumed in the order that Q_2 expects them to be provided. This means that Q_1'' can substitute for Q_1 under an asynchronous communication paradigm with no loss of functionality. The implication of this conclusion is that restricting substitutability analysis to cases where synchronous communication is possible can cause valid substitute services to be incorrectly rejected.

An intuitive possible solution to this problem would be to alter the existing substitutability analysis technique from [35] to assume that all communication between component services will be asynchronous, instead of assuming that all such communication must be synchronous. However, this approach suffers from serious drawbacks, the most important of which is that model checking against a set of LTSs that communicate asynchronously with unbounded queues has been shown in [19] to be undecidable. A variation on this approach involves simulating asynchronous communication by inserting finite-length queues between the component services in the proposed composition. This strategy attempts to find a set of queues of different lengths that will allow a substitution candidate to replace the existing component. Although this avoids the undecidability that comes with using unbounded queues for asynchronous communication, the complexity of this approach means that it is impractical for any substantial applications.

In our work, we have expanded on the idea of using queues or buffers to support asynchronous communication between components. Instead of naively inserting buffers between

services at each interaction point, we create a single buffer process for the entire composition that can handle any type of message that the component services send or receive. This buffer process encapsulates all of the asynchronicity of the communications between component services, because it is always ready to accept an unused output or supply a stored input as needed by other components; however, it is included in a synchronous composition with the original components of the composition. In this way, we transform an asynchronous composition into a synchronous composition that can be analyzed like any other synchronous composition.

4.2 Formalizing Composition of Asynchronous Web Services

Before proceeding further, it is necessary to define a new operator for asynchronous composition of LTSs. The existing LTS composition operator || defined in Definition 2 only allows for synchronous composition. Compositions formed using this operator are allowed to communicate only by synchronizing on actions in the restricted action set R. Consider a composition $Q_s \mid\mid Q_e$ that consists of a service Q_s in composition with several other services that form its environment Q_e, and consider actions a and b, where $a, b \in R$ and $inv(a, b)$. If Q_s reaches a state where it is ready to perform action a, but its environment Q_e has not reached a state where it can perform action b, then Q_s is required to block until Q_e is ready to perform action b. This is true even if a is an output action that is independent of the current state of Q_e. However, under asynchronous communication, the notion of a "restricted action" can be relaxed somewhat. Instead of a strict two-way codependency requiring that an action must never occur except simultaneously with its matching inverse action, a pair of restricted actions now has a weaker one-way dependency, where the output action is completely independent and the input action may occur at any time during or after its corresponding output action. In order to allow this, we must introduce a message store `St` into the composition formalism to keep track of outputs from each participating LTS that can be consumed later. This will allow the strict synchronization requirement to be removed from the logic of the composition formalism as desired.

The formal definition of an asynchronous composition of LTSs (and thus of Web services)

proceeds as follows:

Definition 5 (Asynchronous LTS composition) *Given* $Q_1 = (S_1, s_{0,1}, A_1, \Delta_1)$ *and* $Q_2 = (S_2, s_{0,2}, A_2, \Delta_2)$*, their asynchronous composition under a set of "restrictions"* R*, denoted by* $(Q_1 \,//\, Q_2)\backslash R$*, is a tuple* $Q = (S, s_0, A, \Delta)$*, where* $S \subseteq S_1 \times S_2 \times \mathcal{P}(A)$*,* $s_0 = (s_{0,1}, s_{0,2}, \emptyset)$*, and* $A \subseteq A_1 \cup A_2 \cup \{\tau\}$*. The asynchronous transition relation is* $\Delta \subseteq S \times \mathcal{P}(A) \times A \times S \times \mathcal{P}(A)$*, where* $\mathcal{P}(A)$ *denotes the powerset of* A*.* Δ *is defined as follows:*

1. ***Autonomous move:*** $(s_1, s_2, \mathtt{St}) \xrightarrow{a} (t_1, t_2, \mathtt{St}') \in \Delta$ *if* $a \notin R$ *and there exists*
   (i) $s_1 \xrightarrow{a} t_1 \in \Delta_1$*,* $s_2 = t_2$*, and* $\mathtt{St}' = \mathtt{St}$*; or (ii)* $s_2 \xrightarrow{a} t_2 \in \Delta_2$*,* $s_1 = t_1$*, and* $\mathtt{St}' = \mathtt{St}$*.*

2. ***Asynchronous output:*** $(s_1, s_2, \mathtt{St}) \xrightarrow{\tau} (t_1, t_2, \mathtt{St}') \in \Delta$ *if there exists*
   (i) $s_1 \xrightarrow{a!} t_1 \in \Delta_1$*,* $s_2 = t_2$*,* $a! \in R$*, and* $\mathtt{St}' = \mathtt{St} \cup \{a!\}$*; or*
   (ii) $s_2 \xrightarrow{a!} t_2 \in \Delta_2$*,* $s_1 = t_1$*,* $a! \in R$*, and* $\mathtt{St}' = \mathtt{St} \cup \{a!\}$*.*

3. ***Asynchronous input:*** $(s_1, s_2, \mathtt{St}) \xrightarrow{\tau} (t_1, t_2, \mathtt{St}') \in \Delta$ *if there exists*
   (i) $s_1 \xrightarrow{a?} t_1 \in \Delta_1$*,* $s_2 = t_2$*,* $a? \in R$*,* $a! \in \mathtt{St}$*, and* $\mathtt{St}' = \mathtt{St}\backslash\{a!\}$*; or*
   (ii) $s_2 \xrightarrow{a?} t_2 \in \Delta_2$*,* $s_1 = t_1$*,* $a! \in R$*,* $a! \in \mathtt{St}$*, and* $\mathtt{St}' = \mathtt{St}\backslash\{a!\}$*.*

We observe that Definitions 2 and 5 imply the following proposition.

Proposition 1 *Given any set of* n *services* $Q_1, Q_2, \ldots, Q_n$ *and any property* φ*, if the synchronous composition* $Q_1 \,||\, Q_2 \,||\, \cdots \,||\, Q_n$ *satisfies* φ*, then the asynchronous composition* $Q_1 \,//\, Q_2 \,//\, \cdots \,//\, Q_n$ *also satisfies* φ*.*

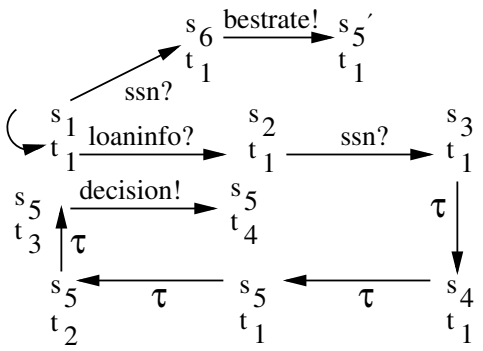
The asynchronous composition operator // defined in Definition 5 is essentially an extension of the synchronous composition operator defined in Definition 2 that provides the additional theoretical machinery for handling asynchronous communication between services. The bulk of the increased complexity results from the introduction of a message store $\mathtt{St} \subseteq \mathcal{P}(A)$. This store holds output messages from any component and provides these stored messages to any component that requires them. Because the contents of the store St may change during any given transition, each state in the composite LTS must include the current contents of the store St at that state.

The asynchronous transition relation Δ encompasses three conditions. The first condition describes an *autonomous move* in which one component LTS makes a move that does not change the contents of the store `St`. This condition is essentially the same as for an autonomous move in a synchronous composition (Condition 1 in Definition 2). The second condition represents an *asynchronous output*, where one LTS makes a move on an output action in R, creating an output message that is not consumed by a corresponding input action in any other LTS. As a result, the output action is added to the store `St` for future consumption. Finally, the third condition corresponds to an *asynchronous input*, where a component LTS performs an input action in R by consuming the appropriate output from the store `St`. An input action for each participating LTS is allowed only when the requested message is available in the store or from the user; if the appropriate message is not available, the LTS must block until it becomes available. Output actions may be performed by any component LTS at any time. Note that none of these conditions describe strictly synchronized communication between component LTSs. While synchronous communication is allowed under this model, synchronization cannot be directly represented. Instead, each synchronized move can be represented as an asynchronous output followed immediately by its corresponding asynchronous input.

Figure 4.2 presents the asynchronous composition of Q_1'' and Q_2 with the restriction set $R = \{$`rate!, rate?, credscr!, credscr?`$\}$. In Q_1'' // Q_2, the store `St` is empty at the start state (s_1, t_1) and remains empty until the τ-transition to (s_4, t_1). As `credscr!` is produced by Q_1'' but not consumed by Q_2 in this transition, `credscr!` is added to `St` at state (s_4, t_1); likewise, `rate!` is added to `St` during the τ-transition to (s_5, t_1). The stored outputs are consumed by Q_2 in the remaining τ-transitions, leaving `St` empty at states (s_5, t_3) and (s_5, t_4).

4.3 Handling Asynchronous Behavior by Using a Buffer Process

The core of our technique for allowing asynchronous behavior in substitutability analysis depends on obtaining a buffer process that is composed with the proposed composition. The purpose of the buffer process in our approach is to provide the facilities required to handle asynchronous communication between the components of a composition by synchronizing with



State	Contents of `St`
(s_1, t_1)	{}
(s_2, t_1)	{}
(s_3, t_1)	{}
(s_4, t_1)	{`credscr!`}
(s_5, t_1)	{`rate!, credscr!`}
(s_5, t_2)	{`rate!`}
(s_5, t_3)	{}
(s_5, t_4)	{}
(s_6, t_1)	{}
$(s_5, t_1)'$	{}

Figure 4.2 The asynchronous composition $Q_1'' \,//\, Q_2$

them as needed to store each component's unused outputs until they are needed as inputs to another component. It does not generate any actions by itself; rather, it exists only to act as an intermediary between the components. We will first provide a formal definition for such a buffer process and then present an algorithm for efficiently constructing an appropriate buffer process for a given asynchronous composition of Web services. After this, we will demonstrate the creation of a buffer process to accomplish an asynchronous composition of Q_1'' and Q_2 in the example from the beginning of this chapter.

We define a buffer process for a given asynchronous composition of LTSs as follows:

Definition 6 (Buffer Process) *Given an asynchronous composition* $(Q_1 \,//\, Q_2)\backslash R$*, where* $Q_i = (S_i, s_{0,i}, A_i, \Delta_i)$ *and* $i \in \{1, 2\}$*, the corresponding buffer process is defined as* $Q_{B12} = ||\{Q_B^a \mid a! \in A_i \cap R\}$*, where each* $Q_B^a = (\{q_0^a, q_1^a, q_2^a\}, q_0^a, \{a!, a?\}, \Delta_B^a)$ *is an LTS such that* $\Delta_B^a = \{q_0^a \xrightarrow{a?} q_1^a, q_1^a \xrightarrow{a?} q_1^a, q_1^a \xrightarrow{a!} q_2^a\}$.

The buffer process Q_{B12} is a synchronous composition of $|R|/2$ buffer LTSs of the form Q_B^a — one such LTS for each input/output action pair in the restriction set R. Each buffer LTS Q_B^a has three states and three transitions (one of which is a "self-loop" transition that does not result in a change in state), and each Q_B^a is capable of consuming an output from any LTS and providing input to another. The buffer process Q_{B12} is simply the synchronous composition of all of the Q_B^as that have been created. Note that the composition Q_{B12} does not have any restriction set, i.e., the participating Q_B^as are not capable of communicating with each other.

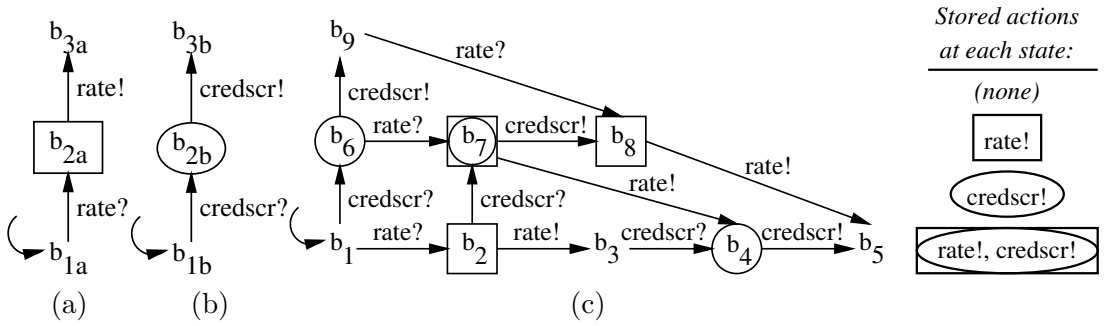


Figure 4.3 Buffer processes (a) $Q_B^{\texttt{rate}}$, (b) $Q_B^{\texttt{credscr}}$, (c) Q_{B12} for our example

Figure 4.3 presents the simple buffers and the resulting buffer process for the asynchronous composition in Figure 4.2. The self-loop transitions are not illustrated in this figure; instead, shapes are used to denote the actions for which a self-loop transition is available at each state. For example, states b_{2a} and b_{2b} have self-loops on `rate?` and `credscr?` actions, respectively; similarly, state b_7 has self-loops on both `rate?` and `credscr?` actions. Note that a path exists for every permitted ordering of the actions.

A buffer process created according to Definition 6 must have exactly $3^{|R|/2}$ states, because the set of states of the buffer process is the Cartesian product of the sets of states of the $|R|/2$ buffer LTSs. We have determined that a buffer process must also have exactly $(|R|/2)\left(3^{|R|/2}\right)$ transitions. Because the set of actions for the buffer process is comparatively small ($|R|$) and therefore occupies relatively little space, the space required to store the buffer process is $O\left(|R|3^{|R|}\right)$.

Before presenting our algorithm for creating a buffer process, it is necessary to present an alternative definition of a restriction set that will be used in the algorithm. It can be observed that the actions in the restriction set R generally occur in pairs, where one action in each pair must always be synchronized with the other action in the same pair. This insight can be formalized by defining a pairwise restriction set as follows:

Definition 7 (Pairwise Restriction Set) *Let* $Q_1 = (S_1, s_{0,1}, A_1, \Delta_1)$ *and* $Q_2 = (S_2, s_{0,2}, A_2, \Delta_2)$ *be any two LTSs, and let* $Q_1 \,||\, Q_2$ *be the synchronous composition of* Q_1 *and* Q_2*. The*

pairwise restriction set $R_P \subseteq A_1 \times A_2$ *of* $Q_1 \mid\mid Q_2$ *is the set of pairs of actions on which* Q_1 *and* Q_2 *must make synchronized moves and generate a* τ*-transition in the composition* $Q_1 \mid\mid Q_2$*. In notation:*

$$R_P = \{(a, b) : a \in A_1, b \in A_2, \text{inv}(a, b)\}$$

where the relation inv *is as defined in Section 3.1.*

We now present Algorithm 1, which creates a buffer process as defined in Definition 6 corresponding to a given *pairwise* restriction set. To understand the underlying premise of this algorithm, recall that each pair of actions $(a?, a!)$ in a pairwise restriction set R_P gives rise to a corresponding buffer Q_B^a having three states (Q_0^a, Q_1^a, and Q_2^a), two actions ($a?$ and $a!$), and three transitions ($Q_0^a \xrightarrow{a?} Q_1^a$, $Q_1^a \xrightarrow{a?} Q_1^a$, and $Q_1^a \xrightarrow{a!} Q_2^a$). Let us place Q_0^a, Q_1^a, and Q_2^a at coordinates 0, 1, and 2 on the number line, respectively; then the transition $Q_0^a \xrightarrow{a?} Q_1^a$ forms the line segment $(0, 1)$, the transition $Q_1^a \xrightarrow{a?} Q_1^a$ forms a self-loop on coordinate 1, and the transition $Q_1^a \xrightarrow{a!} Q_2^a$ forms the line segment $(1, 2)$. If the buffer LTS for each action pair in R_P is placed in its own dimension in space, then the LTS corresponding to the composition of these $|R_P|$ buffers can be viewed as a set of $3^{|R_P|}$ state points in $|R_P|$-space that are connected by transition arcs such that each state point has a transition to a successor state point along each axis where the coordinate of that state point is either 0 or 1. For example, if $|R_P| = 3$, the start state at $(0, 0, 0)$ has three outgoing transitions to states at $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$; the state at $(1, 0, 1)$ has three outgoing transitions to states at $(2, 0, 1)$, $(1, 1, 1)$, and $(1, 0, 2)$; the state at $(1, 1, 2)$ has two outgoing transitions to states at $(2, 1, 2)$ and $(1, 2, 2)$; and the state at $(2, 2, 1)$ has one outgoing transition to the final state at $(2, 2, 2)$. Figure 4.3(c) illustrates an example where $|R_P| = 2$. In addition, each state point has one transition to itself for each axis where the coordinate of that state point is 1. This accounts for the fact that if an action of type a is already stored in the store `St`, then further actions of type a are accepted but not added to the store (i.e., they are lost) until the stored a-message is consumed.

This conception of the composition's structure suggests an intuitive method for composing the buffer process:

1. Create $3^{|R_P|}$ states and map each state to an element of $\prod_{|R_P|} \{0, 1, 2\}$.

2. For each state s, beginning with $(0, \ldots, 0, 0)$, then $(0, \ldots, 0, 1)$, then $(0, \ldots, 0, 2)$, then $(0, \ldots, 1, 0)$, and so on until $(2, \ldots, 2, 2)$, do the following for each axis a:

   (a) If the position of state s on axis a is 0, then create a transition $s \xrightarrow{a?} s'$, where state s' has the same coordinates as s except its position on axis a is 1.

   (b) If the position of state s on axis a is 1, then create a self-loop transition $s \xrightarrow{a?} s$ and a transition $s \xrightarrow{a!} s'$, where state s' has the same coordinates as s except its position on axis a is 2.

   (c) If the position of state s on axis a is 2, then do not create a transition.

Algorithm 1 follows this pattern to generate a complete buffer process from a given pairwise restriction set R_P.

4.4 Combining a Buffer Process and Quotienting to Solve the Problem

Our main objective in this work has been to produce a general solution for the *asynchronous Web service substitutability* problem, or the problem of finding a condition that must be satisfied by a service in order for that service to substitute for a particular component of an existing Web service composition in which the services communicate *asynchronously* without loss of functionality. By combining the formal definition of asynchronous composition given in Definition 5, the algorithm for creating a buffer process that corresponds to a given asynchronous composition presented in Section 4.3, and the application of the quotienting technique from [35] that was presented in Section 3.6, we obtain a general solution for the asynchronous Web service substitutability problem. In this section, we present that solution, prove its correctness, and discuss its complexity.

4.4.1 Solution and Proof of Correctness

We begin the presentation of our solution to the asynchronous substitutability problem by proving that an asynchronous Web service composition and its corresponding buffered synchronous composition created according to the technique described in this chapter are

Algorithm 1 Create a buffer process from a pairwise restriction set

Input: R_P: a pairwise restriction set

Output: $B = (S_B, s_{0,B}, A_B, \Delta_B)$: an LTS for the buffer service corresponding to R_P

procedure BUFFERGEN(R_P)

if $R_P = \emptyset$ **then** ▷ create a trivial buffer

$S_B \leftarrow \{b\}$, $s_{0,B} \leftarrow b$, $A_B \leftarrow \{no\_action\}$, $\Delta_B \leftarrow \{b \xrightarrow{no\_action} b\}$

else ▷ R_P contains at least one restricted action pair

$A_B \leftarrow R_P$

let $actNames = \{a.name : a \in R_P\}$ ▷ the names, without types, of the actions in R_P

let $r = |actNames|$ ▷ the number of matched action pairs in R_P

let $stCtr \in \{0,1,2\}^r$ ▷ array representing the state of each component buffer Q_B^a

$S_B \leftarrow \{b_0, b_1, \ldots, b_{3^r-2}, b_{3^r-1}\}$

define $stLocnMap : S_B \to \{0,1,2\}^r$ such that:

$$b_0 \mapsto (0,\ldots,0,0) \quad b_2 \mapsto (0,\ldots,0,2) \quad \ldots \quad b_{3^r-2} \mapsto (2,\ldots,2,1)$$
$$b_1 \mapsto (0,\ldots,0,1) \quad b_3 \mapsto (0,\ldots,1,0) \qquad\quad b_{3^r-1} \mapsto (2,\ldots,2,2)$$

$stCtr \leftarrow (0,\ldots,0)$

for $i \leftarrow (r-1)$ to 0 **do** ▷ for each value of $stCtr$, in order from $(0,\ldots,0)$ to $(2,\ldots,2)$

if $i = r-1$ **then** ▷ if on last digit (axis) of $stCtr$

while $stCtr[i] \leq 2$ **do**

$start \leftarrow stLocnMap^{-1}(stCtr)$ ▷ choose state mapped to $stCtr$ as transition start

for $j \leftarrow 0$ to $(r-1)$ **do**

if $stCtr[j] < 2$ **then**

$stCtr[j] \leftarrow stCtr[j] + 1$ ▷ find $start$'s successor along the jth axis

$end \leftarrow stLocnMap^{-1}(stCtr)$ ▷ choose this state as transition end

$stCtr[j] \leftarrow stCtr[j] - 1$ ▷ restore the value of $stCtr$

$a.name \leftarrow actNames[j]$ ▷ action name matches jth element of $stCtr$

if $stCtr[j] = 0$ **then**

$\Delta_B \leftarrow \Delta_B \cup \{start \xrightarrow{a?} end\}$

else if $stCtr[j] = 1$ **then**

$\Delta_B \leftarrow \Delta_B \cup \{start \xrightarrow{a?} start, start \xrightarrow{a!} end\}$

$stCtr[i] \leftarrow stCtr[i] + 1$ ▷ move to the next value of $stCtr$

else ▷ if on any other digit (axis) of $stCtr$

if $stCtr[i] < 2$ **then** ▷ if current digit < 2

$stCtr[i] \leftarrow stCtr[i] + 1$ ▷ increment current digit

while $i < r-1$ **do** ▷ reset all previously exhausted digits to zero

$i \leftarrow i + 1$

$stCtr[i] \leftarrow 0$

$i \leftarrow i + 1$ ▷ needed to make **for** loop work correctly

$s_{0,B} \leftarrow b_0$

return B

equivalent. Using this result, we will further show that the condition for substitutability in the asynchronous setting can be obtained from the substitutability condition in the corresponding synchronous setting. It is then possible to perform formal verification of a buffered synchronous composition against this asynchronous substitutability condition, as desired.

Theorem 2 (Composition Equivalence) *Let Q_1 and Q_2 denote two LTSs that represent Web services, let Q_{B12} denote the buffer process generated from $(Q_1 \,//\, Q_2)\backslash R$ as described in Definition 6, and let φ denote a property satisfied by a given composition. The transition relation of the asynchronous composition $(Q_1 \,//\, Q_2)\backslash R$ is bisimulation equivalent to the transition relation of the buffered synchronous composition $(Q_1 \,||\, Q_2 \,||\, Q_{B12})\backslash R$. In notation:*

$$(Q_1 \,||\, Q_2 \,||\, Q_{B12})\backslash R \models \varphi \Leftrightarrow (Q_1 \,//\, Q_2)\backslash R \models \varphi$$

Proof. Given an action $a \in A$, let $\xrightarrow{a}_{||}$ denote a synchronous a-transition, and let $\xrightarrow{a}_{//}$ denote an asynchronous a-transition. Let Δ_1, Δ_2, and Δ_{B12} denote the transition relations of Q_1, Q_2, and Q_{B12}, respectively. We will prove Theorem 2 by showing that $\forall s_1 \in S_1$, $\forall s_2 \in S_2$, $\forall \mathtt{St}, \mathtt{St}' \subseteq \mathcal{P}(A)$, $\forall s_b, s_b' \in S_{B12}$, and $\forall a \in A$, there exists a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{a}_{//} (s_1', s_2', \mathtt{St}')$ in the transition relation of $(Q_1 \,//\, Q_2)\backslash R$ if and only if there exists a transition $(s_1, s_2, s_b) \xrightarrow{a}_{||} (s_1', s_2', s_b')$ in the transition relation of $(Q_1 \,||\, Q_2 \,||\, Q_{B12})\backslash R$. The transition relation of each composition is partitioned into communication moves (i.e., those transitions labeled with a τ-action) and autonomous moves (i.e., all other transitions); each type of move will be addressed separately.

Case 1: Autonomous Move. This is the case in which one component of a composition moves autonomously on an unrestricted action, i.e., any action that is not in the restricted action set R. Autonomous moves are described in rule 2 of the synchronous transition relation defined in Definition 2 and in rule 1 of the asynchronous transition relation defined in Definition 5.

Suppose that $(Q_1 \,||\, Q_2 \,||\, Q_{B12})\backslash R$ contains a transition $(s_1, s_2, s_b) \xrightarrow{a}_{||} (s_1', s_2, s_b)$, such that $a \notin R$ and $s_1 \xrightarrow{a} s_1' \in \Delta_1$. Then by Definition 5, $(Q_1 \,//\, Q_2)\backslash R$ contains a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{a}_{//} (s_1', s_2, \mathtt{St})$, where the contents of $\mathtt{St} \subseteq \mathcal{P}(A)$ reflect the current

state of the buffer process s_b. Conversely, suppose that $(Q_1 \mathbin{//} Q_2)\backslash R$ contains a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{a}_{//} (s_1', s_2, \mathtt{St})$, where $a \notin R$, $\mathtt{St} \subseteq \mathcal{P}(A)$, and $s_1 \xrightarrow{a} s_1' \in \Delta_1$. By Definition 2, $(Q_1 \parallel Q_2 \parallel Q_{B12})\backslash R$ contains a transition $(s_1, s_2, s_b) \xrightarrow{a}_{||} (s_1', s_2, s_b)$, where the state of the buffer process s_b corresponds to the contents of the store $\mathtt{St}$. Similar results occur if the autonomous move occurs in Q_2 instead of in Q_1. Note that the buffer process state s_b and the contents of the store $\mathtt{St}$ have no effect on the relevant transitions, because autonomous moves are always independent of the buffer process or the store, respectively.

Case 2: Communication Move. This is the case in which two components of a composition make a common move, which is illustrated in the LTS diagram as a τ action. Communication moves are described in rule 1 of the synchronous transition relation defined in Definition 2 and in rules 2 and 3 of the asynchronous transition relation defined in Definition 5.

Suppose that $(Q_1 \parallel Q_2 \parallel Q_{B12})\backslash R$ contains a transition $(s_1, s_2, s_b) \xrightarrow{\tau}_{||} (s_1', s_2', s_b')$. This transition represents an input or output action generated by one component service that is immediately consumed by either another component service or the buffer process. Let $(a?, a!) \in R_P$. Suppose further that $s_1 \xrightarrow{a?} s_1' \in \Delta_1$, $s_2 = s_2'$, and $s_b \xrightarrow{a!} s_b' \in \Delta_{B12}$; then by Rule 3 of Definition 5, $(Q_1 \mathbin{//} Q_2)\backslash R$ contains a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{\tau}_{//} (s_1', s_2', \mathtt{St} \setminus \{a!\})$. Alternatively, if $s_1 \xrightarrow{a!} s_1' \in \Delta_1$, $s_2 = s_2'$, and $s_b \xrightarrow{a?} s_b' \in \Delta_{B12}$, then by Rule 2 of Definition 5, $(Q_1 \mathbin{//} Q_2)\backslash R$ contains a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{\tau}_{//} (s_1', s_2', \mathtt{St} \cup \{a!\})$. Because the buffered synchronous composition $(Q_1 \parallel Q_2 \parallel Q_{B12})\backslash R$ is derived from the corresponding asynchronous composition $(Q_1 \mathbin{//} Q_2)\backslash R$, the component services never directly synchronize with each other; rather, the buffer process Q_{B12} always mediates the communication.

Now suppose that $(Q_1 \mathbin{//} Q_2)\backslash R$ contains a transition $(s_1, s_2, \mathtt{St}) \xrightarrow{\tau}_{//} (s_1', s_2', \mathtt{St}')$, where $\mathtt{St} \neq \mathtt{St}'$ and either $s_1 \neq s_1'$ or $s_2 \neq s_2'$. This transition can exist for two reasons:

1. The move results from an $a?$ input action from either s_1 or s_2. This implies that in $(Q_1 \parallel Q_2 \parallel Q_{B12})\backslash R$, there is a τ-transition from the equivalent state triple (s_1, s_2, s_b) resulting from an $a?$ action for which the matching $a!$ action is produced immediately by either a component service or the buffer process.

2. The move results from an $a!$ output action from either s_1 or s_2. This implies that in $(Q_1 \mid\mid Q_2 \mid\mid Q_{B12})\backslash R$, there is a τ-transition from the equivalent state triple (s_1, s_2, s_b) resulting from an $a!$ action that is consumed immediately by either a component service or the buffer process.

In either case, the resulting transition is $(s_1, s_2, s_b) \xrightarrow{\tau}_{||} (s_1', s_2', s_b')$, where s_b is the state of the buffer process that corresponds to the store's original contents and s_b' is the buffer process state that matches the updated contents of the store.

Thus we have shown that each transition in the transition relation of the asynchronous composition $(Q_1 \mathbin{//} Q_2)\backslash R$ implies the existence of an equivalent transition in the transition relation of the corresponding buffered synchronous composition $(Q_1 \mid\mid Q_2 \mid\mid Q_{B12})\backslash R$, and each transition in the transition relation of $(Q_1 \mid\mid Q_2 \mid\mid Q_{B12})\backslash R$ implies the existence of an equivalent transition or pair of transitions in the transition relation of $(Q_1 \mathbin{//} Q_2)\backslash R$. This proves that $(Q_1 \mid\mid Q_2 \mid\mid Q_{B12})\backslash R$ and $(Q_1 \mathbin{//} Q_2)\backslash R$ are bisimulation equivalent, as desired.

□

It follows directly from Theorem 2 that any property φ that is satisfied by an asynchronous composition is also satisfied by its corresponding buffered synchronous composition. Formally:

$$\forall \varphi : (Q_1 \mathbin{//} Q_2)\backslash R \models \varphi \Leftrightarrow ((Q_1 \mid\mid Q_2) \mid\mid Q_{B12})\backslash R \models \varphi$$

We are now ready to formalize the general substitutability condition for asynchronous compositions of Web services in the following theorem:

Theorem 3 (Substitutability Condition) *Given an LTS composition, either* $(Q_1 \mid\mid Q_2)\backslash R$ *or* $(Q_1 \mathbin{//} Q_2)\backslash R$*, which satisfies* φ*,* Q_1 *can be substituted by* Q_1'' *in an asynchronous composition with* Q_2 *if and only if* $Q_1'' \models ((\varphi \diagup_{\emptyset,R} Q_{B12}) \diagup_{\emptyset,\emptyset} Q_2)$*.*

Proof. The proof of the theorem proceeds as follows:

$$\begin{aligned} Q_1'' \models ((\varphi \diagup_{\emptyset,R} Q_{B12}) \diagup_{\emptyset,\emptyset} Q_2) &\Leftrightarrow (Q_1'' \mid\mid Q_2) \models (\varphi \diagup_{\emptyset,R} Q_{B12}) && \text{[Theorem 1]} \\ &\Leftrightarrow ((Q_1'' \mid\mid Q_2) \mid\mid Q_{B12}) \backslash R \models \varphi && \text{[Theorem 1]} \\ &\Leftrightarrow (Q_1'' \mathbin{//} Q_2)\backslash R \models \varphi && \text{[Theorem 2]} \end{aligned}$$

□

4.4.2 Complexity of the Solution

The bulk of the complexity of our solution derives from the processes of creating the buffer process that is necessary for transforming an asynchronous composition into a synchronous composition, quotienting the required property against the buffer process, further quotienting this intermediate result against the equivalent synchronous composition without the component service being replaced, and using model checking to verify whether a substitute service satisfies the resulting substitutability condition. We do not consider the complexity of creating an asynchronous Web service composition in our analysis, as our work does not directly address this topic.

The time complexity of Algorithm 1 depends primarily on the number of action pairs in the restriction set. The algorithm creates the $3^{|R_P|}$ required states, maps each state to an $|R_P|$-tuple of coordinates, and then iterates through the states in order. At each state, each of the state's $|R_P|$ coordinates are checked and all required self-loop transitions and transitions to successor states are created; this occurs during the **if** block within the outer loop. In addition, the current set of coordinates $stCtr$ that is used to create the correct number of transitions from each state must be updated $3^{(|R_P|-1)}$ times; this is done in the **else if** block within the outer loop. The time complexity of Algorithm 1 is therefore $O\left(|R_P|3^{|R_P|}\right)$.

The complexity of verifying whether a service can substitute for one of the constituents in a synchronous composition of Web services depends on the complexity of quotienting and the complexity of model checking mu-calculus formulas. The complexity of quotienting and the size of the formula generated by quotienting have been shown in [3] to be $O(|\varphi| \times |S|^{nd} \times B)$, where $|\varphi|$ is the size of the formula, $|S|$ is the number of states in the LTS used in quotienting, nd is the nesting depth of φ, and B is the maximum branching factor of any state in the LTS. The nesting depth of the generated formula is $O(|S|^{nd})$. The complexity of model checking mu-calculus formulas is $O(|S| \times |\varphi|^{ad})$, where ad denotes the alternation depth of φ. The alternation depth of a formula is the number of nestings of alternating fixed point sub-formulas. Therefore, the worst case complexity for verifying the substitutability of a service Q_1 in a composition of Q_1 with Q_2 in an asynchronous setting is exponential in the number of states in Q_2 and the

nesting depth of the formula (composition property), and it is linear in the number of states in the buffer process.

CHAPTER 5. IMPLEMENTATION

The implementation portion of our work consists of several extensions and modifications to the MoSCoE [36] framework for Web service composition. A number of changes were made to the core data structures of the existing MoSCoE implementation to remove unnecessary or currently unsupported portions and improve the maintainability and extensibility of the remaining parts of the framework. Support for XML-based input and output was also added to the MoSCoE framework as part of the implementation process, including newly developed XML representations for mu-calculus formulas and labeled transition systems. In addition to these changes, the MoSCoE framework was extended and complemented by creating new tools for analyzing the substitutability of Web services in both synchronous and asynchronous settings. These new tools provide support for performing the quotienting operation described in Section 3.5 and for constructing a buffer process according to the technique given in Section 4.3.

Our implementation is written in Java using the Eclipse [16] integrated development environment. The Java 1.6 runtime environment is required to run our implementation because our XML conversion module uses the Streaming API for XML (StAX) [39], which is not natively supported in earlier versions of the Java runtime environment. We chose to use StAX to implement the necessary XML handling functionality because the API is easy to learn and use and because it does not require an XML Schema document to verify the correctness of each XML document. Because of time constraints, the tool set currently does not have a graphical user interface (GUI) available. Development of a GUI will be a high priority for future work.

This chapter begins with a brief overview of the MoSCoE framework and a description of the modifications that we have made to that framework. After presenting our changes to MoSCoE, we describe our tools for performing quotienting and creating buffer processes.

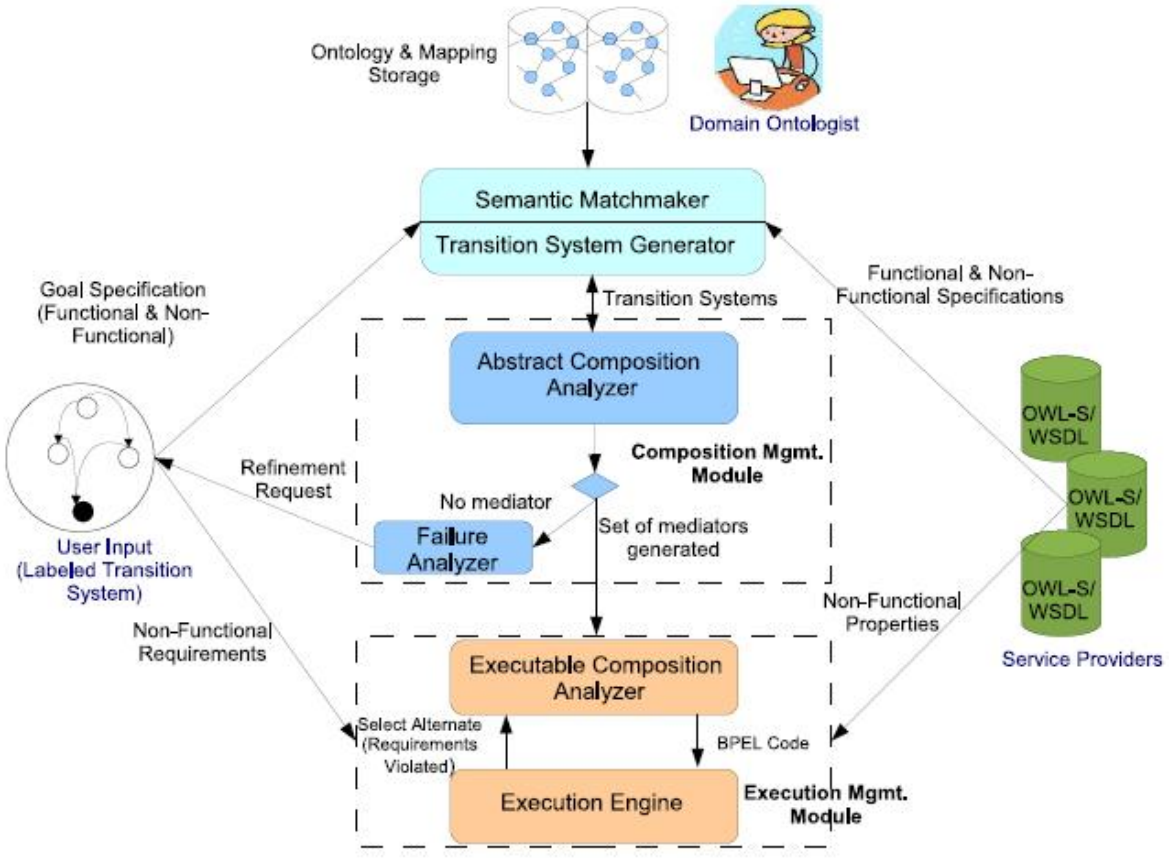


Figure 5.1 MoSCoE architectural diagram [34]

5.1 MoSCoE Framework: An Overview

The architecture of the MoSCoE framework, illustrated in Figure 5.1, encompasses two main modules: a composition management module, which performs static analysis to identify a set of existing services that can be composed to provide some required functionality, and an execution management module, which dynamically deploys, monitors, and maintains the service compositions identified by the composition management module [34]. These two main modules are assisted by several support modules, including a semantic matchmaker that uses domain ontologies to resolve differences in terminology and a transition system generator that handles the details of translation between the transition systems used within MoSCoE and the formalisms used by Web service providers to describe their services, e.g., WSDL and BPEL.

According to the framework, the user provides a high-level and perhaps incomplete specification of the service to be provided (the *goal service*). The MoSCoE composition management module then uses a composition algorithm such as the one presented in [36] to search the available service repositories for a set of services that can work together to satisfy the goal service. If no viable composition is found, a message is sent to the user to explain where and why the composition process failed, allowing the user to reformulate the goal service and try again. If the composition process is successful, one or more possible compositions will have been identified; all possible compositions identified by the composition algorithm are passed to the execution management module for deployment. The execution management module uses a set of non-functional requirements supplied by the user to determine automatically which of the possible compositions best satisfies the given functional and non-functional requirements. It then generates the necessary BPEL code for the composition and calls the MoSCoE service execution engine to deploy the best composition. Once a Web service composition has been deployed, the execution management module monitors the performance of the composition. If the composition violates any of its requirements, the execution management module attempts to replace it with an equivalent composition [34].

Our implementation work affects all of the MoSCoE framework to some degree, since part of our work was a redesign of the core transition system data structures used to represent services within the entire MoSCoE framework. We classify the new XML input/output support that we added as part of our implementation as a separate support module because it provides support for XML representations of transition systems, restriction sets for the composition process, and mu-calculus properties for substitutability analysis. Because the execution management module is responsible for ensuring the continued reliability of service compositions in the MoSCoE framework, substitutability analysis for existing service compositions falls within its domain; as a result, our tools for performing quotienting and for creating buffer processes naturally fit into the execution management module.

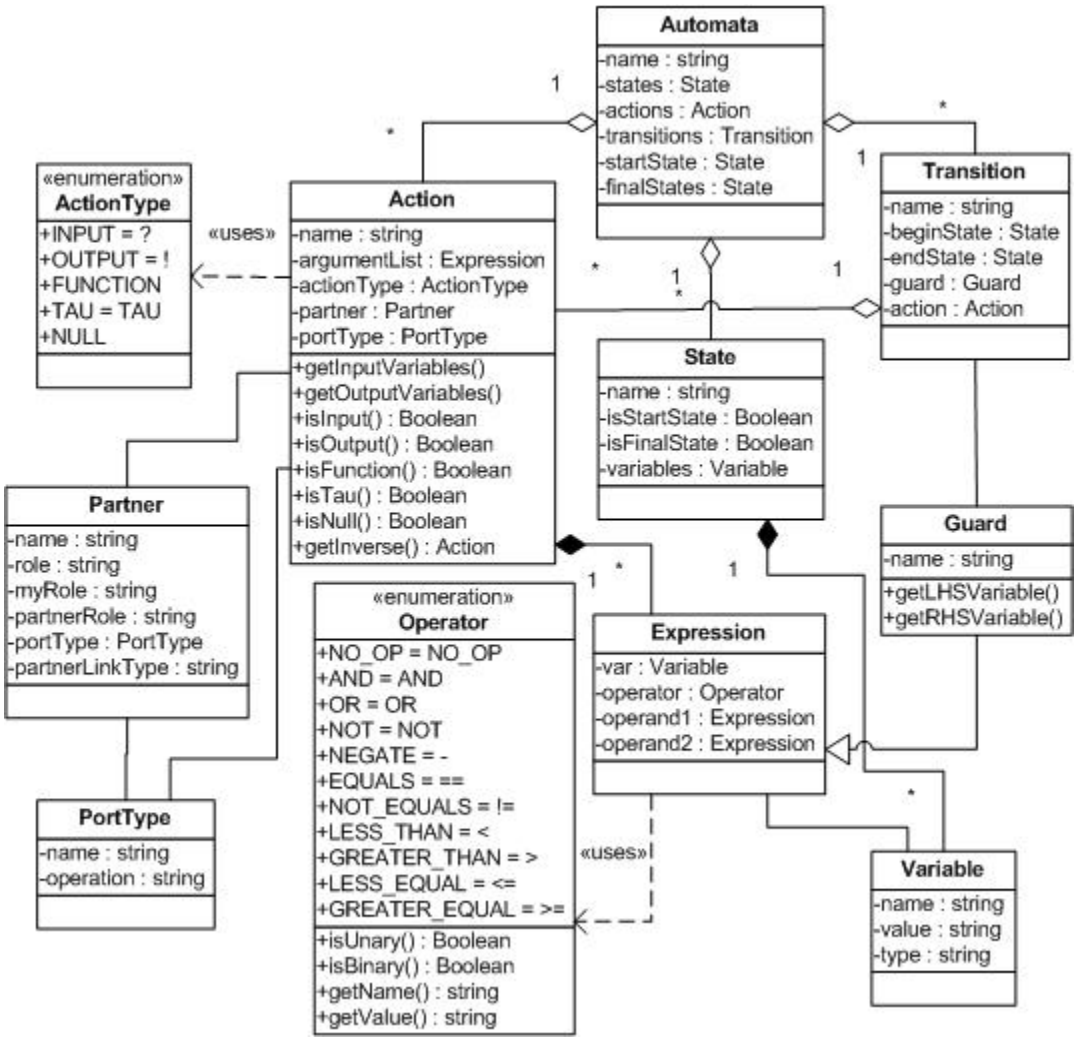


Figure 5.2 Revised core data structure for MoSCoE framework

5.2 Modifications to MoSCoE

Our tool set is designed to be compatible with the MoSCoE [36] framework for Web service composition. However, the existing MoSCoE implementation was difficult to work with because it included a number of data types and features that were not fully implemented. In addition, the graphical user interface (GUI) for MoSCoE was heavily implementation-dependent, which presented serious problems for future extension and refinement of the MoSCoE framework. Therefore, the first task in our implementation process, before writing any tools for substitutability analysis, was to revise and simplify the core data structures of MoSCoE to

provide support for both the LTS model used to represent Web services in our work and in [35] and the Symbolic Transition System (STS) model used in [36]. The resulting data structure, which is shown in Figure 5.2, is the basis of the operations performed by our tool set. While these two models are similar in most respects, there are important differences between them. For example, transitions between states of an STS may include *guards*, which are conditions that must be satisfied before the transition is executed; transitions in an LTS cannot include guards. To accommodate these differences, the data structure contains more classes and fields than strictly necessary for our purposes in this work. Our tool set ignores the contents of all fields and objects that are not part of the LTS model; it does not read or modify the contents of any such fields or objects.

Along with our revisions to the core MoSCoE data structures, we added new XML-based input and output capabilities to the MoSCoE framework. These new capabilities include an XML schema for representing Web service data, along with tools for translating MoSCoE data from XML files to Java data structures and vice versa. They replace the previous mechanisms for input and output, which depended on the existing GUI to function. Our XML schema for Web service analysis allows one or more LTSs, a restriction set for a Web service composition, and one or more mu-calculus formulas to be encoded within a single XML file. The grammar corresponding to this XML schema is presented in Figure 5.3. It should be noted that this schema is somewhat informal, as our tools do not yet use a formal XML Schema [21] document to validate input. Future work will incorporate this functionality into our tool set.

5.3 New Tools for Web Service Substitutability Analysis

In addition to the changes that we made to the MoSCoE framework, we have created a set of several tools that can be used to compute substitutability conditions for Web services and determine whether a Web service can substitute for a component of an existing Web service composition. The high-level architecture of our tool set for Web service substitutability analysis is illustrated in Figure 5.4. Our tool set consists primarily of three main components: a *substitutability analysis manager*, a *buffer creation module*, and a *quotienting module*. The

```
quotienting-data = '<quotienting-data>', [services], [restriction-set],
                   [property], '</quotienting-data>'
services         = '<services>', {service}, '</services>'
service          = '<service name="', [string], '" />', states, actions,
                   transitions, '</service>'
states           = '<states>', {state}, '</states>'
state            = '<state name="', [string], '" isStart="', tf, '" isFinal="',
                    tf, '" />'
tf               = 'true' | 'false'
actions          = '<actions>', {action}, '</actions>'
action           = '<action name="', [string], '"type="', action-type, '" />'
action-type      = '?' | '!' | 'TAU' | 'FUNC'
transitions      = '<transitions>', {transition}, '</transitions>'
transition       = '<transition name="', [string], '" startState="', state-name,
                   '" endState="', state-name, '" action="', action-name, '" />'
state-name       = string (* where the string must match the name of a state
                   defined in the states element *)
action-name      = string (* where the string must match the name of an action
                   defined in the actions element *)

restriction-set  = '<restriction-set>', {action}, '</restriction-set>'

property         = '<property>', property-parts, '</property>'
property-parts   = fixpt-formula | diamond | box | and-formula | or-formula
                   | proposition | fixpt-variable
fixpt-formula    = '<fixpt-formula type="', fixpt-type, '" variable="', string,
                   '">', property-parts, '</fixpt-formula>'
fixpt-type       = 'mu' | 'nu'
diamond          = '<diamond action="', string, action-type, '">',
                   property-parts, '</diamond>'
box              = '<box action="', string, action-type, '">',
                   property-parts, '</box>'
and-formula      = '<and-formula>', {and-or-part}, '</and-formula>'
or-formula       = '<or-formula>', {and-or-part}, '</or-formula>'
and-or-part      = '<part>', property-parts, '</part>'
proposition      = '<proposition name="', string, '" />'
fixpt-variable   = '<fixpt-variable name="', fixpt-var-name, '" />'
fixpt-var-name   = string (* where the string must match a variable declared
                   in an enclosing fixpt-formula element *)
```

Figure 5.3 Grammar for XML schema used by our tool set

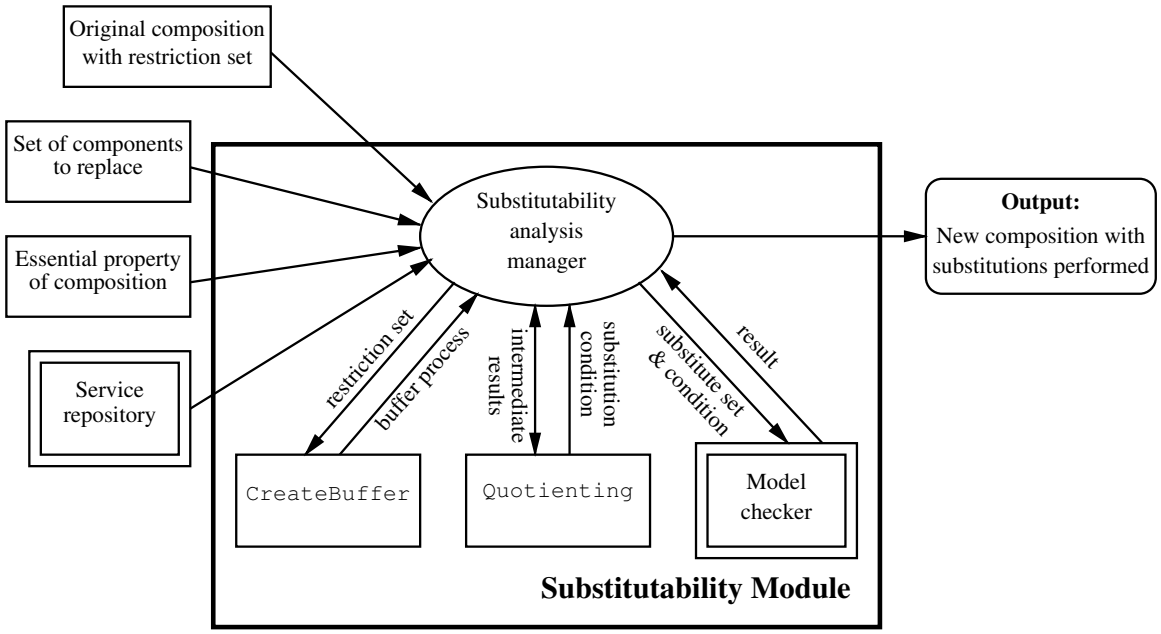


Figure 5.4 Architecture for Web service substitutability analysis tool set

user interacts directly with the substitutability analysis manager, which automates the steps of the analysis process by coordinating the use of the other tools in the tool set. The tool set may optionally include access to a tool for verifying satisfiability of mu-calculus formulas. While any such tool will suffice, we currently have not incorporated a model checking tool into our tool set; in future work, we plan to explore the use of the model checking tools XMC [38] and CWB-NC [14] within our tool set. The remaining parts displayed in Figure 5.4 are assumed to exist externally. The user may (and perhaps should) use an automated search assistant tool to identify one or more possible substitute services from a service repository, but we have omitted this tool from the architecture diagram because it is not required and its presence or absence does not change the essential structure of the system.

The flow of control within the Web service substitutability analysis tool set is illustrated by Figure 5.5. Execution begins when the user provides a Web service composition, its corresponding restriction set, and the property that it must satisfy. If the component services of the composition communicate asynchronously, then the buffer creation module is called to

form a buffer process for the composition and the quotienting module is called to quotient the property against the buffer process; if the component services communicate synchronously, these steps are unnecessary. Next, the user specifies the component of the composition that must be substituted. After this component is removed from the composition, the remainder of the composition is quotiented against the required property (or the portion of it that remains after quotienting against the buffer property). The result of this quotienting operation is the substitutability condition, which is presented to the user. If no possible substitute services were supplied by the user, then the control flow ends here. Otherwise, a model checking tool is used to verify whether each of the possible substitutes can satisfy the substitutability condition, and the results are communicated to the user. Note that if a proposed substitution is approved, another tool must be used to perform the substitution; our tool set does not perform the substitution automatically, and therefore this functionality is not included in the control flow. If the substitution is rejected, the user may try again with one or more different substitute services.

The control flow for the tool set is coordinated by the *substitutability analysis manager*. The substitutability analysis manager accepts as input the original composition, its restriction set, the component of the composition that must be replaced, and a property (specified as a mu-calculus formula) that expresses the essential functionality of the original composition and must therefore be satisfied by the new composition resulting from the substitution. A user may additionally specify one or more services that should be tested to determine whether they can substitute for the component service that is to be replaced. The substitutability analysis manager then coordinates execution of the buffer creation module, the quotienting modules, and the model checker in the appropriate order, providing inputs and storing intermediate results as needed. When finished, it provides the substitutability condition for the component to be replaced. It also informs the user whether the proposed substitute service(s), if any were tested, can correctly replace the original component in the composition.

The role of the *quotienting module* is to perform quotienting of an LTS against a mu-calculus formula according to the rules given in Figure 3.4. The quotienting module accepts

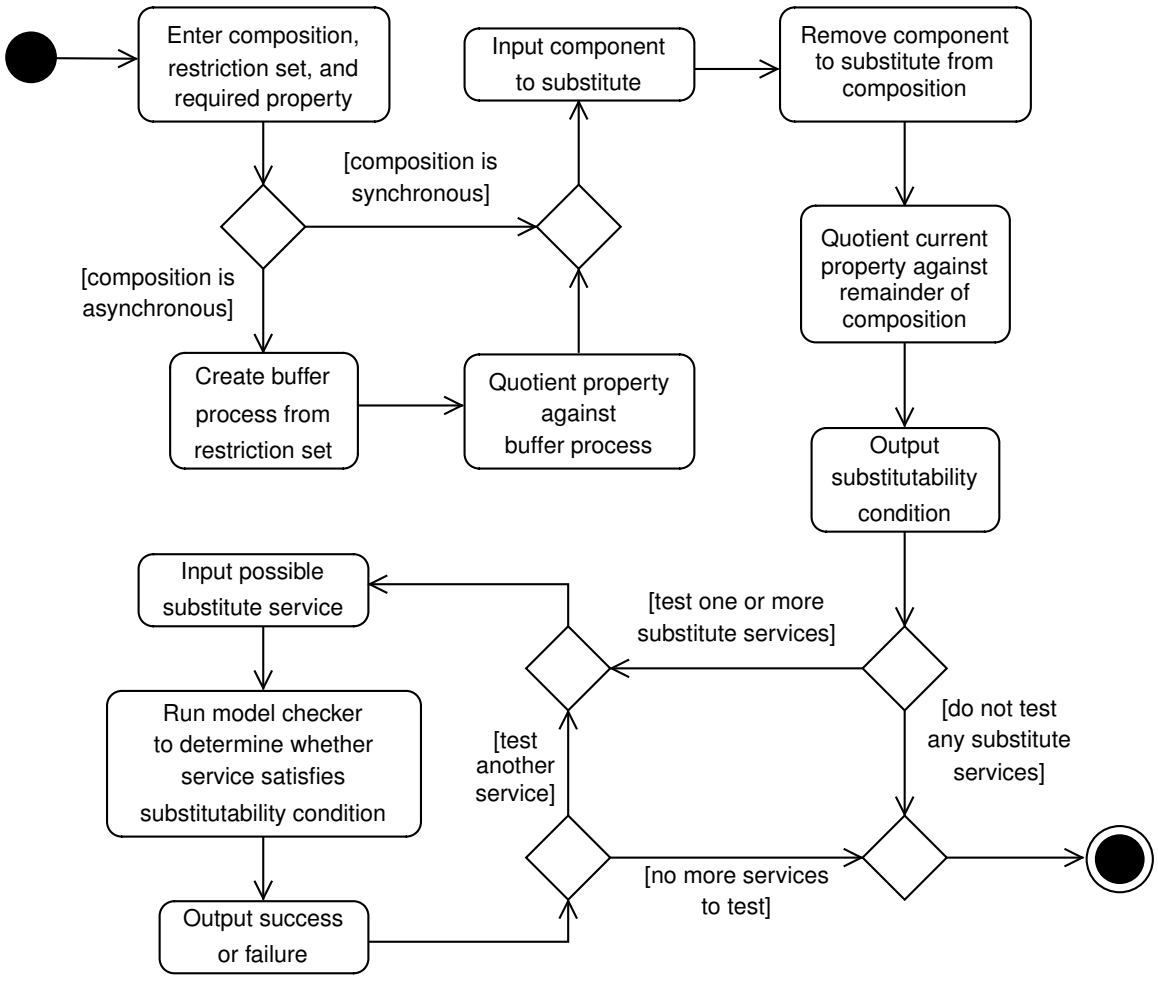


Figure 5.5 Control flow for Web service substitutability analysis tool set

input in the form of an XML input file formatted according to the schema given in Figure 5.3. It then recursively quotients the specified mu-calculus formula against the given LTS. When the quotienting process is completed, the resulting mu-calculus formula is written to a new XML file along with the LTS against which quotienting was performed. This output file can then be used immediately as an input to another tool, e.g., a model checking tool, for verifying whether the mu-calculus formula generated by the quotienting process is satisfied by a given substitute service.

The *buffer-creation module* creates a buffer process that is capable of handling asynchronous production and consumption of all actions that appear in the restriction set of a given synchronous composition. The only input required by the buffer-creation module is a restriction set encoded according to the XML schema required by our tool set; this restriction set need not be pairwise. Because the algorithm for generating the buffer process assumes that the restriction set is a pairwise restriction set as defined in Definition 7, the module first converts the given restriction set to a pairwise restriction set by removing any actions for which a corresponding inverse action cannot be found. This is done by attempting to match the name of each input action in the restriction set to the name of an output action in the set using simple string comparisons. Matched pairs of actions are allowed to remain in the pairwise restriction set; actions for which matches are not found are removed. Once the pairwise restriction set has been constructed, the buffer-creation module then executes Algorithm 1 to create the appropriate buffer process corresponding to the given restriction set. When finished, the module writes the completed buffer process LTS to a new XML file, following our XML schema for representing LTSs. This file may immediately be used as input to the MoSCoE composition tool, the quotienting module, or any other Web service analysis tool that supports our XML schema.

CHAPTER 6. CONCLUSION

6.1 Summary

Most previous work on identifying conditions for substitutability of components in a composite service makes two major assumptions: that behavioral equivalence must be established between the service being replaced and the replacement service to guarantee correctness, and that all components of a composite service must communicate synchronously. It was shown in [35] that considering the environment in which a replacement service will be deployed makes it unnecessary to require behavioral equivalence between the original service and its substitute. However, [35] continued to assume synchronous communication between component services in order to simplify the computation of substitutability conditions for Web services. This assumption prevents some acceptable substitute services from being identified.

In this thesis, we have extended the work in [35] to relax this assumption by showing that asynchronous composition of services can be reduced to synchronous composition by adding a buffer for each internal input/output action pair. We have presented a new formalism for representing compositions of asynchronously communicating Web services. We have defined a buffer process as the synchronous composition of all of the buffers created for a given asynchronous composition, and we have shown that an asynchronous Web service composition can be represented as a synchronous composition composed with such a buffer process. We have proven that our technique produces correct results, and we have defined an algorithm to design the necessary buffers for handling asynchronous composition. Finally, we have developed tools for creating buffer processes to handle asynchronous composition and for quotienting a property specified as a mu-calculus formula against a labeled transition system. These tools have been incorporated into the MoSCoE framework for Web service analysis and composition.

6.2 Future Work

Future theoretical work on substitutability of asynchronous Web services will include refinement of the communication model used to represent a given Web service composition and development of strategies for improving discovery of substitute services. Most existing work, including ours, assumes that all communication between a service and its environment is either completely asynchronous or completely synchronous. There is no "middle ground" that allows buffers to be created for only the actions that need them; either buffers are created for all actions (asynchronous) or no buffers are created (synchronous). We plan to refine our formalism to allow buffering only for input/output pairs that are required to communicate asynchronously. This refinement could potentially incorporate different levels of buffering based on a hierarchy of communication models for Web service compositions similar to the hierarchy presented by Kazhamiakin et al. in [24]. The asynchronous composition formalism presented in this thesis can be extended to incorporate such a hierarchy of communication models, and our existing tool set can be modified to include an algorithm for automatically selecting the communication model that provides the minimum sufficient set of buffers to permit each service to be used within the composition. We believe that in many cases this will increase the efficiency of the quotienting process while preserving the equivalence property of Theorem 2, which is essential to the correctness of the results.

Another topic of interest for future work is determining whether it is necessary to create a composite buffer process when determining a substitutability condition under asynchronous communication. For an asynchronous composition $Q_1 \,//\, Q_2$, instead of creating the appropriate buffer process Q_{B12} and then performing quotienting against the buffer process, it may be possible to obtain the same result by quotienting against each of the buffer process's component LTSs Q_B^a individually in sequence. This would eliminate the time required to compose the buffer process, potentially resulting in significantly faster performance when the restriction set contains more than a few actions. Suppose that $Q_1 \,//\, Q_2$ has the restriction set $R = (a?, a!, b?, b!)$ and consider the synchronous composition $(Q_1 \,||\, Q_2 \,||\, (Q_B^a \,||\, Q_B^b))\backslash R$. This composition is identical to the buffered synchronous composition $(Q_1 \,||\, Q_2 \,||\, Q_{B12})$, because

$Q_{B12} = Q_B^a \mid\mid Q_B^b$ as defined in Definition 6; therefore, Theorem 2 holds. To show that quotienting against individual simple buffers would have the same result as quotienting against the entire buffer process, it would be sufficient to prove a result similar to Theorem 3 showing, for example, that a service Q_1' can replace Q_1 in the asynchronous composition $(Q_1 \mathbin{//} Q_2)\backslash R$ if and only if $Q_1' \models (((\varphi \diagup_{\emptyset,\emptyset} Q_B^a) \diagup_{\emptyset,\emptyset} Q_B^b) \diagup_{\emptyset,R} Q_2)$. If this can be proven, then the next step after proving this result would be to compare the performance of this method with the performance of the method presented in this thesis.

In addition, we are currently investigating the applicability of formula graph analysis [4] to identification of replacement component services. Translating a composition's required mu-calculus properties into formula graphs may enable the discovery of possible substitute services that satisfy semantically, but not syntactically, equivalent properties; we are not aware of any existing methods for discovering such services. Using formula graphs to represent mu-calculus properties may also improve the efficiency of the quotienting operation, although further study is needed to determine whether this is the case. We have already developed a tool for transforming mu-calculus formulas into formula graphs according to the rules presented in [4], which will prove useful for future research in this area.

A significant amount of future implementation work is needed to improve the performance of the MoSCoE framework in general and our substitutability analysis tool set in particular. While the core of our tool set has already been developed and tested, additional features should be added to improve the usability of the tool set and the overall MoSCoE framework. Three major priorities that complement each other are the development of an appropriate graphical user interface for the MoSCoE framework, the creation of tools that correctly and efficiently translate BPEL and WSDL specifications of Web services into their equivalent XML representations for MoSCoE, and the continued refinement and formalization of the current informal XML schema for representing MoSCoE input and output. Because the architecture of our substitutability analysis tool set envisions access to a model checking tool for automated verification of service substitutability, we also intend to include a model checker within our tool set in the future. To that end, as mentioned in Chapter 5, we intend to explore the feasibility

of incorporating the model checking tools XMC [38] and CWB-NC [14] within our tool set. In addition, while not directly related to substitutability analysis, it will be helpful to provide support within the MoSCoE execution management module for on-the-fly replacement of a component service with a previously identified substitute service in the event that the original component fails or becomes unavailable. The eventual goal is to integrate our work with the previous contributions of [34] to allow for automatic re-composition of a composite service at runtime according to both functional and non-functional requirements for substitution.

We are also planning to explore the applicability of our tool set in practical settings. In particular, we are investigating approximate quotienting algorithms to compute substitutability conditions, which can potentially increase the efficiency of the computation without compromising the soundness of the process. We also intend to test the effectiveness of our tool set and the MoSCoE framework as a whole when applied to a benchmark or similar set of significant Web services and/or compositions. While no benchmark has yet been widely accepted throughout the Web service community, WSBen [32] is a proposed benchmark that shows some promise for this purpose. We hope that the results from this testing will provide additional impetus toward realizing the vision of MoSCoE as a comprehensive, effective framework for creating and managing Web service compositions.

BIBLIOGRAPHY

[1] Henrik Reif Andersen. Partial Model Checking (extended abstract). In *Logic in Computer Science*, pages 398–407. IEEE Computer Society, 1995.

[2] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, Version 1.1. URL: *http://www.ibm.com/developerworks/library/ws-bpel/*, 2003.

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

[4] Samik Basu and C. R. Ramakrishnan. Compositional Analysis for Verification of Parameterized Systems. *Theoretical Computer Science*, 354(2):211–229, 2006.

[5] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, Analysing and Managing Web Service Protocols. *Data and Knowledge Engineering*, 58(3):327–357, 2006.

[6] Daniela Berardi, Diego Calvanese, De Giacomo Giuseppe, Richard Hull, and Massimo Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *31st International Conference on Very Large Databases*, pages 613–624, 2005.

[7] Dirk Beyer, Arindam Chakrabarti, and Thomas Henzinger. Web Services Interfaces. In *15th World Wide Web Conference*, pages 148–159. ACM Press, 2005.

[8] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. W3C Working Group Note, World Wide Web Consortium, February 2004. URL: *http://www.w3.org/TR/ws-arch/*.

[9] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are Two Web Services Compatible? In *5th International Workshop on Technologies for E-Services*, pages 15–28. LNCS 3324, Springer-Verlag, 2004.

[10] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language, Version 1.1. W3C Recommendation, World Wide Web Consortium, September 2006. URL: *http://www.w3.org/TR/xml11/*.

[11] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation Specification: A New Approach to Design and Analysis of e-Service Composition. In *12th Intl. Conference on World Wide Web*, pages 403–410. ACM Press, 2003.

[12] Tevfik Bultan, Jianwen Su, and Xiang Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[13] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language Version 2.0 Part 1: Core Language. W3C Recommendation, World Wide Web Consortium, June 2007. URL: *http://www.w3.org/TR/wsdl20/*.

[14] CWB-NC: The Concurrency Workbench of the New Century. URL: *http://www.cs.sunysb.edu/~cwb/*, 2001.

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

[16] Eclipse. Integrated development environment (IDE) and software development kit (SDK) for Java. URL: *http://www.eclipse.org*.

[17] E. Allen Emerson. Model Checking and the Mu-Calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of

DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 185–214. American Mathematical Society Press, 1997.

[18] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, Boston, MA, USA, 2003.

[19] Xiang Fu, Tevfik Bultan, and Jianwen Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1–2):19–37, November 2004.

[20] Google Docs. Web-based software. URL: *http://docs.google.com*.

[21] XML Schema Working Group. W3C XML Schema. W3C Recommendation, World Wide Web Consortium, October 2004. URL: *http://www.w3.org/XML/Schema*.

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

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

[24] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of Communication Models in Web Service Compositions. In *15th International Conference on World Wide Web*, pages 267–276. ACM Press, 2006.

[25] Yunyao Li and H.V. Jagadish. Compatibility Determination in Web Services. In *ICEC Workshop on E-Government and Web Services*, 2003.

[26] Fangfang Liu, Liang Zhang, Yuliang Shi, Lili Lin, and Baile 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.

[27] Axel Martens, Simon Moser, Achim Gerhardt, and Karoline Funk. Analyzing Compatibility of BPEL Processes. In *International Conference on Internet and Web Applications and Services*, pages 147–155. IEEE CS Press, 2006.

[28] Massimo Mecella, Barbara Pernici, and Paolo Craca. Compatibility of e-Services in a Cooperative Multi-platform Environment. In *1st International Workshop on Technologies for e-Services*, volume 2193 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2001.

[29] Microsoft Office Live. Web-based software. URL: *http://workspace.officelive.com/en-us/*.

[30] Robin Milner. *Communication and Concurrency*. Prentice Hall, Upper Saddle River, NJ, 1989.

[31] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[32] Seog-Chan Oh and Dongwon Lee. WSBen: A Web Services Discovery and Composition Benchmark Toolkit. *Intl. Journal of Web Service Research*, 6(1):1–19, 2009.

[33] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007.

[34] Jyotishman Pathak. *Interactive and Verifiable Web Services Composition, Specification Reformulation, and Substitution*. PhD dissertation, Iowa State University, 2007.

[35] Jyotishman Pathak, Samik Basu, and Vasant Honavar. On Context-Specific Substitutability of Web Services. In *IEEE International Conference on Web Services*, pages 192–199, 2007.

[36] Jyotishman Pathak, Samik Basu, Robyn Lutz, and Vasant 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.

[37] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *3rd Intl. Conference on Web Services*, pages 293–301. IEEE Press, 2005.

[38] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrishnan. XMC: A Logic-Programming-Based Verification Toolset. In E. Allen Emerson and A. Prasad Sistla, editors, *12th Intl. Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 576–580. Springer, 2000.

[39] Streaming API for XML (StAX). A package within the standard Java platform for parsing XML as a stream. URL: *http://stax.codehaus.org/Home*.

[40] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an Approach for Web Services Substitution. In *10th Intl. Database Engineering and Applications Symposium*, pages 166–173. IEEE CS Press, 2006.

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